Учебник для вузов

ПИТЕР

С. А. Орлов Б. Я. Цилькер

Организация ЭВМ и систем

Фундаментальный курс по архитектуре и структуре современных компьютерных средств

2-е издание

ДОПУЩЕНО МИНИСТЕРСТВОМ ОБРАЗОВАНИЯ И НАУКИ РФ

ББК 32.973.23я7
УДК 004.382.7(075)
О66

Рецензенты:

Сергеев М. Б., заведующий кафедрой вычислительных систем и сетей Санкт-Петербургского государственного университета аэрокосмического приборостроения, доктор технических наук;

Яшин А. И., профессор кафедры АСОИУ Санкт-Петербургского государственного электротехнического университета.

Орлов С. А., Цилькер Б. Я.

О66 Организация ЭВМ и систем: Учебник для вузов. 2-е изд. — СПб.: Питер, 2011. — 688 с.: ил.

ISBN 978-5-49807-862-5

Учебник посвящен систематическому изложению вопросов организации структуры и функционирования вычислительных машин и систем, при этом большое внимание уделяется вопросам эффективности традиционных и перспективных решений в области компьютерной техники. Рассмотрены структура и функционирование классических фон-неймановских машин, принципы организации шин, внутренней и внешней памяти, операционных устройств и устройств управления, систем ввода-вывода. Изложены основные тенденции в архитектуре современных процессоров. Значительная часть материала посвящена идеологии построения и функционирования параллельных и распределенных вычислительных систем самых разнообразных классов. Показаны наиболее перспективные направления в области организации и архитектуры вычислительных машин и систем. В основу работы положен 30-летний университетский опыт преподавания авторами соответствующих дисциплин.

Допущено Министерством образования и науки РФ в качестве учебника для студентов высших учебных заведений, обучающихся по направлению «Информатика и вычислительная техника».

ББК 32.973.23я7
УДК 004.382.7(075)

ISBN 978-5-49807-862-5

Краткое оглавление

Оглавление

Предисловие ко второму изданию

Время бежит чрезвычайно быстро, а в такой динамической области знаний, как компьютерная техника, оно спрессовывается в мгновения. Прошло уже шесть лет с момента первого издания, и вот, уважаемый читатель, вы держите в руках второе издание учебника.

Учебная дисциплина «Организация ЭВМ и систем» играет особую роль в линейке дисциплин, посвященных различным аспектам аппаратных средств компьютерной техники, она является «ядром», интегрирующим знания многочисленных дисциплин-предшественниц и обеспечивающим базис для широкого спектра дисциплин системного и сетевого направлений.

В центре внимания первого издания учебника находилась информационная, логическая модель аппаратных средств вычислительной машины (ВМ) и системы (ВС). Первое и, пожалуй, ключевое слово в названии книги — организация. Мы стремились показать организацию аппаратных элементов и узлов для обеспечения автоматического режима вычислений как на микропрограммном, так и на программном уровне. Слово фиксирует ту точку зрения, что такая сложная, многоуровневая совокупность аппаратных средств, как ВМ, нуждается в организации и регламентации, наборе соглашений и правил. Именно организация сообщает синергетический эффект набору «аппаратных деталей», обеспечивая их согласованную работу, гармоничное взаимодействие. Эти соображения справедливы и для второго издания учебника.

Целью дисциплины «Организация ЭВМ и систем» является обучение основным принципам построения и функционирования современных вычислительных машин и вычислительных систем, привитие навыков их анализа и применения.

В государственном образовательном стандарте высшего профессионального образования содержание дисциплины «Организация ЭВМ и систем» определено следующим образом:

- основные характеристики, области применения ЭВМ различных классов;
- функциональная и структурная организация процессора;
- организация памяти ЭВМ;

- основные стадии выполнения команды;
- организация прерываний в ЭВМ;
- организация ввода-вывода;
- периферийные устройства;
- архитектурные особенности организации ЭВМ различных классов;
- параллельные системы;
- понятие о многомашинных и многопроцессорных вычислительных системах.

Все эти вопросы освещены в учебнике, который вы держите в руках, уважаемый читатель. Иными словами, данный учебник отвечает всем требованиям образовательного стандарта.

Что нового в этом издании?

Во-первых, переработан и, практически, переписан заново весь материал. При этом преследовались две цели: отразить новации компьютерной техники и накопленный педагогический опыт.

Во-вторых, изменились содержание и последовательность изложения материала. В главе 1 по-прежнему дается обзор прошлого, настоящего и будущего компьютерных средств, но мы расширили набор определяемых (и объясняемых) здесь базовых терминов. В частности, именно здесь теперь поясняются концептуальные понятия «быстродействие» и «производительность», «критерий эффективности». Глава 2 (архитектура системы команд) претерпела ряд изменений: изъята «статистическая» аргументация, но существенно расширено описание обрабатываемых данных. При обсуждении принципов работы классической ВМ (глава 3) использована улучшенная редакция языка микропрограммирования. Главы 4 и 5 теперь посвящены компонентам процессора — устройству управления и операционному устройству. В них введены новые разделы по системам прерывания программ, усовершенствованным алгоритмам и структурам умножения и деления. В главе 6 рассказывается о памяти (здесь добавлено описание самых современных тенденций развития микросхем ОЗУ, флэш-памяти, фазовой памяти, твердотельных дисков, заново написан материал по магнитным и оптическим дискам, магнитным лентам). В главе 7 (организация шин) появились новые примеры шин большого (PCI Express, HyperTransport, QPI) и малого (USB, FireWire, Bluetooth, IrDA) интерфейсов. Усовершенствована терминологическая база пояснения системы ввода/вывода (глава 8). Значительно раздвинуты рамки примеров, поясняющих архитектурные приемы борьбы за производительность процессора (глава 9): здесь и многоядерные процессоры, и гиперпотоковая обработка, и архитектуры VLIW, EPIC. Серьезному обновлению подверглись главы по вычислительным системам. Среди новаций отметим: закон Сана–Ная и метрику Карпа–Флэтта (глава 10), архитектуру памяти DSM (глава 11), топологию Бэтчера–Баньяна (глава 12), ассоциативные процессоры и ассоциативные многопроцессорные системы (глава 13), параллельные векторные системы, Constellation-системы, тенденции развития MIMD-систем (глава 14), вычислительные системы волнового фронта (глава 15).

В-третьих, добавлены приложения с описанием арифметических, логических и схемотехнических основ вычислительных машин.

В-четвертых, исправлены имевшиеся ошибки и опечатки. Увы, время от времени приходится вспоминать пословицу, что «каждая последняя ошибка является предпоследней...», поэтому авторы будут признательны всем читателям, кто заметит неточности и опечатки и сообщит о них редакции.

В заключение хочется поблагодарить всех читателей, приславших благожелательные отзывы и конструктивные замечания.

Введение

Мы живем в информационную эпоху: документы ЮНЕСКО свидетельствуют, что сейчас в информационной сфере занято больше половины населения развитых стран. Основу современных информационных технологий, их базис, составляют аппаратные средства компьютерной техники.

Современные вычислительные машины (ВМ) и системы (ВС) являются одним из самых значительных достижений научной и инженерной мысли, влияние которого на прогресс во всех областях человеческой деятельности трудно переоценить. Поэтому понятно то пристальное внимание, которое уделяется изучению ВМ и ВС в направлении «Информатика и вычислительная техника» высшего профессионального образования.

Настоящий учебник посвящен систематическому изложению принципов построения и функционирования современных и перспективных ВМ и ВС.

В основу материала положен многолетний опыт постановки и преподавания ряда дисциплин по аппаратным средствам компьютерной техники. Базовый курс «Архитектура ВМ и ВС» прослушали больше двух тысяч студентов, работающих теперь в инфраструктуре компьютерной индустрии, в самых разных странах и на самых разных континентах.

Авторы стремились к достижению трех целей:

- изложить классические основы, демонстрирующие накопленный отечественный и мировой опыт вычислительных машин и систем;
- показать последние научные и практические достижения, характеризующие динамику развития аппаратных средств компьютерной техники;
- обобщить и отразить 30-летний университетский опыт преподавания соответствующих дисциплин.

Первая глава учебника посвящена базовым положениям. Обсуждаются понятия «организация» и «архитектура» вычислительных машин и систем, уровни абстракции, на которых эти понятия могут быть раскрыты. Прослеживается эволюция ВМ и ВС как последовательности идей, предопределивших современное состояние

в области вычислительной техники. Анализируются тенденции дальнейшего развития архитектуры ВМ и ВС с учетом технологического прогресса и последних достижений в проектировании вычислительных средств.

Во второй главе дается понятие архитектуры системы команд и обсуждаются различные аспекты этой архитектуры. Рассматриваются основные виды информации, являющейся объектом обработки и хранения в ВМ и ВС. Приводятся основные способы представления такой информации: форматы, стандарты, размещение в памяти, способы доступа к данным. Представлены классификация и характеристика команд ВМ. Обсуждаются принципы выбора эффективной системы операций и системы адресации.

Третья глава является основой для понимания принципов функционирования вычислительных машин с классической фон-неймановской архитектурой. На примере гипотетической ВМ прослеживается взаимодействие узлов вычислительной машины в ходе выполнения типовых команд. Приводится описание языка микропрограммирования как средства формальной записи вычислительных процессов на уровне архитектуры ВМ.

Содержание четвертой главы — это описание принципов организации устройств управления (УУ) ВМ. Обсуждаются вопросы построения, функционирования и проектирования УУ с аппаратной логикой и программируемой логикой, а также способы ускорения их работы. Описываются основные идеи и механизмы системы прерывания программ.

Предметом внимания пятой главы являются операционные устройства ВМ. Рассматриваются жесткие и магистральные структуры операционных устройств, их организация и классификация, способы реализации в ВМ основных арифметических и логических операций с учетом обработки данных в различных формах представления и форматах. Наряду со «стандартными» способами реализации арифметических операций обсуждаются и такие алгоритмы, использование которых ведет к существенному ускорению вычислений.

В шестой главе определены принципы и средства, используемые при построении системы памяти ВМ. Поясняется концепция иерархического построения памяти. В первой части главы обсуждаются вопросы организации внутренней памяти с учетом ее реализации на базе полупроводниковых запоминающих устройств (ЗУ): структура памяти с произвольным доступом, матричная организация микросхем ЗУ, основные типы оперативных и постоянных запоминающих устройств. Описываются архитектурные аспекты внутренней памяти ВМ — модульное построение, конвейеризация, расслоение, обнаружение и исправление ошибок. Значительное внимание уделено принципам организации и функционирования кэш-памяти. Обсуждаются вопросы виртуализации памяти ВМ, методы и средства защиты памяти от несанкционированного доступа. Вторая часть главы содержит краткую характеристику различных типов внешних запоминающих устройств, включая магнитные и оптические дисковые ЗУ, флэш-память, фазовую память, твердотельные диски, магнитоленточные запоминающие устройства. Приводится классификация и описание массивов магнитных дисков с избыточностью (RAID).

Седьмая глава отведена принципам организации системы коммуникаций между элементами структуры ВМ. Дается понятие шины «процессор-память», шины ввода/вывода, системной шины. Рассматриваются методы повышения эффективности шин, способы синхронизации и арбитража устройств, подключенных к шине. Обсуждаются популярные и перспективные шины большого и малого интерфейсов.

Восьмая глава учебника посвящена вопросам организации систем ввода/вывода (СВВ). Рассматриваются способы организации ввода/вывода (ввод/вывод с опросом, ввод/вывод по прерываниям, прямой доступ к памяти) и их влияние на эволюцию принципов построения СВВ. Описываются особенности систем ввода/вывода больших универсальных ВМ с их концепцией процессоров (каналов) ввода/вывода.

В девятой главе излагаются вопросы, касающиеся архитектуры процессоров вычислительных машин. Дается понятие конвейера команд, обсуждаются принципы организации такого конвейера и проблемы, возникающие при его реализации. Особое внимание уделяется конфликтам в конвейере команд и способам борьбы с ними. Поясняется концепция суперконвейеризации, организация суперскалярных процессоров, гиперпотоковая обработка. Рассматривается проблема семантического разрыва и способы его преодоления в ВМ с архитектурами CISC и RISC. Глава завершается изложением концепций процессоров с архитектурой VLIW и EPIC, а также многоядерных процессоров.

Десятая глава предваряет вторую часть учебника, посвященную вопросам построения вычислительных систем, реализующих концепцию распараллеливания вычислений. Излагается теоретический базис таких вычислений. Приводится схема классификации параллельных вычислительных систем.

В одиннадцатой главе рассматриваются два основных принципа организации памяти ВС: разделяемая (совместно используемая) память и распределенная память. Рассказывается об особенностях различных моделей как той, так и другой памяти. Значительное внимание в главе уделено вопросам когерентности кэш-памяти.

Двенадцатая глава содержит достаточно подробный обзор топологии сетей межсоединений, связывающих между собой компоненты вычислительных систем.

В тринадцатой главе сосредоточен материал по системам, которые согласно классификации Флинна можно отнести к классу SIMD. Несмотря на достаточную расплывчатость границ того или иного класса, допускаемую классификацией в схеме Флинна, в учебнике к SIMD-системам отнесены: векторные и векторно-конвейерные ВС, матричные ВС, ассоциативные ВС, вычислительные системы с систолической структурой.

В четырнадцатой главе рассматриваются системы класса MIMD: симметричные мультипроцессорные системы (SMP), кластерные ВС, системы с массовым параллелизмом (MPP), ВС на базе транспьютеров, системы с неоднородным доступом к памяти.

В пятнадцатой главе описываются системы с нетрадиционным способом управления вычислениями: потоковые (статические, динамические) и мультипотоковые ВС, системы с обработкой по принципу волнового фронта, а также вычислительные системы с управлением по запросу.

Учебник предназначен для студентов инженерного, бакалаврского и магистерского уровней компьютерных специальностей, может быть полезен аспирантам, преподавателям и разработчикам вычислительных машин и систем.

Вот, пожалуй, и все. Насколько удалась эта работа — судить вам, уважаемый читатель.

Благодарности

Прежде всего наши слова искренней любви родителям.

Самые теплые слова благодарности нашим семьям, родным, любимым и близким людям. Без их долготерпения, внимания, поддержки, доброжелательности и сердечной заботы эта книга никогда бы не была написана.

Выход в свет данной работы был бы невозможен вне творческой атмосферы, бесчисленных вопросов и положительной обратной связи, которую создавали наши многочисленные студенты.

Авторы искренне признательны всем талантливым сотрудникам издательства «Питер».

И, конечно, огромная признательность нашим коллегам, общение с которыми поддерживало огонь творчества, и нашим учителям, давшим базис образования, подкрепляемого нами всю жизнь.

От издательства

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

Мы будем рады узнать ваше мнение!

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

Глава 1

Становление и эволюция цифровой вычислительной техники

Изучение любого вопроса принято начинать с договоренностей о терминологии. В нашем случае определению подлежат понятия *вычислительная машина* (ВМ) и *вычислительная система* (ВС). Сразу же оговорим, что предметом рассмотрения будут исключительно цифровые машины и системы, то есть устройства, оперирующие дискретными величинами. В литературе можно найти множество самых различных определений терминов «вычислительная машина» и «вычислительная система». Причина такой терминологической неопределенности кроется в невозможности дать удовлетворяющее всех четкое определение, достойное роли стандарта. Любая из известных формулировок несет в себе стремление авторов отразить наиболее существенные, по их мнению, моменты, в силу чего не может быть всеобъемлющей. Не отдавая предпочтения ни одной из известных формулировок терминов «вычислительная машина» и «вычислительная система», воспользуемся определениями из международного стандарта ISO/IEC 2382/1-93 и государственного стандарта ГОСТ 15971-90, условившись уточнять их смысловое наполнение по мере необходимости.

Вычислительная машина (ВМ) — совокупность технических средств, создающая возможность проведения обработки информации (данных) и получение результата в необходимой форме. Под техническими средствами понимают все оборудование, предназначенное для автоматизированной обработки данных. Как правило, в состав ВМ входит и системное программное обеспечение.

ВМ, основные функциональные устройства которой выполнены на электронных компонентах, называют электронной вычислительной машиной (ЭВМ).

В свою очередь, *вычислительную систему* (ВС) стандарт ISO/IEC 2382/1-93 определяет как одну или несколько вычислительных машин, периферийное оборудование и программное обеспечение, которые выполняют обработку данных.

Таким образом, формально отличие ВС от ВМ выражается в количестве вычислительных средств. Множественность этих средств позволяет реализовать в ВС параллельную обработку. С другой стороны, современные вычислительные машины

также обладают определенными возможностями распараллеливания вычислительного процесса. Иными словами, грань между ВМ и ВС часто бывает весьма расплывчатой, что дает основание там, где это целесообразно, рассматривать ВМ как одну из реализаций ВС. И напротив, вычислительные системы часто строятся из традиционных ВМ и их частей, поэтому многие из положений, относящихся к ВМ, могут быть распространены и на ВС.

И, наконец, заключительное замечание. Специфика главы вынуждает использовать в ней многие понятия, полное содержание которых станет ясным лишь после изучения последующих разделов книги. Там, где это возможно, пояснения будут даваться по ходу изложения (правда, в упрощенном виде). В любом случае, для получения полной картины к материалу данной главы имеет смысл еще раз вернуться после ознакомления со всем учебником.

Определение понятий «организация» и «архитектура»

Для конкретной вычислительной машины, как и для любой сложной системы, характерна определенная организация.

Термин «организация» происходит от латинского *organizo* (сообщаю стройный вид, устраиваю) и имеет два значения:

1) внутренняя упорядоченность, согласованность, взаимодействие частей целого;
2) совокупность процессов или действий, ведущих к образованию и совершенствованию взаимосвязей между частями целого.

В вычислительной технике рассматривают два вида организации ВМ и систем, которые определяются двумя взглядами на их функционирование и построение: взглядом пользователя и взглядом разработчика.

Для пользователя важен набор функций и услуг ВМ, которые обеспечивают эффективное решение его задач. Его не интересуют вопросы технической реализации этих функций и принятые технические решения. Поэтому при проектировании ВМ в первую очередь создается ее *абстрактная модель*, описывающая функциональные возможности машины и предоставляемые ею услуги, то есть ее функциональную организацию.

Функциональная организация ВМ — абстрактная модель совокупности функциональных возможностей и услуг, призванных удовлетворить потребности пользователей. Эти возможности и услуги обеспечивают всю последовательность действий пользователя: кодирование исходных данных, программирование, ввод данных и программ, управление ходом обработки данных, вывод результатов и их документирование.

В свою очередь, разработчик должен создать техническую реализацию функций, предусмотренных абстрактной моделью вычислительной машины, на основе реальных физических средств: элементов, узлов, блоков и устройств.

Структурная организация ВМ — это физическая модель, которая устанавливает состав, порядок и принципы взаимодействия основных функциональных частей

машины. Графическим отображением структурной организации является структурная схема[1]. В инженерной практике применяется более краткий термин, соответствующий структурной схеме — *структура*[2].

Итак, функциональная организация позволяет получить абстрактный «портрет» ВМ — в нем фиксируются функциональные возможности машины, но эти возможности не поддерживаются материально (физическими средствами). Структурная организация обеспечивает материализацию функциональной организации ВМ.

Термин «функциональная организация» сформулирован российскими учеными. В международном пространстве он трансформировался в термин «архитектура вычислительной машины».

Впервые термин «архитектура вычислительной машины» (computer architecture) был употреблен фирмой IBM при разработке машин семейства IBM 360 для описания тех средств, которыми может пользоваться программист, составляя программу на уровне машинных команд. Подобную трактовку называют «узкой», так как охватывает она лишь часть вопросов, описывающих функциональные возможности: перечень и формат команд, формы представления данных, механизмы ввода/вывода, способы адресации памяти и т. п.

В государственном стандарте 15971-90 под архитектурой вычислительной машины понимается концептуальная структура ВМ, определяющая проведение обработки информации и включающая методы преобразования информации в данные и принципы взаимодействия технических средств и программного обеспечения.

Международный стандарт ISO/IEC 2382/1-93 дает такое же определение, используя следующую формулировку: это логическая структура и функциональные характеристики ВМ, включая взаимосвязи между ее аппаратными и программными компонентами.

В нашем учебнике будем ориентироваться на стандартные определения термина «архитектура».

Уровни детализации структуры вычислительной машины

Вычислительная машина как законченный объект являет собой плод усилий специалистов в самых различных областях человеческих знаний. Каждый специалист рассматривает вычислительную машину с позиций стоящей перед ним задачи, абстрагируясь от несущественных, по его мнению, деталей. В табл. 1.1 перечислены специалисты, принимающие участие в создании ВМ, и круг вопросов, входящих в их компетенцию.

Круг вопросов, рассматриваемых в данном учебнике, по большей части, относится к компетенции системного архитектора и охватывает различные степени

---

[1] Структурные схемы применяют также и для описания функциональной организации.

[2] Международный стандарт ISO/IEC 2382/1-93 определяет, что структура задает отношения между элементами системы.

детализации ВМ и ВС. В принципе, таких уровней может быть достаточно много, однако сложившаяся практика ограничивает их число четырьмя уровнями (рис. 1.1).

Таблица 1.1. Распределение функций между разработчиками вычислительной машины

Специалист	Круг вопросов
Производитель полупроводниковых материалов	Материал для интегральных микросхем (легированный кремний, диоксид кремния и т. п.)
Разработчик электронных схем	Электронные схемы узлов ВМ (разработка и анализ)
Разработчик интегральных микросхем	Сверхбольшие интегральные микросхемы (схемы электронных элементов, их размещение на кристалле)
Системный архитектор	Архитектура и организация вычислительной машины (устройства и узлы, система команд и т. п.)
Системный программист	Операционная система, компиляторы
Теоретик	Алгоритмы, абстрактные структуры данных

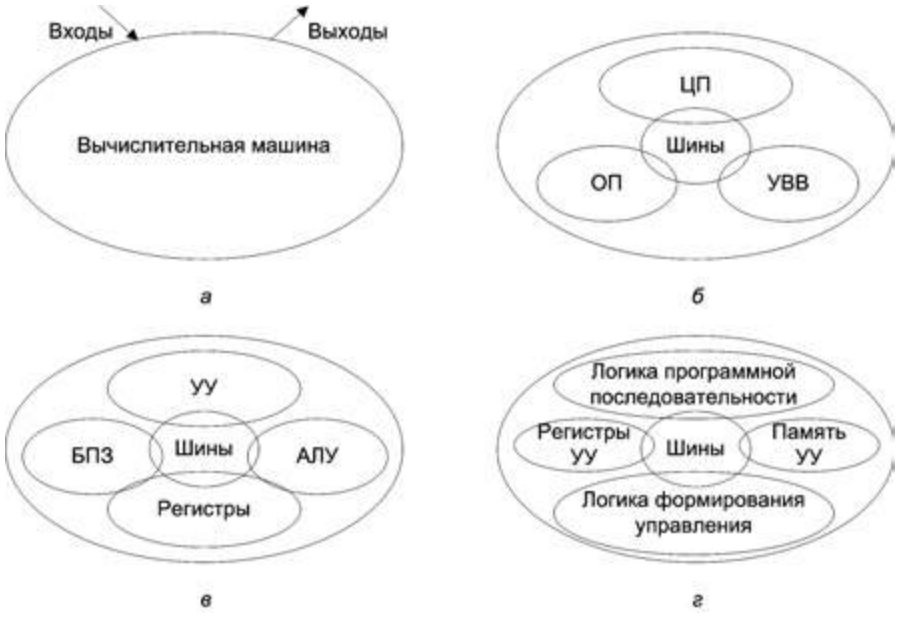


Рис. 1.1. Уровни детализации вычислительной машины: *а* — уровень «черного ящика»; *б* — уровень общей архитектуры; *в* — уровень архитектуры центрального процессора; *г* — уровень архитектуры устройства управления

На первом уровне вычислительная машина рассматривается как устройство, способное хранить и обрабатывать информацию, а также обмениваться данными с внешним миром (см. рис. 1.1, *а*). ВМ представляется «черным ящиком», который может быть подключен к коммуникационной сети и к которому, в свою очередь, могут подсоединяться периферийные устройства.

Уровень общей архитектуры (см. рис. 1.1, *б*) предполагает представление ВМ в виде четырех составляющих: центрального процессора (ЦП), основной памяти (ОП), устройства ввода/вывода (УВВ) и системы шин.

На третьем уровне детализируется каждое из устройств второго уровня. Для примера взят центральный процессор (см. рис. 1.1, *в*). В простейшем варианте в нем можно выделить:

- арифметико-логическое устройство (АЛУ), обеспечивающее обработку целых чисел;
- блок обработки чисел в формате с плавающей запятой (БПЗ);
- регистры процессора, использующиеся для краткосрочного хранения команд, данных и адресов;
- устройство управления (УУ), обеспечивающее совместное функционирование устройств ВМ;
- внутренние шины.

На четвертом уровне детализируются элементы третьего уровня. Так, на рис. 1.1, *г* раскрыта структура устройства управления. УУ представлено в виде четырех составляющих:

- логики программной последовательности — электронных схем, обеспечивающих выполнение команд программы в последовательности, предписываемой программой;
- регистров и дешифраторов устройства управления;
- управляющей памяти;
- логики формирования управления, генерирующей все необходимые управляющие сигналы.

Эволюция средств автоматизации вычислений

Попытки облегчить, а в идеале автоматизировать процесс вычислений имеют давнюю историю, насчитывающую более 5000 лет. С развитием науки и технологий средства автоматизации вычислений непрерывно совершенствовались. Современное состояние вычислительной техники (ВТ) являет собой результат многолетней эволюции.

В традиционной трактовке эволюцию вычислительной техники представляют как последовательную смену поколений ВТ. Появление термина «поколение» относится к 1964 году, когда фирма IBM выпустила серию компьютеров IBM 360, назвав эту серию «компьютерами третьего поколения». В стандарте ГОСТ 15971-90 дано следующее определение термина:

«Поколение вычислительных машин — это классификационная группа ВМ, объединяющая ВМ по используемой технологии реализации ее устройств, а также по уровню развития функциональных свойств и программного обеспечения и характеризующая определенный период в развитии промышленности средств вычислительной техники».

Одной из идей развития функциональных свойств принято считать концепцию вычислительной машины с хранимой в памяти программой, сформулированную Джоном фон Нейманом. Взяв ее за точку отсчета, историю развития ВТ можно представить в виде трех этапов:

- донеймановского периода;
- эры вычислительных машин и систем с фон-неймановской архитектурой;
- постнеймановской эпохи — эпохи параллельных и распределенных вычислений, где наряду с традиционным подходом все большую роль начинают играть отличные от фон-неймановских принципы организации вычислительного процесса.

Значительно большее распространение, однако, получила привязка поколений к смене технологий. Принято говорить о «механической» эре (нулевое поколение) и последовавших за ней пяти поколениях ВС [28, 152]. Первые четыре поколения традиционно связывают с элементной базой вычислительных систем: электронные лампы, полупроводниковые приборы, интегральные схемы малой степени интеграции (ИМС), большие (БИС), сверхбольшие (СБИС) и ультрабольшие (УБИС) интегральные микросхемы. Пятое поколение в общепринятой интерпретации ассоциируют не столько с новой элементной базой, сколько с интеллектуальными возможностями ВС. Работы по созданию ВС пятого поколения велись в рамках четырех достаточно независимых программ, осуществлявшихся учеными США, Японии, стран Западной Европы и стран Совета Экономической Взаимопомощи. Ввиду того, что ни одна из программ не привела к ожидаемым результатам, разговоры о ВС пятого поколения понемногу утихают. Трактовка пятого поколения явно выпадает из «технологического» принципа. С другой стороны, причисление всех ВС на базе сверхбольших интегральных схем (СБИС) к четвертому поколению не отражает принципиальных изменений в архитектуре ВС, произошедших за последние годы. Чтобы в какой-то мере проследить роль таких изменений, воспользуемся несколько отличной трактовкой, предлагаемой в [125]. В работе выделяется шесть поколений ВС. Попытаемся кратко охарактеризовать каждое из них, выделяя наиболее значимые события.

Нулевое поколение (1492–1945)

Для полноты картины упомянем два события, произошедшие до нашей эры: первые счеты — абак, изобретенные в древнем Вавилоне за 3000 лет до н. э., и их более «современный» вариант с косточками на проволоке, появившийся в Китае примерно за 500 лет также до н. э.

«Механическая» эра (нулевое поколение) в эволюции ВТ связана с механическими, а позже — электромеханическими вычислительными устройствами. Основным элементом механических устройств было зубчатое колесо. Начиная с XX века роль базового элемента переходит к электромеханическому реле. Не умаляя значения многих идей «механической» эры, необходимо отметить, что ни одно из созданных устройств нельзя с полным основанием назвать вычислительной машиной в современном ее понимании. Чтобы подчеркнуть это, вместо термина «вычислительная машина» будем использовать такие слова, как «вычислитель», «калькулятор» и т. п.

Хронология основных событий «механической» эры выглядит следующим образом.

1492 год. В одном из своих дневников Леонардо да Винчи приводит рисунок тринадцатиразрядного десятичного суммирующего устройства на основе зубчатых колес.

1623 год. Вильгельм Шиккард (Wilhelm Schickard, 1592–1635), профессор университета Тюбингена, разрабатывает устройство на основе зубчатых колес («считающие часы») для сложения и вычитания шестиразрядных десятичных чисел. Было ли устройство реализовано при жизни изобретателя, достоверно неизвестно, но в1960 году оно было воссоздано и проявило себя вполне работоспособным.

1642 год. Блез Паскаль (Blaise Pascal, 1623–1663) представляет «Паскалин» — первое реально осуществленное и получившее известность механическое цифровое вычислительное устройство. Прототип устройства суммировал и вычитал пятиразрядные десятичные числа. Паскаль изготовил более десяти таких вычислителей, причем последние модели оперировали числами длиной в восемь цифр.

1673 год. Готфрид Вильгельм Лейбниц (Gottfried Wilhelm Leibniz, 1646–1716) создает «пошаговый вычислитель» — десятичное устройство для выполнения всех четырех арифметических операций над 12-разрядными десятичными числами. Результат умножения представлялся 16 цифрами. Помимо зубчатых колес, в устройстве использовался новый элемент — ступенчатый валик.

1786 год. Немецкий военный инженер Иоганн Мюллер (Johann Mueller, 1746–1830) выдвигает идею «разностной машины» — специализированного калькулятора для табулирования логарифмов, вычисляемых разностным методом. Калькулятор, построенный на ступенчатых валиках Лейбница, получился достаточно небольшим (13 см в высоту и 30 см в диаметре), но при этом мог выполнять все четыре арифметических действия над 14-разрядными числами.

1801 год. Жозеф Мария Жаккард (Joseph-Marie Jacquard, 1752–1834) строит ткацкий станок с программным управлением, программа работы которого задается с помощью комплекта перфокарт.

1832 год. Английский математик Чарльз Бэббидж (Charles Babbage, 1792–1871) создает сегмент разностной машины, оперирующий шестиразрядными числами и разностями второго порядка. Разностная машина Бэббиджа по идее аналогична калькулятору Мюллера.

1834 год. Пер Георг Шутц (Per George Scheutz, 1785–1873) из Стокгольма, используя краткое описание проекта Бэббиджа, создает из дерева небольшую разностную машину.

1836 год. Бэббидж разрабатывает проект «аналитической машины». Проект предусматривает три считывателя с перфокарт для ввода программ и данных, память (по Бэббиджу — «склад») на пятьдесят 40-разрядных чисел, два аккумулятора для хранения промежуточных результатов. В программировании машины предусмотрена концепция условного перехода. В проект заложен также и прообраз микропрограммирования — содержание инструкций предполагалось задавать путем позиционирования металлических штырей в цилиндре с отверстиями. По оценкам автора, суммирование должно было занимать 3 с, а умножение и деление — 2–4 мин.

1843 год. Георг Шутц совместно с сыном Эдвардом (Edvard Scheutz, 1821–1881) строят разностную машину с принтером для работы с разностями третьего порядка.

1871 год. Бэббидж создает прототип одного из устройств своей аналитической машины — «мельницу» (так он окрестил то, что сейчас принято называть центральным процессором), а также принтер.

1885 год. Дорр Фельт (Dorr E. Felt, 1862–1930) из Чикаго строит свой «комптометр» — первый калькулятор, где числа вводятся нажатием клавиш.

1890 год. Результаты переписи населения в США обрабатываются с помощью перфокарточного табулятора, созданного Германом Холлеритом (Herman Hollerith, 1860–1929) из Массачусетсского технологического института.

1892 год. Вильям Барроуз (William S. Burroughs, 1857–1898) предлагает устройство, схожее с калькулятором Фельта, но более надежное, и от этого события берет старт индустрия офисных калькуляторов.

1937 год. Джорж Стибитц (George Stibitz, 1904–1995) из Bell Telephone Laboratories демонстрирует первый однобитовый двоичный вычислитель на базе электромеханических реле.

1937 год. Алан Тьюринг (Alan M. Turing, 1912–1954) из Кембриджского университета публикует статью, в которой излагает концепцию теоретической упрощенной вычислительной машины, в дальнейшем получившей название машины Тьюринга.

1938 год. Клод Шеннон (Claude E. Shannon, 1916–2001) публикует статью о реализации символической логики на базе реле.

1938 год. Немецкий инженер Конрад Цузе (Konrad Zuse, 1910–1995) строит механический программируемый вычислитель Z1 с памятью на 1000 битов. В последнее время Z1 все чаще называют первым в мире компьютером.

1939 год. Джордж Стибитц и Сэмюэль Вильямс (Samuel Williams, 1911–1977) представили Model I — калькулятор на базе релейной логики, управляемый с помощью модифицированного телетайпа, что позволило подключаться к калькулятору по телефонной линии. Более поздние модификации допускали также определенную степень программирования.

1940 год. Следующая работа Цузе — электромеханическая машина Z2, основу которой составляла релейная логика, хотя память, как и в Z1, была механической.

1941 год. Цузе создает электромеханический программируемый вычислитель Z3. Вычислитель содержит 2600 электромеханических реле. Z3 — это первая попытка реализации принципа программного управления, хотя и не в полном объеме (в общепринятом понимании этот принцип еще не был сформулирован). В частности, не предусматривалась возможность условного перехода. Программа хранилась на перфоленте. Емкость[1] памяти составляла 64 22-битовых слова. Операция умножения занимала 3–5 с.

[1] Емкость характеризует количество элементов данных, которое одновременно может храниться в памяти.

1943 год. Группа ученых Гарвардского университета во главе с Говардом Айкеном (Howard Aiken, 1900–1973) разрабатывает вычислитель ASCC Mark I (Automatic Sequence-Controlled Calculator Mark I) — первый программно управляемый вычислитель, получивший широкую известность. Длина устройства составила 18 м, а весило оно 5 т. Машина состояла из множества вычислителей, обрабатывающих свои части общей задачи под управлением единого устройства управления. Команды считывались с бумажной перфоленты и выполнялись в порядке считывания. Данные считывались с перфокарт. Вычислитель обрабатывал 23-разрядные числа, при этом сложение занимало 0,3 с, умножение — 4 с, а деление — 10 с.

1945 год. Цузе завершает Z4 — улучшенную версию вычислителя Z3. По архитектуре у Z4 очень много общих черт с современными ВМ: память и процессор представлены отдельными устройствами, процессор может обрабатывать числа с плавающей запятой и, в дополнение к четырем основным арифметическим операциям, способен извлекать квадратный корень. Программа хранится на перфоленте и считывается последовательно.

Не умаляя важности каждого из перечисленных фактов, в качестве важнейшего момента «механической» эпохи все-таки выделим аналитическую машину Чарльза Бэббиджа и связанные с ней идеи.

Первое поколение (1937–1953)

На роль первой в истории электронной вычислительной машины в разные периоды претендовало несколько разработок. Общим у них было использование схем на базе электронно-вакуумных ламп вместо электромеханических реле. Предполагалось, что электронные ключи будут значительно надежнее, поскольку в них отсутствуют движущиеся части, однако технология того времени была настолько несовершенной, что по надежности электронные лампы оказались ненамного лучше, чем реле. Однако у электронных компонентов имелось одно важное преимущество: выполненные на них ключи могли переключаться примерно в тысячу раз быстрее своих электромеханических аналогов.

Первой электронной вычислительной машиной чаще всего называют специализированный калькулятор ABC (Atanasoff–Berry Computer). Разработан он был в период с 1939 по 1942 год профессором Джоном Атанасовым (John V. Atanasoff, 1903–1995) совместно с аспирантом Клиффордом Берри (Clifford Berry, 1918–1963) и предназначался для решения системы линейных уравнений (до 29 уравнений с 29 переменными). ABC обладал памятью на 50 слов длиной 50 битов, а запоминающими элементами служили конденсаторы с цепями регенерации. В качестве вторичной памяти использовались перфокарты, где отверстия не перфорировались, а прожигались. ABC стал считаться первой электронной ВМ, после того как судебным решением были аннулированы патенты создателей другого электронного калькулятора — ENIAC. Необходимо все же отметить, что ни ABC, ни ENIAC не являются вычислительными машинами в современном понимании этого термина и их правильней классифицировать как калькуляторы.

Вторым претендентом на первенство считается вычислитель Colossus, построенный в 1943 году в Англии в местечке Bletchley Park близ Кембриджа. Изобретателем

машины был профессор Макс Ньюмен (Max Newman, 1987–1984), а изготовил его Томми Флауэрс (Tommy Flowers, 1905–1998). Colossus был создан для расшифровки кодов немецкой шифровальной машины «Лоренц Шлюссель-цузат-40». В состав команды разработчиков входил также Алан Тьюринг. Машина была выполнена в виде восьми стоек высотой 2,3 м, а общая длина ее составляла 5,5 м. В логических схемах машины и в системе оптического считывания информации использовалось 2400 электронных ламп, главным образом тиратронов. Информация считывалась с пяти вращающихся длинных бумажных колец со скоростью 5000 символов/с.

Наконец, третий кандидат на роль первой электронной ВМ — уже упоминавшийся программируемый электронный калькулятор общего назначения ENIAC (Electronic Numerical Integrator and Computer — электронный цифровой интегратор и вычислитель). Идея калькулятора, выдвинутая в 1942 году Джоном Мочли (John J. Mauchly, 1907–1980) из университета Пенсильвании, была реализована им совместно с Преспером Эккертом (J. Presper Eckert, 1919–1995) в 1946 году. С самого начала ENIAC активно использовался в программе разработки водородной бомбы. Машина эксплуатировалась до 1955 года и применялась для генерирования случайных чисел, предсказания погоды и проектирования аэродинамических труб. ENIAC весил 30 тонн, содержал 18 000 радиоламп, имел размеры 2,5 × 30 м и обеспечивал выполнение 5000 сложений и 360 умножений в секунду. Использовалась десятичная система счисления. Программа задавалась схемой коммутации триггеров на 40 наборных полях. Когда все лампы работали, инженерный персонал мог настроить ENIAC на новую задачу, вручную изменив подключение 6000 проводов. При пробной эксплуатации выяснилось, что надежность машины чрезвычайно низка — поиск неисправностей занимал от нескольких часов до нескольких суток. По своей структуре ENIAC напоминал механические вычислительные машины. 10 триггеров соединялись в кольцо, образуя десятичный счетчик, который исполнял роль счетного колеса механической машины. Десять таких колец плюс два триггера для представления знака числа представляли запоминающий регистр. Всего в ENIAC было 20 таких регистров. Система переноса десятков в накопителях была аналогична предварительному переносу в машине Бэббиджа.

При всей важности каждой из трех рассмотренных разработок основное событие, произошедшее в этот период, связано с именем Джона фон Неймана. Американский математик Джон фон Нейман (John von Neumann, 1903–1957) принял участие в проекте ENIAC в качестве консультанта. Еще до завершения ENIAC Эккерт, Мочли и фон Нейман приступили к новому проекту — EDVAC, главной особенностью которого стала идея *хранимой в памяти программы*.

Технология программирования в рассматриваемый период была еще на зачаточном уровне. Первые программы составлялись в машинных кодах — числах, непосредственно записываемых в память ВМ. Лишь в 50-х годах началось использование языка ассемблера, позволявшего вместо числовой записи команд использовать символьную их нотацию, после чего специальной программой, также называемой ассемблером, эти символьные обозначения транслировались в соответствующие коды.

Несмотря на свою примитивность, машины первого поколения оказались весьма полезными для инженерных целей и в прикладных науках. Так, Атанасофф подсчитал, что решение системы из восьми уравнений с восемью переменными с помощью популярного тогда электромеханического калькулятора Маршана заняло бы восемь часов. В случае же 29 уравнений с 29 переменными, с которыми калькулятор ABC справлялся менее чем за час, устройство с калькулятором Маршана затратило бы 381 час. С первой задачей в рамках проекта водородной бомбы ENIAC справился за 20 с, в противовес 40 часам, которые понадобились бы при использовании механических калькуляторов.

В 1947 году под руководством С. А. Лебедева начаты работы по созданию малой электронной счетной машины (МЭСМ). Эта ВМ была запущена в эксплуатацию в 1951 году и стала первой электронной ВМ в СССР и континентальной Европе.

В 1952 году Эккерт и Мочли создали первую коммерчески успешную машину UNIVAC. Именно с помощью этой ВМ было предсказано, что Эйзенхауэр в результате президентских выборов победит Стивенсона с разрывом в 438 голосов (фактический разрыв составил 442 голоса).

Также в 1952 году в опытную эксплуатацию была запущена вычислительная машина М-1 (И. С. Брук, Н. Я. Матюхин, А. Б. Залкинд). М-1 содержала 730 электронных ламп, оперативную память емкостью 256 25-разрядных слов, рулонный телетайп и обладала производительностью 15–20 операций/с. Впервые была применена двухадресная система команд. Чуть позже группой выпускников МЭИ под руководством И. С. Брука создана машина М-2 с емкостью оперативной памяти 512 34-разрядных слов и быстродействием 2000 операций/с.

В апреле 1953 года в эксплуатацию поступила самая быстродействующая в Европе ВМ БЭСМ (С. А. Лебедев). Ее быстродействие составило 8000–10 000 операций/с. Примерно в то же время выпущена ламповая ВМ «Стрела» (Ю. А. Базилевский, Б. И. Рамееев) с быстродействием 2000 операций/с.

Второе поколение (1954–1962)

Второе поколение характеризуется рядом достижений в элементной базе, структуре и программном обеспечении. Принято считать, что поводом для выделения нового поколения ВМ стали технологические изменения и, главным образом, переход от электронных ламп к полупроводниковым диодам и транзисторам со временем переключения порядка 0,3 мс.

Первой ВМ, выполненной полностью на полупроводниковых диодах и транзисторах, стала TRADIC (TRAnisitor DIgital Computer), построенная в Bell Labs по заказу военно-воздушных сил США как прототип бортовой ВМ. Машина состояла из 700 транзисторов и 10 000 германиевых диодов. За два года эксплуатации TRADIC отказали только 17 полупроводниковых элементов, что говорит о прорыве в области надежности, по сравнению с машинами на электронных лампах. Другой достойной упоминания полностью полупроводниковой ВМ стала TX-0, созданная в 1957 году в Массачусетсском технологическом институте.

Со вторым поколением ВМ ассоциируют еще одно принципиальное технологическое усовершенствование — переход от устройств памяти на базе ртутных линий

задержки к устройствам на магнитных сердечниках. В запоминающих устройствах (ЗУ) на линиях задержки данные хранились в виде акустической волны, непрерывно циркулирующей по кольцу из линий задержки, а доступ к элементу данных становился возможным лишь в момент прохождения соответствующего участка волны вблизи устройства считывания/записи. Главным преимуществом ЗУ на магнитных сердечниках стал произвольный доступ к данным, когда в любой момент доступен любой элемент данных, причем время доступа не зависит от того, какой это элемент.

Технологический прогресс дополняют важные изменения в архитектуре ВМ. Прежде всего это касается появления в составе процессора ВМ индексных регистров, что позволило упростить доступ к элементам массивов. Раньше, при циклической обработке элементов массива, необходимо было модифицировать код команды, в частности хранящийся в нем адрес элемента массива. Как следствие, в ходе вычислений коды некоторых команд постоянно изменялись, что затрудняло отладку программы. С использованием индексных регистров адрес элемента массива вычисляется как сумма адресной части команды и содержимого индексного регистра. Это позволяет обратиться к любому элементу массива, не затрагивая код команды, а лишь модифицируя содержимое индексного регистра.

Вторым принципиальным изменением в структуре ВМ стало добавление аппаратного блока обработки чисел в формате с плавающей запятой. До этого обработка вещественных чисел производилась с помощью подпрограмм, каждая из которых имитировала выполнение какой-то одной операции с плавающей запятой (сложение, умножение и т. п.), используя для этой цели обычное целочисленное арифметико-логическое устройство.

Третье значимое нововведение в архитектуре ВМ — появление в составе вычислительной машины процессоров ввода/вывода, позволяющих освободить центральный процессор от рутинных операций по управлению вводом/выводом и обеспечивающих более высокую пропускную способность тракта «память — устройства ввода/вывода» (УВВ).

Ко второму поколению относятся и две первые суперЭВМ, разработанные для ускорения численных вычислений в научных приложениях. Термин «суперЭВМ» первоначально применялся по отношению к ВМ, производительность которых на один или более порядков превосходила таковую для прочих вычислительных машин того же поколения. Во втором поколении этому определению отвечали две ВМ (правильнее сказать системы): LARC (Livermore Atomic Research Computer) и IBM 7030. Помимо прочего, в этих ВМ нашли воплощение еще две новинки: совмещение операций процессора с обращением к памяти и простейшие формы параллельной обработки данных.

Заметным событием данного периода стало появление в 1958 году машины М-20. В этой ВМ, в частности, были реализованы: частичное совмещение операций, аппаратные средства поддержки программных циклов, возможность параллельной работы процессора и устройства вывода. Оперативная память емкостью 4096 45-разрядных слов была выполнена на магнитных сердечниках.

Шестидесятые годы XX века стали периодом бурного развития вычислительной техники в СССР. За этот период разработаны и запущены в производство вычислительные машины «Урал-1», «Урал-4», «Урал-11», «Урал-14», БЭСМ-2, М-40, «Минск-1», «Минск-2», «Минск-22», «Минск-32». В 1960 году под руководством В. М. Глушкова и Б. Н. Малиновского разработана первая полупроводниковая управляющая машина «Днепр».

Наконец, нельзя не отметить значительные события в сфере программного обеспечения, а именно создание языков программирования высокого уровня: Фортрана (1956), Алгола (1958) и Кобола (1959).

Третье поколение (1963–1972)

Третье поколение ознаменовалось резким увеличением вычислительной мощности ВМ, ставшим следствием больших успехов в области архитектуры, технологии и программного обеспечения. Основные технологические достижения связаны с переходом от дискретных полупроводниковых элементов к интегральным микросхемам и началом применения полупроводниковых запоминающих устройств, начинающих вытеснять ЗУ на магнитных сердечниках. Существенные изменения произошли и в архитектуре ВМ. Это, прежде всего, микропрограммирование как эффективная техника построения устройств управления сложных процессоров, а также наступление эры конвейеризации и параллельной обработки. В области программного обеспечения определяющими вехами стали первые операционные системы и реализация режима разделения времени.

В первых ВМ третьего поколения использовались интегральные схемы с малой степенью интеграции (small-scale integrated circuits, SSI), где на одном кристалле размещается порядка 10 транзисторов. Ближе к концу рассматриваемого периода на смену SSI стали приходить интегральные схемы средней степени интеграции (medium-scale integrated circuits, MSI), в которых число транзисторов на кристалле увеличилось на порядок. К этому же времени относится повсеместное применение многослойных печатных плат. Все шире востребуются преимущества параллельной обработки, реализуемые за счет множественных функциональных блоков, совмещения во времени работы центрального процессора и операций ввода/вывода, конвейеризации потоков команд и данных.

В 1964 году Сеймур Крей (Seymour Cray, 1925–1996) построил вычислительную систему CDC 6600, в архитектуру которой впервые был заложен функциональный параллелизм. Благодаря наличию 10 независимых функциональных блоков, способных работать параллельно, и 32 независимых модулей памяти удалось достичь быстродействия в 1 MFLOPS (миллион операций с плавающей запятой в секунду). Пятью годами позже Крей создал CDC 7600 с конвейеризированными функциональными блоками и быстродействием 10 MFLOPS. CDC 7600 называют первой конвейерной вычислительной системой (конвейерным процессором). Революционной вехой в истории ВТ стало создание семейства вычислительных машин IBM 360, архитектура и программное обеспечение которых на долгие годы служили эталоном для последующих больших универсальных ВМ (mainframes). В машинах этого семейства нашли воплощение многие новые для того периода

идеи, в частности: предварительная выборка команд, отдельные блоки для операций с фиксированной и плавающей запятой, конвейеризация команд, кэш-память. К третьему поколению ВС относятся также первые параллельные вычислительные системы: SOLOMON корпорации Westinghause и ILLIAC IV — совместная разработка Иллинойского университета и компании Burroughs. Третье поколение ВТ ознаменовалось также появлением первых конвейерно-векторных ВС: TI-ASC (Texas Instruments Advanced Scientific Computer) и STAR-100 фирмы CBC.

Среди вычислительных машин, разработанных в этот период в СССР, прежде всего необходимо отметить «быстродействующую электронно-счетную машину» — БЭСМ-6 (С. А. Лебедев) с производительностью 1 млн операций/с. Продолжением линии М-20 стали М-220 и М-222 с производительностью до 200 000 операций/с. Оригинальная ВМ для инженерных расчетов «Мир-1» была создана под руководством В. М. Глушкова. В качестве входного языка этой ВМ использован язык программирования высокого уровня «Аналитик», во многом напоминающий язык Алгол.

В сфере программного обеспечения необходимо отметить создание в 1970 году Кеном Томпсоном (Kenneth Thompson) из Bell Labs языка B, прямого предшественника популярного языка программирования C, и появление ранней версии операционной системы UNIX.

Четвертое поколение (1972–1984)

Отсчет четвертого поколения обычно ведут с перехода на интегральные микросхемы большой (large-scale integration, LSI) и сверхбольшой (very large-scale integration, VLSI) степени интеграции. К первым относят схемы, содержащие около 1000 транзисторов на кристалле, в то время как число транзисторов на одном кристалле VLSI имеет порядок 100 000. При таких уровнях интеграции стало возможным уместить в одну микросхему не только центральный процессор, но и вычислительную машину (ЦП, основную память и систему ввода/вывода).

Конец 70-х и начало 80-х годов — это время становления и последующего победного шествия микропроцессоров и микроЭВМ, что, однако, не снижает важности изменений, произошедших в архитектуре других типов вычислительных машин и систем.

Одним из наиболее значимых событий в области архитектуры ВМ стала идея вычислительной машины с сокращенным набором команд (RISC, Redused Instruction Set Computer), выдвинутая в 1975 году и впервые реализованная в 1980 году. В упрощенном изложении суть концепции RISC заключается в сведении набора команд ВМ к наиболее употребительным простейшим командам. Это позволяет упростить схемотехнику процессора и добиться резкого сокращения времени выполнения каждой из «простых» команд. Более сложные команды реализуются как подпрограммы, составленные из быстрых «простых» команд.

В ВМ и ВС четвертого поколения практически уходят со сцены ЗУ на магнитных сердечниках, и основная память строится из полупроводниковых запоминающих устройств (ЗУ). До этого использование полупроводниковых ЗУ ограничивалось лишь регистрами и кэш-памятью.

В сфере высокопроизводительных вычислений доминируют векторные вычислительные системы, более известные как суперЭВМ. Разрабатываются новые параллельные архитектуры, однако подобные работы пока еще носят экспериментальный характер. На замену большим ВМ, работающим в режиме разделения времени, приходят индивидуальные микроЭВМ и рабочие станции (этим термином обозначают сетевой компьютер, использующий ресурсы сервера).

В области программного обеспечения выделим появление языков программирования сверхвысокого уровня, таких как FP (functional programming — функциональное программирование) и Пролог (Prolog, programming in logic). Эти языки ориентированы на *аппликативный* и *декларативный стили программирования* соответственно, в отличие от Паскаля, С, Фортрана и т. д. — языков *императивного стиля программирования*. При аппликативном стиле вычисления задаются только как вызовы функций. При декларативном стиле программист дает математическое описание того, что должно быть вычислено, а детали того, каким образом это должно быть сделано, возлагаются на компилятор и операционную систему. Такие языки пока используются недостаточно широко, но выглядят многообещающими для ВС с массовым параллелизмом, состоящими из более чем 1000 процессоров. В компиляторах для ВС четвертого поколения начинают применяться сложные методы оптимизации кода.

Два события в области программного обеспечения связаны с Кеном Томпсоном (Kenneth Thompson) и Деннисом Ритчи (Dennis Ritchie) из Bell Labs. Это создание языка программирования С и его использование при написании операционной системы UNIX для машины DEC PDP-11. Такая форма написания операционной системы позволила быстро распространить UNIX на многие ВМ.

Пятое поколение (1984–1990)

Главным поводом для выделения вычислительных систем второй половины 80-х годов в самостоятельное поколение стало стремительное развитие ВС с сотнями процессоров, ставшее побудительным мотивом для прогресса в области параллельных вычислений. Ранее параллелизм вычислений выражался лишь в виде конвейеризации, векторной обработки и распределения работы между небольшим числом процессоров. Вычислительные системы пятого поколения обеспечивают такое распределение задач по множеству процессоров, при котором каждый из процессоров может выполнять задачу отдельного пользователя.

В рамках пятого поколения в архитектуре вычислительных систем сформировались два принципиально различных подхода: архитектура с совместно используемой памятью и архитектура с распределенной памятью.

Характерным примером первого подхода может служить система Sequent Balance 8000, в которой имеется большая основная память, разделяемая 20 процессорами. Помимо этого, каждый процессор оснащен собственной кэш-памятью. Каждый из процессоров может выполнять задачу своего пользователя, но при этом в составе программного обеспечения имеется библиотека подпрограмм, позволяющая программисту привлекать для решения своей задачи более одного процессора.

Система широко использовалась для исследования параллельных алгоритмов и техники программирования.

Второе направление развития систем пятого поколения — системы с распределенной памятью, где каждый процессор обладает своим модулем памяти, а связь между процессорами обеспечивается сетью взаимосвязей. Примером такой ВС может служить система iPSC-1 фирмы Intel, более известная как «гиперкуб». Максимальный вариант системы включал 128 процессоров. Применение распределенной памяти позволило устранить ограничения в пропускной способности тракта «процессор-память», но потенциальным «узким местом» здесь становится сеть взаимосвязей.

Наконец, третье направление в архитектуре вычислительных систем пятого поколения — это ВС, в которых несколько тысяч достаточно простых процессоров работают под управлением единого устройства управления и одновременно производят одну и ту же операцию, но каждый над своими данными. К этому классу можно отнести Connection Machine фирмы Thinking Machines Inc. и MP-1 фирмы MasPar Inc.

В научных вычислениях по-прежнему ведущую роль играют векторные суперЭВМ. Многие производители предлагают более эффективные варианты с несколькими векторными процессорами, но число таких процессоров обычно невелико (от 2 до 8).

RISC-архитектура выходит из стадии экспериментов и становится базовой архитектурой для рабочих станций (workstations).

Знаковой приметой рассматриваемого периода стало стремительное развитие технологий глобальных и локальных компьютерных сетей. Это стимулировало изменения в технологии работы индивидуальных пользователей. В противовес мощным универсальным ВС, работающим в режиме разделения времени, пользователи все более отдают предпочтение подключенным к сети индивидуальным рабочим станциям. Такой подход позволяет для решения небольших задач задействовать индивидуальную машину, а при необходимости в большой вычислительной мощности обратиться к ресурсам подсоединенных к той же сети мощных файл-серверов или суперЭВМ.

Шестое поколение (1990-)

На ранних стадиях эволюции вычислительных средств смена поколений ассоциировалась с революционными технологическими прорывами. Каждое из первых четырех поколений имело четко выраженные отличительные признаки и вполне определенные хронологические рамки. Последующее деление на поколения уже не столь очевидно и может быть понятно лишь при ретроспективном взгляде на развитие вычислительной техники. Пятое и шестое поколения в эволюции ВТ — это отражение нового качества, возникшего в результате последовательного накопления частных достижений, главным образом в архитектуре вычислительных систем и, в несколько меньшей мере, в сфере технологий.

Поводом для начала отсчета нового поколения стали значительные успехи в области параллельных вычислений, связанные с широким распространением вычи-

слительных систем с массовым параллелизмом. Особенности организации таких систем, обозначаемых аббревиатурой MPP (massively parallel processing), будут рассмотрены в последующих главах. Здесь же упрощенно определим их как совокупность большого количества (до нескольких тысяч) взаимодействующих, но достаточно автономных вычислительных машин. По вычислительной мощности такие системы уже успешно конкурируют с суперЭВМ, которые, как ранее отмечалось, по своей сути являются векторными ВС. Появление вычислительных систем с массовым параллелизмом дало основание говорить о производительности, измеряемой в TFLOPS (1 TFLOPS соответствует 1012 операциям с плавающей запятой в секунду).

Вторая характерная черта шестого поколения — резко возросший уровень рабочих станций. В процессорах новых рабочих станций успешно совмещаются RISC-архитектура, конвейеризация и параллельная обработка. Некоторые рабочие станции по производительности сопоставимы с суперЭВМ четвертого поколения. Впечатляющие характеристики рабочих станций породили интерес к гетерогенным (неоднородным) вычислениям, когда программа, запущенная на одной рабочей станции, может найти в локальной сети не занятые в данный момент другие станции, после чего вычисления распараллеливаются и на эти простаивающие станции.

Наконец, третьей приметой шестого поколения в эволюции ВТ стал взрывной рост глобальных сетей. Этот момент, однако, выходит за рамки данного учебника, поэтому далее комментироваться не будет.

Завершая обсуждение эволюции ВТ, отметим, что верхняя граница шестого поколения хронологически пока не определена, и дальнейшее развитие вычислительной техники может внести в его характеристику новые коррективы. Не исключено также, что последующие события дадут повод говорить и об очередном поколении.

Концепция машины с хранимой в памяти программой

Исходя из целей данного раздела, введем еще одно определение термина «вычислительная машина» как совокупности технических средств, служащих для автоматизированной обработки дискретных данных по заданному алгоритму.

Алгоритм — одно из фундаментальных понятий математики и вычислительной техники. Международная организация стандартов (ISO) формулирует понятие *алгоритм* как «конечный упорядоченный набор четко определенных правил для решения проблемы» (ISO 2382/1-93). Помимо этой стандартизированной формулировки существуют и другие определения. Приведем наиболее распространенные из них. Итак, алгоритм — это:

- способ преобразования информации, задаваемый с помощью конечной системы правил;
- совокупность правил, определяющих эффективную процедуру решения любой задачи из некоторого заданного класса задач;

- точно определенное правило действий, для которого задано указание, как и в какой последовательности это правило необходимо применять к исходным данным задачи, чтобы получить ее решение;
- точное предписание, определяющее содержание и порядок действий, которые необходимо выполнить над исходными и промежуточными данными для получения конечного результата при решении всех задач определенного типа.

Основными свойствами алгоритма являются: дискретность, определенность, массовость и результативность.

Дискретность выражается в том, что алгоритм описывает действия над дискретной информацией (например, числовой или символьной), причем сами эти действия также дискретны.

Свойство *определенности* означает, что в алгоритме указано все, что должно быть сделано, причем ни одно из действий не должно трактоваться двояко.

Массовость алгоритма подразумевает его применимость к множеству значений исходных данных, а не только к каким-то уникальным значениям.

Наконец, *результативность* алгоритма состоит в возможности получения результата за конечное число шагов.

Рассмотренные свойства алгоритмов предопределяют возможность их реализации на ВМ, при этом процесс, порождаемый алгоритмом, называют *вычислительным процессом*.

В основе архитектуры современных ВМ лежит представление алгоритма решения задачи в виде программы. Согласно стандарту ISO 2382/1-93, программа для ВМ состоит из команд, необходимых для выполнения функций, задач. Причем эти команды соответствуют правилам конкретного языка программирования.

ВМ, где определенным образом закодированные команды программы хранятся в памяти, известна под названием *вычислительной машины с хранимой в памяти программой*. Идея принадлежит создателям вычислителя ENIAC Эккерту, Мочли и фон Нейману. Еще до завершения работ над ENIAC они приступили к новому проекту — EDVAC, главной особенностью которого стала концепция хранимой в памяти программы, на долгие годы определившая базовые принципы построения последующих поколений вычислительных машин. Относительно авторства существует несколько версий, но поскольку в законченном виде идея впервые была изложена в 1945 году в статье фон Неймана [161], именно его фамилия фигурирует в обозначении архитектуры подобных машин, составляющих подавляющую часть современного парка ВМ и ВС.

Сущность фон-неймановской концепции вычислительной машины можно свести к четырем принципам:

- двоичного кодирования;
- программного управления;
- однородности памяти;
- адресуемости памяти.

Принцип двоичного кодирования

Согласно этому принципу, вся информация, как данные, так и команды, кодируется двоичными цифрами 0 и 1. Каждый тип информации представляется в двоичном виде и имеет свой *формат*. Последовательность битов в формате, имеющая определенный смысл, называется *полем*. В формате числа обычно выделяют *поле знака* и *поле значащих разрядов*. В формате команды можно выделить два поля (рис. 1.2): *поле кода операции* (КОП) и *поле адресов* (адресную часть — АЧ).

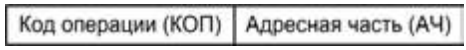


Рис. 1.2. Структура команды

Код операции представляет собой указание, какая операция должна быть выполнена, и задается с помощью r-разрядной двоичной комбинации.

Вид адресной части и число составляющих ее адресов зависят от типа команды: в командах преобразования данных АЧ содержит адреса объектов обработки (*операндов*) и результата; в командах изменения порядка вычислений — адрес следующей команды программы; в командах ввода/вывода — номер устройства ввода/вывода. Адресная часть также представляется двоичным кодом, длину которого обозначим через p. Таким образом, команда в вычислительной машине имеет вид $(r + p)$-разрядной двоичной комбинации.

Принцип программного управления

Все вычисления, предусмотренные алгоритмом решения задачи, должны быть представлены в виде *программы*, состоящей из последовательности управляющих слов — *команд*. Каждая команда предписывает некоторую операцию из набора операций, реализуемых вычислительной машиной. Команды программы хранятся в последовательности смежных ячеек памяти вычислительной машины и выполняются *в естественном порядке*, то есть в порядке их расположения в программе. При необходимости, с помощью специальных команд, естественный порядок выполнения может быть изменен. Решение об изменении порядка выполнения команд программы принимается либо на основании анализа результатов предшествующих вычислений, либо безусловно.

Принцип однородности памяти

Команды и данные хранятся в одной и той же памяти и внешне в памяти неразличимы. Распознать их можно только по способу использования. Это позволяет производить над командами те же операции, что и над числами, и, соответственно, открывает ряд возможностей. Так, циклически изменяя адресную часть команды, можно обеспечить обращение к последовательности смежных элементов массива данных. Такой прием носит название *модификации команд* и с позиций современного программирования не приветствуется. Более полезным является другое следствие принципа однородности, когда команды одной программы могут быть получены как результат исполнения другой программы. Эта возможность лежит

в основе *трансляции* — перевода текста программы с языка высокого уровня на машинный язык конкретной ВМ.

Концепция вычислительной машины, изложенная в статье фон Неймана, предполагает единую память для хранения команд и данных. Такой подход был принят в вычислительных машинах, создававшихся в Принстонском университете, из-за чего и получил название *принстонской архитектуры*. Практически одновременно в Гарвардском университете предложили иную модель, в которой ВМ имела отдельную память команд и отдельную память данных. Этот вид архитектуры называют *гарвардской архитектурой*. Долгие годы преобладающей была и остается принстонская архитектура, хотя она порождает проблемы пропускной способности тракта «процессор-память». В последнее время, в связи с широким использованием кэш-памяти, разработчики ВМ все чаще обращаются к гарвардской архитектуре.

Принцип адресуемости памяти

Структурно основная память состоит из пронумерованных ячеек, причем процессору в произвольный момент доступна любая ячейка. Двоичные коды команд и данных разделяются на единицы информации, называемые *словами*, и хранятся в ячейках памяти, а для доступа к ним используются номера соответствующих ячеек — *адреса*.

Фон-неймановская архитектура

В статье фон Неймана определены основные устройства ВМ, с помощью которых должны быть реализованы вышеперечисленные принципы. Большинство современных ВМ по своей структуре отвечают принципу программного управления. Типичная фон-неймановская ВМ (рис. 1.3) содержит: память, устройство управления, арифметико-логическое устройство и устройство ввода/вывода[1].

В любой ВМ имеются средства для ввода программ и данных к ним. Информация поступает из подсоединенных к ВМ *периферийных устройств* (ПУ) ввода. Результаты вычислений выводятся на периферийные устройства вывода. Связь и взаимодействие ВМ и ПУ обеспечивают *порты ввода* и *порты вывода*. Термином *порт* обозначают аппаратуру сопряжения периферийного устройства с ВМ и управления им. Совокупность портов ввода и вывода называют *устройством ввода/вывода* (УВВ) или *модулем ввода/вывода* вычислительной машины (МВВ).

Память компьютера имеет сложную многоуровневую структуру, реализованную в виде взаимодействующих запоминающих устройств (ЗУ), которые могут использовать различные физические принципы для хранения данных.

Введенная информация сначала запоминается в основной памяти, а затем переносится во вторичную память, для длительного хранения. Чтобы программа могла выполняться, команды и данные должны располагаться в *основной памяти* (ОП),

[1] Показанные на рисунке кэш-память и вторичная память в состав типичной фон-неймановской ВМ не входят (являются ее расширением).

организованной таким образом, что каждое двоичное слово хранится в отдельной ячейке, идентифицируемой адресом, причем соседние ячейки памяти имеют следующие по порядку адреса.

Доступ к любым ячейкам основной памяти может производиться в произвольном порядке. Такой вид памяти известен как *память с произвольным доступом*. ОП современных ВМ в основном состоит из полупроводниковых *оперативных запоминающих устройств* (ОЗУ), обеспечивающих как считывание, так и запись информации. Для таких ЗУ характерна энергозависимость — хранимая информация теряется при отключении электропитания. Если необходимо, чтобы часть основной памяти была энергонезависимой, в состав ОП включают *постоянные запоминающие устройства* (ПЗУ), также обеспечивающие произвольный доступ. Хранящаяся в ПЗУ информация может только считываться.

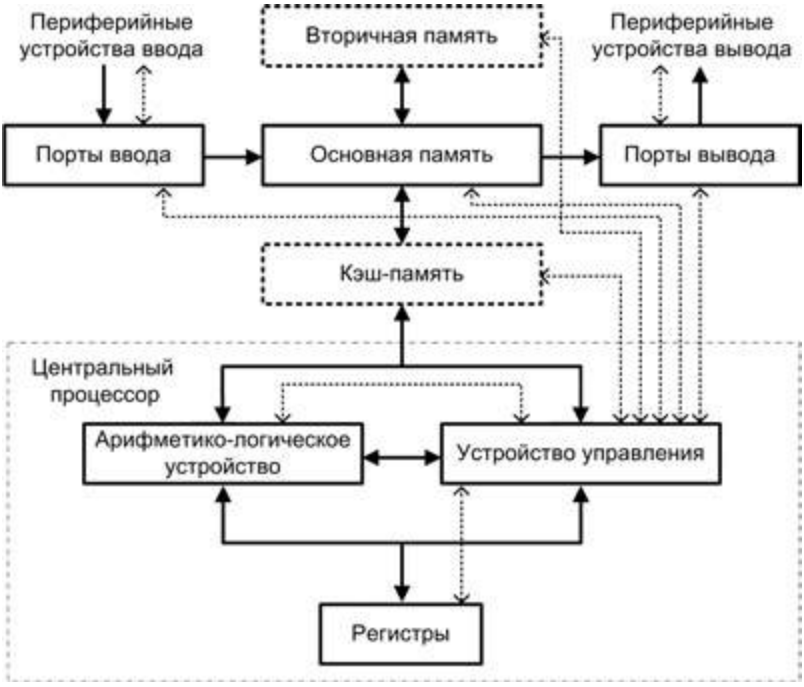


Рис. 1.3. Структура фон-неймановской вычислительной машины

Размер ячейки основной памяти обычно принимается равным 8 двоичным разрядам — *байту*. Для хранения больших чисел используются 2, 4 или 8 байтов, размещаемых в последовательности ячеек со смежными адресами. В этом случае за адрес числа часто принимается адрес его младшего байта. Так, при хранении 32-разрядного числа в ячейках с адресами 200, 201, 202 и 203 адресом числа будет 200. Такой прием называют адресацией по младшему байту или методом «остроконечников» (*little endian addressing*). Возможен и противоположный подход — по меньшему из адресов располагается старший байт. Этот способ известен как адресация

по старшему байту или метод «тупоконечников» (*big endian addressing*)[1]. Адресация по младшему байту характерна для микропроцессоров фирмы Intel, а по старшему байту — для микропроцессоров фирмы Motorola и больших ВМ фирмы IBM. В принципе, выбор порядка записи байтов существенен лишь при пересылке данных между ВМ с различными формами их адресации или при манипуляциях с отдельными байтами числа. В большинстве ВМ предусмотрены специальные команды для перехода от одного способа к другому.

Для долговременного хранения больших программ и массивов данных в ВМ обычно имеется дополнительная память, известная как *вторичная*. Вторичная память энергонезависима и чаще всего реализуется на базе магнитных дисков. Информация в ней хранится в виде специальных программно поддерживаемых объектов — *файлов* (согласно стандарту ISO, файл — это именуемый набор записей, обрабатываемых как единый блок).

Неотъемлемой частью современных ВМ стала *кэш-память* — память небольшой емкости, но высокого быстродействия. В нее из основной памяти копируются наиболее часто используемые команды и данные. При обращении со стороны процессора информация берется не из основной памяти, а из соответствующей копии, находящейся в более быстродействующей кэш-памяти.

Наконец, четвертый вид памяти, показанный на схеме, — это регистры центрального процессора. Каждый регистр можно рассматривать как одну ячейку памяти[2], обращение к которой занимает значительно меньше времени даже по сравнению с остаточно быстродействующей кэш-памятью. Число регистров в ВМ обычно невелико, и они образуют небольшую память, иногда называемую сверхоперативной памятью или регистровым файлом. В регистры обычно помещают часто используемые константы или промежуточные результаты вычислений, что позволяет сократить число обращений к более медленным видам памяти.

Отметим, что обязательным элементом фон-неймановской архитектуры можно считать лишь основную память. Включение в состав ВМ остальных видов запоминающих устройств (ЗУ) обусловлено в основном технологическими проблемами, препятствующими созданию достаточно быстродействующих, дешевых и энергонезависимых ЗУ большой емкости.

Устройство управления (УУ) — важнейшая часть вычислительной машины, организующая автоматическое выполнение программ (путем реализации функций управления) и обеспечивающая функционирование ВМ как единой системы. Для пояснения функций УУ вычислительную машину следует рассматривать как совокупность элементов, между которыми происходит пересылка информации, в ходе которой эта информация может подвергаться определенным видам обработки.

---

[1] Термины «остроконечники» и «тупоконечники» заимствованы из книги «Путешествия Гулливера» Дж. Свифта, где упоминается религиозная война между двумя группами, представители одной из которых разбивали яйцо с острого (little) конца, а их антагонисты — с тупого (big).

[2] В отличие от ячейки основной памяти разрядность регистра обычно больше одного байта и совпадает с разрядностью ВМ.

Пересылка информации между любыми элементами ВМ инициируется своим *сигналом управления* (СУ), то есть управление вычислительным процессом сводится к выдаче нужного набора СУ в нужной временной последовательности. Цепи СУ показаны на рис. 1.3 пунктирными линиями. Основной функцией УУ является формирование управляющих сигналов, отвечающих за извлечение команд из памяти в порядке, определяемом программой, и последующее исполнение этих команд. Кроме того, УУ формирует СУ для синхронизации и координации внутренних и внешних устройств ВМ.

Еще одной неотъемлемой частью ВМ является *арифметико-логическое устройство* (АЛУ). АЛУ обеспечивает арифметическую и логическую обработку двух входных переменных (операндов), в итоге которой формируется выходная переменная (результат). Функции АЛУ обычно сводятся к простым арифметическим и логическим операциям, а также операциям сдвига. Помимо результата операции, АЛУ формирует ряд *признаков результата* (флагов), характеризующих полученный результат и события, произошедшие в ходе его получения (равенство нулю, знак, четность, перенос, переполнение и т. д.). Флаги могут анализироваться в УУ с целью принятия решения о дальнейшем порядке следования команд программы.

УУ и АЛУ тесно взаимосвязаны и их обычно рассматривают как единое устройство, известное как *центральный процессор* (ЦП) или просто *процессор*. Помимо УУ и АЛУ в процессор входит также набор *регистров общего назначения* (РОН), служащих для промежуточного хранения информации в процессе ее обработки. В ГОСТ 15971-90 назначение процессора определено как «интерпретация программ».

Типы структур вычислительных машин и систем

Достоинства и недостатки архитектуры вычислительных машин и систем изначально зависят от способа соединения компонентов. При самом общем подходе можно говорить о двух основных типах структур вычислительных машин и двух типах структур вычислительных систем.

Структуры вычислительных машин

В настоящее время примерно одинаковое распространение получили два способа построения вычислительных машин: *с непосредственными связями* и *на основе шины*.

Типичным представителем первого способа может служить классическая фоннеймановская ВМ (см. рис. 1.3). В ней между взаимодействующими устройствами (процессор, память, устройство ввода/вывода) имеются непосредственные связи. Особенности связей (число линий в шинах, пропускная способность и т. п.) определяются видом информации, характером и интенсивностью обмена. Достоинством архитектуры с непосредственными связями можно считать возможность развязки «узких мест» путем улучшения структуры и характеристик только определенных связей, что экономически может быть наиболее выгодным решением. У фоннеймановских ВМ таким «узким местом» является канал пересылки данных между ЦП и памятью и «развязать» его достаточно непросто [50]. Кроме того, ВМ с непосредственными связями плохо поддаются реконфигурации.

В варианте с общей шиной все устройства вычислительной машины подключены к магистральной шине, служащей единственным трактом для потоков команд, данных и управления (рис. 1.4). Наличие общей шины существенно упрощает реализацию ВМ, позволяет легко менять состав и конфигурацию машины. Благодаря этим свойствам шинная архитектура получила широкое распространение в мини- и микроЭВМ. Вместе с тем, именно с шиной связан и основной недостаток архитектуры: в каждый момент передавать информацию по шине может только одно устройство. Основную нагрузку на шину создают обмены между процессором и памятью, связанные с извлечением из памяти команд и данных и записью в память результатов вычислений. На операции ввода/вывода остается лишь часть пропускной способности шины. Практика показывает, что даже при достаточно быстрой шине для 90 % приложений этих остаточных ресурсов обычно не хватает, особенно в случае ввода или вывода больших массивов данных.

Рис. 1.4. Структура вычислительной машины на базе общей шины

В целом, следует признать, что при сохранении фон-неймановской концепции последовательного выполнения команд программы (одна команда в единицу времени) шинная архитектура в чистом ее виде оказывается недостаточно эффективной. Более распространена *архитектура с иерархией шин*, где помимо магистральной шины имеется еще несколько дополнительных шин. Они могут обеспечивать непосредственную связь между устройствами с наиболее интенсивным обменом, например процессором и кэш-памятью. Другой вариант использования дополнительных шин — объединение однотипных устройств ввода/вывода с последующим выходом с дополнительной шины на магистральную. Все эти меры позволяют снизить нагрузку на общую шину и более эффективно расходовать ее пропускную способность.

Структуры вычислительных систем

Понятие «вычислительная система» предполагает наличие множества процессоров или законченных вычислительных машин, при объединении которых используется один из двух подходов.

В вычислительных *системах с общей памятью* (рис. 1.5) имеется общая основная память, совместно используемая всеми процессорами системы. Связь процессоров с памятью обеспечивается с помощью коммуникационной сети, чаще всего вырождающейся в общую шину. Таким образом, структура ВС с общей памятью аналогична рассмотренной выше архитектуре с общей шиной, в силу чего ей свойственны те же недостатки. Применительно к вычислительным системам данная схема имеет дополнительное достоинство: обмен информацией между процессорами не связан с дополнительными операциями и обеспечивается за счет доступа к общим областям памяти.

Рис. 1.5. Структура вычислительной системы с общей памятью

Альтернативный вариант организации — *распределенная* система, где общая память вообще отсутствует, а каждый процессор обладает собственной локальной памятью (рис. 1.6). Часто такие системы объединяют отдельные ВМ. Обмен информацией между составляющими системы обеспечивается с помощью коммуникационной сети посредством обмена сообщениями.

Рис. 1.6. Структура распределенной вычислительной системы

Подобное построение ВС снимает ограничения, свойственные для общей шины, но приводит к дополнительным издержкам на пересылку сообщений между процессорами или машинами.

Основные показатели вычислительных машин

Использование конкретной вычислительной машины имеет смысл, если ее показатели соответствуют показателям, определяемым требованиями к реализации заданных алгоритмов. В качестве основных показателей ВМ обычно рассматривают: емкость памяти, быстродействие и производительность, стоимость и надежность

[20]. В данном учебнике остановимся только на показателях быстродействия и производительности, обычно представляющих основной интерес для пользователей.

Быстродействие

Целесообразно рассматривать два вида быстродействия: номинальное и среднее.

Номинальное быстродействие характеризует возможности ВМ при выполнении стандартной операции. В качестве стандартной обычно выбирают короткую операцию сложения. Если обозначить через $\tau_{сл}$ время сложения, то номинальное быстродействие определится из выражения:

$$V_{ном} = \frac{1}{\tau_{сл}} \left[\frac{оп}{с}\right].$$

Среднее быстродействие характеризует скорость вычислений при выполнении эталонного алгоритма или некоторого класса алгоритмов. Величина среднего быстродействия зависит как от параметров ВМ, так и от параметров алгоритма и определяется соотношением:

$$V_{ср} = \frac{N}{T_э},$$

где $T_э$ — время выполнения эталонного алгоритма; N — количество операций, содержащихся в эталонном алгоритме.

Обозначим через n_i число операций i-го типа; l — количество типов операций в алгоритме ($i = 1, 2, ..., l$); τ_i — время выполнения операции i-го типа.

Время выполнения эталонного алгоритма рассчитывается по формуле:

$$T_э = \sum_{i=1}^{l} \tau_i n_i. \qquad (1.1)$$

Подставив (1.1) в выражение для $V_{ср}$, получим

$$V_{ср} = \frac{N}{\sum_{i=1}^{l} \tau_i n_i}. \qquad (1.2)$$

Разделим числитель и знаменатель в (1.2) на N:

$$V_{ср} = \frac{1}{\sum_{i=1}^{l} \frac{n_i}{N} \tau_i}. \qquad (1.3)$$

Обозначив частоту появления операции i-го типа в (1.3) через $q_i = \frac{n_i}{N}$, запишем окончательную формулу для расчета среднего быстродействия:

$$V_{\text{ср}} = \frac{1}{\sum_{i=1}^{l} q_i \tau_i} \left[\frac{\text{оп}}{\text{с}}\right]. \quad (1.4)$$

В выражении (1.4) вектор $\{\tau_1, \tau_2, ..., \tau_l\}$ характеризует систему команд ВМ, а вектор $\{q_1, q_2, ..., q_l\}$, называемый частотным вектором операций, характеризует алгоритм.

Очевидно, что для эффективной реализации алгоритма необходимо стремиться к увеличению $V_{\text{ср}}$. Если $V_{\text{ном}}$ главным образом отталкивается от быстродействия элементной базы, то $V_{\text{ср}}$ очень сильно зависит от оптимальности выбора команд ВМ.

Формула (1.4) позволяет определить среднее быстродействие машины при реализации одного алгоритма. Рассмотрим более общий случай, когда полный алгоритм состоит из нескольких частных, периодически повторяемых алгоритмов. Среднее быстродействие при решении полной задачи рассчитывается по формуле:

$$V_{\text{ср}}^{n} = \frac{1}{\sum_{j=1}^{m}\sum_{i=1}^{l} \beta_j q_{ji} \tau_i} \left[\frac{\text{оп}}{\text{с}}\right], \quad (1.5)$$

где m — количество частных алгоритмов; β_j — частота появления операций j-го частного алгоритма в полном алгоритме; q_{ij} — частота операций i-го типа в j-м частном алгоритме.

Обозначим через N_j и T_j — количество операций и период повторения j-го частного алгоритма; $T_{\max} = \max_j(T_1, ..., T_j, ..., T_m)$ — период повторения полного алгоритма; $\alpha_j = \frac{T_{\max}}{T_j}$ — цикличность включения j-го частного алгоритма в полном алгоритме.

Тогда за время $T_{\max}$ в ВМ будет выполнено $N_{\max} = \sum_{j=1}^{m} \alpha_j N_j$ операций, а частоту появления операций j-го частного алгоритма в полном алгоритме можно определить из выражения:

$$\beta_j = \frac{\alpha_j N_j}{N_{\max}}. \quad (1.6)$$

Для расчета по формулам (1.5, 1.6) необходимо знать параметры ВМ, представленные вектором $\{\tau_1, \tau_2, ..., \tau_l\}$, параметры каждого j-го частного алгоритма — вектор $\{q_{j1}, q_{j2}, ..., q_{jl}\}$ и параметры полного алгоритма — вектор $\{\beta_1, \beta_2, ..., \beta_m\}$.

Производительность ВМ оценивается количеством эталонных алгоритмов, выполняемых в единицу времени:

$$П = \frac{1}{T_э} \left[\frac{\text{задач}}{\text{с}}\right].$$

Производительность при выполнении полного алгоритма оценивается по формуле:

$$П_{\text{п}} = \frac{1}{\sum_{j=1}^{m}\sum_{i=1}^{l}\alpha_j N_j q_{ij}\tau_i} \left[\frac{\text{задач}}{\text{с}}\right]. \quad (1.7)$$

Производительность является более универсальным показателем, чем среднее быстродействие, поскольку в явном виде зависит от порядка прохождения задач через ВМ.

Критерии эффективности вычислительных машин

Вычислительную машину можно определить множеством показателей, характеризующих отдельные ее свойства. Возникает задача введения меры для оценки степени приспособленности ВМ к выполнению возложенных на нее функций — меры эффективности.

Эффективность определяет степень соответствия ВМ своему назначению. Она измеряется либо количеством затрат, необходимых для получения определенного результата, либо результатом, полученным при определенных затратах. Произвести сравнительный анализ эффективности нескольких ВМ, принять решение на использование конкретной машины позволяет критерий эффективности.

Критерий эффективности — это правило, служащее для сравнительной оценки качества вариантов ВМ. Критерий эффективности можно назвать правилом предпочтения сравниваемых вариантов.

Строятся критерии эффективности на основе частных показателей эффективности (показателей качества). Способ связи между частными показателями определяет вид критерия эффективности.

Способы построения критериев эффективности

Возможны следующие способы построения критериев из частных показателей.

Выделение главного показателя. Из совокупности частных показателей A_1, A_2, ..., A_n выделяется один, например A_1, который принимается за главный. На остальные показатели накладываются ограничения:

$$A_i \leqslant A_{i\text{доп}} \ (i = 2, 3, ..., n),$$

где $A_{i\text{доп}}$ — допустимое значение i-го показателя. Например, если в качестве A_1 выбирается производительность, а на показатели надежности P и стоимости S накладываются ограничения, то критерий эффективности ВМ принимает вид:

$$П_{\text{упр}} \to \max, P \leqslant P_{\text{доп}}, S \leqslant S_{\text{доп}}.$$

Способ последовательных уступок. Все частные показатели нумеруются в порядке их важности: наиболее существенным считается показатель A_1, а наименее важным — A_n. Находится минимальное значение показателя A_1 — $\min A_1$ (если нужно найти максимум, то достаточно изменить знак показателя). Затем делается «уступка» первому показателю ΔA_1, и получается ограничение $\min A_1 + \Delta A_1$.

На втором шаге отыскивается min A_2 при ограничении $A_1 \le \min A_1 + \Delta A_1$. После этого выбирается «уступка» для A_2: min $A_2 + \Delta A_2$. На третьем шаге отыскивается *min* A_3 при ограничениях $A_1 \le \min A_1 + \Delta A_1$; $A_2 \le \min A_2 + \Delta A_2$ и т. д. На последнем шаге ищут min A_n при ограничениях

$$A_1 \le \min A_1 + \Delta A_1;$$
$$A_2 \le \min A_2 + \Delta A_2;$$
$$\ldots$$
$$A_{n-1} \le \min A_{n-1} + \Delta A_{n-1}.$$

Полученный на этом шаге вариант вычислительной машины и значения ее показателей $A_1, A_2, ..., A_n$ считаются окончательными. Недостатком данного способа (критерия) является неоднозначность выбора ΔA_i.

Отношение частных показателей. В этом случае критерий эффективности получают в виде:

$$K_1 = \frac{A_1, A_2, ..., A_n}{B_1, B_2, ..., B_m} \to \max, \tag{1.8}$$

или в виде

$$K_2 = \frac{B_1, B_2, ..., B_m}{A_1, A_2, ..., A_n} \to \min, \tag{1.9}$$

где A_i ($i = 1, 2, ..., n$) — частные показатели, для которых желательно увеличение численных значений, а B_i ($i = 1, 2, ..., m$) — частные показатели, численные значения которых нужно уменьшить. В частном случае критерий может быть представлен в виде:

$$K_3 = \frac{B_1}{A_1} \to min. \tag{1.10}$$

Наиболее популярной формой выражения (1.10) является критерий цены эффективного быстродействия:

$$K_4 = \frac{S}{V_{\text{ср}}} \to \min, \tag{1.11}$$

где S — стоимость, $V_{\text{ср}}$ — среднее быстродействие ВМ. Формула критерия K_4 характеризует аппаратные затраты, приходящиеся на единицу быстродействия.

Аддитивная форма. Критерий эффективности имеет вид:

$$K_5 = \sum_{i=1}^{n} \alpha_i A_i \to \max, \tag{1.12}$$

где $\alpha_1, \alpha_2, ..., \alpha_n$ — положительные и отрицательные весовые коэффициенты частных показателей. Положительные коэффициенты ставятся при тех показателях, которые желательно максимизировать, а отрицательные — при тех, которые желательно минимизировать.

Весовые коэффициенты могут быть определены методом экспертных оценок. Обычно они удовлетворяют условиям:

$$0 \le \alpha_i < 1, \ \sum_{i=1}^{n} \alpha_i = 1. \tag{1.13}$$

Основной недостаток критерия заключается в возможности взаимной компенсации частных показателей.

Мультипликативная форма. Критерий эффективности имеет вид:

$$K_6 = \prod_{i=1}^{n} A_i^{\alpha_i} \to \max, \tag{1.14}$$

где, в частном случае, коэффициенты α_i полагают равными единице.

От мультипликативной формы можно перейти к аддитивной, используя выражение:

$$\lg K_6 = \sum_{i=1}^{n} \alpha_i \lg A_i. \tag{1.15}$$

Критерий K_6 имеет тот же недостаток, что и критерий K_5.

Максиминная форма. Критерий эффективности описывается выражением:

$$K_7 = \min_i A_i \to \max. \tag{1.16}$$

Здесь реализована идея равномерного повышения уровня всех показателей за счет максимального «подтягивания» наихудшего из показателей (имеющего минимальное значение).

У максиминного критерия нет того недостатка, который присущ мультипликативному и аддитивному критериям.

Нормализация частных показателей

Частные показатели качества обычно имеют различную физическую природу и различные масштабы измерений, из-за чего их простое сравнение становится практически невозможным. Поэтому появляется задача приведения частных показателей к единому масштабу измерений, то есть их нормализация.

Рассмотрим отдельные способы нормализации.

Использование отклонения частного показателя от максимального.

$$\Delta A_i = A_{\max_i} - A_i. \tag{1.17}$$

В данном случае переходят к отклонениям показателей, однако способ не устраняет различия масштабов отклонений.

Использование безразмерной величины $\overline{A_i}$.

$$\overline{A_i} = \frac{A_{\max_i} - A_i}{A_{\max_i}}, \tag{1.18}$$

$$\overline{A_i} = \frac{A_i}{A_{\max_i}}. \tag{1.19}$$

Формула (1.18) применяется тогда, когда уменьшение A_i приводит к увеличению (улучшению) значения аддитивной формулы критерия. Выражение (1.19) используется, когда к увеличению значения аддитивной формулы критерия приводит увеличение A_i.

Учет приоритета частных показателей

Необходимость в учете приоритетов возникает в случае, когда частные показатели имеют различную степень важности.

Приоритет частных показателей задается с помощью ряда приоритета I, вектора приоритета (b_1, ..., b_q, ..., b_n) и вектора весовых коэффициентов (α_1, α_2, ..., α_n).

Ряд приоритета представляет собой упорядоченное множество индексов частных показателей I = (1, 2, ..., n). Он отражает чисто качественные отношения доминирования показателей, а именно отношения следующего типа: показатель A_1 важнее показателя A_2, а показатель A_2 важнее показателя A_3 и т. д.

Элемент b_q вектора приоритета показывает, во сколько раз показатель A_q важнее показателя A_{q+1} (здесь A_q — показатель, которому отведен номер q в ряду приоритета). Если A_q и A_{q+1} имеют одинаковый ранг, то $b_q = 1$. Для удобства принимают $b_n = 1$.

Компоненты векторов приоритета и весовых коэффициентов связаны между собой следующим отношением:

$$b_q = \frac{\alpha_q}{\alpha_{q+1}}.$$

Зависимость, позволяющая по известным значениям b_i определить величину α_q, имеет вид:

$$\alpha_q = \frac{\prod_{i=q}^{n} b_i}{\sum_{q=1}^{n}\prod_{i=q}^{n} b_i}. \tag{1.20}$$

Знание весовых коэффициентов позволяет учесть приоритет частных показателей.

Перспективы совершенствования архитектуры ВМ и ВС

Совершенствование архитектуры вычислительных машин и систем началось с момента появления первых ВМ и не прекращается по сей день. Каждое изменение в архитектуре направлено на абсолютное повышение производительности или, по крайней мере, на более эффективное решение задач определенного класса.

Эволюцию архитектур определяют самые различные факторы, главные из которых показаны на рис. 1.7. Не умаляя роли ни одного из них, следует признать, что наиболее очевидные успехи в области средств вычислительной техники все же связаны с технологическими достижениями. Характер и степень влияния прочих факторов подробно описаны в [93] и в данном учебнике не рассматриваются.

Рис. 1.7. Факторы, определяющие развитие архитектуры вычислительных систем

С каждым новым технологическим успехом многие из архитектурных идей переходят на уровень практической реализации. Очевидно, что процесс этот будет продолжаться и в дальнейшем, однако возникает вопрос: «Насколько быстро?» Косвенный ответ можно получить, проанализировав тенденции совершенствования технологий, главным образом полупроводниковых.

Тенденции развития больших интегральных схем

На современном уровне вычислительной техники подавляющее большинство устройств ВМ и ВС реализуется на базе полупроводниковых технологий в виде *сверхбольших интегральных микросхем* (СБИС). Каждое нововведение в области архитектуры ВМ и ВС, как правило, связано с необходимостью усложнения схемы процессора или его составляющих и требует размещения на кристалле СБИС все большего числа логических или запоминающих элементов. Задача может быть решена двумя путями: увеличением размеров кристалла и уменьшением площади, занимаемой на кристалле элементарным транзистором, с одновременным повышением плотности упаковки таких транзисторов на кристалле.

Пока основные успехи в плане увеличения емкости СБИС связаны с уменьшением размеров элементарных транзисторов и плотности их размещения на кристалле. Здесь тенденции эволюции СБИС хорошо описываются эмпирическим законом Мура[1] [120]. В 1965 году Мур заметил, что число транзисторов, которое удается разместить на кристалле микросхемы, удваивается каждые 12 месяцев. Он предсказал, что эта тенденция сохранится в 70-е годы, а начиная с 80-х темп роста начнет спадать. В 1995 году Мур уточнил свое предсказание, сделав прогноз, что удвоение числа транзисторов далее будет происходить каждые 24 месяца.

[1] Гордон Мур (Gordon E. Moore) один из основателей фирмы Intel.

Современные технологии производства сверхбольших интегральных микросхем позволяют разместить на одном кристалле логические схемы всех компонентов процессора. В настоящее время процессоры всех вычислительных машин реализуются в виде одной или нескольких СБИС. Более того, во многих многопроцессорных ВС используются СБИС, где на одном кристалле располагаются сразу несколько процессоров (обычно не очень сложных).

Каждый успех создателей процессорных СБИС немедленно положительно отражается на характеристиках ВМ и ВС. Совершенствование процессорных СБИС ведется по двум направлениям: увеличение количества логических элементов, которое может быть размещено на кристалле, повышение быстродействия этих логических элементов. Увеличение быстродействия ведет к наращиванию производительности процессоров даже без изменения их архитектуры, а в совокупности с повышением плотности упаковки логических элементов открывает возможности для реализации ранее недоступных архитектурных решений.

В целом, для процессорных СБИС можно сделать следующие выводы [133]:

- плотность упаковки логических схем процессорных СБИС каждые два года будет возрастать вдвое;
- удвоение внутренней тактовой частоты процессорных СБИС происходит в среднем каждые два года.

По мере повышения возможностей вычислительных средств растут и «аппетиты» программных приложений относительно емкости основной памяти. Эту ситуацию отражает так называемый закон Паркинсона: «Программное обеспечение увеличивается в размерах до тех пор, пока не заполнит всю доступную на данный момент память». В цифрах тенденция возрастания требований к емкости памяти выглядит так: увеличение в полтора раза каждые два года. Основная память современных ВМ и ВС формируется из СБИС полупроводниковых запоминающих устройств, главным образом динамических ОЗУ. Естественные требования к таким СБИС: высокие плотность упаковки запоминающих элементов и быстродействие, низкая стоимость.

Для СБИС памяти также подтверждается справедливость закона Мура и предсказанное им уменьшение темпов повышения плотности упаковки. В целом можно предсказать, что *число запоминающих элементов на кристалле будет возрастать в два раза каждые полтора года*.

С быстродействием СБИС памяти дело обстоит хуже. Высокая скорость процессоров уже давно находится в противоречии с относительной медлительностью запоминающих устройств основной памяти. Проблема постоянно усугубляется несоответствием темпов роста тактовой частоты процессоров и быстродействия памяти, и особых перспектив в этом плане пока не видно. Общая тенденция: *на двукратное уменьшение длительности цикла динамического ЗУ уходит примерно 15 лет*.

В плане снижения стоимости СБИС памяти перспективы весьма обнадеживающие. В течение достаточно длительного времени *стоимость в пересчете на один бит снижается примерно на 25–40 % в год*.

Перспективные направления исследований в области архитектуры вычислительных машин и систем

Основные направления исследований в области архитектуры ВМ и ВС можно условно разделить на две группы: эволюционные и революционные. К первой группе следует отнести исследования, целью которых является совершенствование методов реализации уже достаточно известных идей. Изыскания, условно названные революционными, направлены на создание совершенно новых архитектур, принципиально отличных от уже ставшей традиционной фон-неймановской архитектуры.

Большинство из исследований, относимых к эволюционным, связано с совершенствованием архитектуры микропроцессоров (МП). В принципе, кардинально новых архитектурных подходов в микропроцессорах сравнительно мало. Основные идеи, лежащие в основе современных МП, были выдвинуты много лет тому назад, но из-за несовершенства технологии и высокой стоимости реализации нашли применение только в больших универсальных ВМ (мэйнфреймах) и суперЭВМ. Наиболее значимые из изменений в архитектуре МП связаны с повышением уровня параллелизма на уровне команд (возможности одновременного выполнения нескольких команд). Здесь в первую очередь следует упомянуть конвейеризацию, суперскалярную обработку и архитектуру с командными словами сверхбольшой длины (VLIW). После успешного переноса на МП глобальных архитектурных подходов «больших» систем основные усилия исследователей теперь направлены на частные архитектурные изменения. Примерами таких эволюционных архитектурных изменений могут служить: усовершенствованные методы предсказания переходов в конвейере команд, повышение частоты успешных обращений к кэш-памяти за счет усложненных способов буферизации и т. п.

Наблюдаемые нами достижения в области вычислительных средств широкого применения пока обусловлены именно «эволюционными» исследованиями. Однако уже сейчас очевидно, что, оставаясь в рамках традиционных архитектур, мы довольно скоро натолкнемся на технологические ограничения. Один из путей преодоления технологического барьера лежит в области нетрадиционных подходов. Исследования, проводимые в этом направлении, по нашей классификации отнесены к «революционным». Справедливость такого утверждения подтверждается первыми образцами ВС с нетрадиционной архитектурой.

Оценивая перспективы эволюционного и революционного развития вычислительной техники, можно утверждать, что на ближайшее время наибольшего прогресса можно ожидать на пути использования идей параллелизма на всех его уровнях и создания эффективной иерархии запоминающих устройств.

Контрольные вопросы

1. По каким признакам можно разграничить понятия «вычислительная машина» и «вычислительная система»?
2. В чем заключается различие между функциональной и структурной организацией вычислительной машины? Как они влияют друг на друга?

3. Каким образом трансформируется понятие «структура» при его применении для отображения функциональной организации ВМ?
4. В чем состоит различие между «узкой» и «широкой» трактовкой понятия «архитектура вычислительной машины»?
5. Какой уровень детализации вычислительной машины позволяет определить, можно ли данную ВМ причислить к фон-неймановским?
6. По каким признакам выделяют поколения вычислительных машин?
7. Поясните определяющие идеи для каждого из этапов эволюции вычислительной техники.
8. Какой из принципов фон-неймановской концепции вычислительной машины можно рассматривать в качестве наиболее существенного?
9. Оцените достоинства и недостатки архитектур вычислительных машин с непосредственными связями и общей шиной.
10. Что понимается под номинальным и средним быстродействием ВМ?
11. Каким образом можно охарактеризовать производительность вычислительной машины?
12. Перечислите и охарактеризуйте основные способы построения критериев эффективности ВМ.
13. Какими способами можно произвести нормализацию частных показателей эффективности?
14. Сформулируйте основные тенденции развития интегральной схемотехники.
15. Какие выводы можно сделать, исходя из закона Мура?
16. Охарактеризуйте основные направления в дальнейшем развитии архитектуры вычислительных машин и систем.

Глава 2

Архитектура системы команд

Системой команд вычислительной машины называют полный перечень команд, которые способна выполнять данная ВМ. В свою очередь, под *архитектурой системы команд* (АСК) принято определять те средства вычислительной машины, которые видны и доступны программисту. АСК можно рассматривать как линию согласования нужд разработчиков программного обеспечения с возможностями создателей аппаратуры вычислительной машины (рис. 2.1).

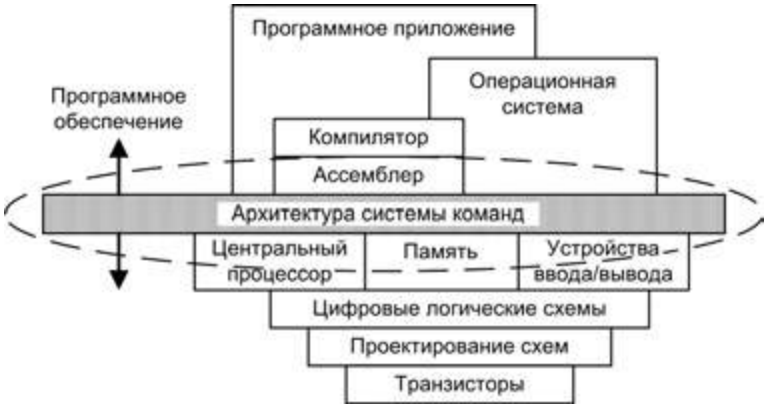


Рис. 2.1. Архитектура системы команд как интерфейс между программным и аппаратным обеспечением

В конечном итоге, цель тех и других — реализация вычислений наиболее эффективным образом, то есть за минимальное время, и здесь важнейшую роль играет правильный выбор архитектуры системы команд.

В упрощенной трактовке время выполнения программы ($T_{\text{выч}}$) можно определить через число команд в программе ($N_{\text{ком}}$), среднее количество тактов процессора, приходящихся на одну команду (CPI), и длительность тактового периода $\tau_{\text{пр}}$:

$$T_{\text{выч}} = N_{\text{ком}} \times CPI \times \tau_{\text{пр}}.$$

Каждая из составляющих выражения зависит от одних аспектов архитектуры системы команд и, в свою очередь, влияет на другие (рис. 2.2), что свидетельствует о необходимости чрезвычайно ответственного подхода к выбору АСК.

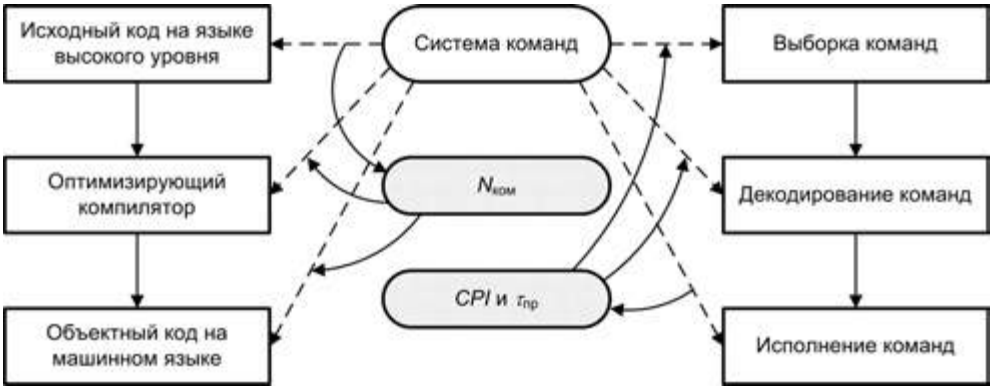


Рис. 2.2. Взаимосвязь между системой команд и факторами, определяющими эффективность вычислений

Общая характеристика архитектуры системы команд вычислительной машины складывается из ответов на следующие вопросы:

1. Какого вида данные будут представлены в вычислительной машине и в какой форме?
2. Где эти данные могут храниться помимо основной памяти?
3. Каким образом будет осуществляться доступ к данным?
4. Какие операции могут быть выполнены над данными?
5. Сколько операндов может присутствовать в команде?
6. Как будет определяться адрес очередной команды?
7. Каким образом будут закодированы команды?

Предметом данной главы является обзор наиболее распространенных архитектур системы команд, как в описательном плане, так и с позиций эффективности.

Классификация архитектур системы команд

В истории развития вычислительной техники как в зеркале отражаются изменения, происходившие во взглядах разработчиков на перспективность той или иной архитектуры системы команд. Сложившуюся на настоящий момент ситуацию в области АСК иллюстрирует рис. 2.3.

Среди мотивов, чаще всего предопределяющих переход к новому типу АСК, остановимся на двух наиболее существенных. Первый — это состав операций, выполняемых вычислительной машиной, и их сложность. Второй — место хранения операндов, что влияет на количество и длину адресов, указываемых в адресной части команд обработки данных. Именно эти характеристики взяты в качестве показателей классификации архитектур системы команд.

Рис. 2.3. Хронология развития архитектур системы команд

Классификация по составу и сложности команд

Современная технология программирования ориентирована на языки высокого уровня (ЯВУ), главная цель которых — облегчить процесс программирования. Но переход к ЯВУ породил серьезную проблему: сложные операторы, характерные для ЯВУ, существенно отличаются от простых машинных операций, реализуемых в большинстве вычислительных машин. Следствием такого несоответствия становится недостаточно эффективное выполнение программ на ВМ. Проблема получила название *семантического разрыва*, а для ее разрешения разработчики вычислительных машин в настоящее время выбирают один из трех подходов и, соответственно, один из трех типов АСК:

- архитектуру с полным набором команд: CISC (Complex Instruction Set Computer);
- архитектуру с сокращенным набором команд: RISC (Reduced Instruction Set Computer);
- архитектуру с командными словами сверхбольшой длины: VLIW (Very Long Instruction Word).

В *CISC-архитектуре* семантический разрыв преодолевается за счет расширения системы команд, дополнения ее сложными командами, семантически аналогичными операторам ЯВУ. Основоположником CISC-архитектуры считается компания IBM, которая начала применять данный подход с семейства машин IBM 360

и продолжает его в своих мощных современных универсальных ВМ (мэйнфреймах). Аналогичный подход характерен и для компании Intel в ее микропроцессорах серии x86. Для CISC-архитектуры типичны:

- наличие в процессоре сравнительно небольшого числа регистров общего назначения;
- большое количество машинных команд, часть из которых аппаратно реализуют сложные операторы ЯВУ;
- разнообразие способов адресации операндов;
- множество форматов команд различной разрядности;
- наличие команд, где обработка совмещается с обращением к памяти.

К типу CISC можно отнести практически все ВМ, выпускавшиеся до середины 1980-х годов, и значительную часть производящихся в настоящее время. Рассмотренный способ решения проблемы семантического разрыва вместе с тем ведет к усложнению аппаратуры ВМ, главным образом, устройства управления, что, в свою очередь, негативно сказывается на производительности ВМ в целом. Это обстоятельство побудило более внимательно проанализировать программы, получаемые после компиляции с ЯВУ. Был предпринят комплекс исследований [99, 115, 128, 151], в результате которых обнаружилось, что доля дополнительных команд, эквивалентных операторам ЯВУ, в общем объеме программ не превышает 10–20%, а для некоторых наиболее сложных команд даже 0,2%. В то же время объем аппаратных средств, требуемых для реализации таких дополнительных команд, возрастает весьма существенно. Так, емкость микропрограммной памяти, хранящей микропрограммы выполнения всех команд ВМ, из-за введения сложных команд может увеличиваться на 60%.

Детальный анализ результатов упомянутых исследований привел к серьезному пересмотру традиционных решений, следствием чего стало появление *RISC-архитектуры*. Термин RISC впервые был использован Д. Паттерсоном и Д. Дитцелем в 1980 году [128]. Идея заключается в ограничении списка команд ВМ наиболее часто используемыми простейшими командами, оперирующими данными, размещенными только в регистрах процессорах. Обращение к памяти допускается лишь с помощью специальных команд чтения и записи. Резко уменьшено количество форматов команд и способов указания адресов операндов. Эти меры позволили существенно упростить аппаратные средства ВМ и повысить их быстродействие. RISC-архитектура разрабатывалась таким образом, чтобы уменьшить $T_{\text{выч}}$ за счет сокращения *CPI* и $\tau_{\text{пр}}$. Оказалось, что реализация сложных команд за счет последовательности из простых, но быстрых RISC-команд не менее эффективна, чем аппаратный вариант сложных команд в CISC-архитектуре.

Элементы RISC-архитектуры впервые появились в вычислительных машинах CDC 6600 и суперЭВМ компании Cray Research. Достаточно успешно реализуется RISC-архитектура и в современных ВМ.

Отметим, что в последнее время в микропроцессорах компаний Intel и AMD широко используются идеи, свойственные RISC-архитектуре, так что многие различия между CISC и RISC постепенно стираются.

Помимо CISC- и RISC-архитектур, в общей классификации был упомянут еще один тип АСК — архитектура с командными словами сверхбольшой длины (VLIW). Концепция VLIW базируется на RISC-архитектуре, но в ней несколько простых RISC-команд объединяются в одну сверхдлинную команду и выполняются параллельно. В плане АСК архитектура VLIW сравнительно мало отличается от RISC. Появился лишь дополнительный уровень параллелизма вычислений, в силу чего архитектуру VLIW логичнее адресовать не к вычислительным машинам, а к вычислительным системам.

Таблица 2.1. Сравнительная оценка CISC-, RISC- и VLIW-архитектур

Характеристика	CISC	RISC	VLIW
Длина команды	Варьируется	Единая	Единая
Расположение полей в команде	Варьируется	Неизменное	Неизменное
Количество регистров	Несколько (часто специализированных)	Много регистров общего назначения	Много регистров общего назначения
Доступ к памяти	Может выполняться как часть команд различных типов	Выполняется только специальными командами	Выполняется только специальными командами

Таблица 2.1 позволяет оценить наиболее существенные различия в архитектурах типа CISC, RISC и VLIW.

Классификация по месту хранения операндов

Количество команд и их сложность, безусловно, являются важнейшими факторами, однако не меньшую роль при выборе АСК играет ответ на вопрос о том, где могут храниться операнды и каким образом к ним осуществляется доступ. С этих позиций различают следующие виды архитектур системы команд:

- стековую;
- аккумуляторную;
- регистровую;
- с выделенным доступом к памяти.

Выбор той или иной архитектуры влияет на принципиальные моменты: сколько адресов будет содержать адресная часть команд, какова будет длина этих адресов, насколько просто будет происходить доступ к операндам и какой, в конечном итоге, будет общая длина команд.

Стековая архитектура

Стеком называется память, по своей структурной организации отличная от основной памяти ВМ. Принципы построения стековой памяти детально рассматриваются позже, здесь же выделим только те аспекты, которые требуются для пояснения особенностей АСК на базе стека.

Стек образует множество логически взаимосвязанных ячеек (рис. 2.4), взаимодействующих по принципу «последним вошел, первым вышел» (LIFO, Last In First Out).

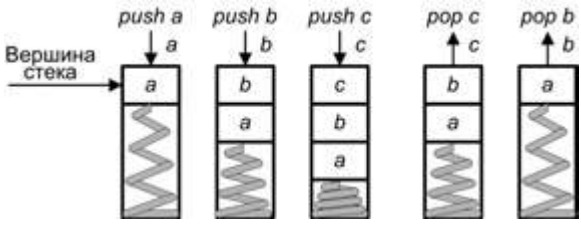


Рис. 2.4. Принцип действия стековой памяти

Верхнюю ячейку называют *вершиной стека*. Для работы со стеком предусмотрены две операции: *push* (проталкивание данных в стек) и *pop* (выталкивание данных из стека). Запись возможна только в верхнюю ячейку стека, при этом вся хранящаяся в стеке информация предварительно проталкивается на одну позицию вниз. Чтение допустимо также только из вершины стека. Извлеченная информация удаляется из стека, а оставшееся его содержимое продвигается вверх. В вычислительных машинах, где реализована АСК на базе стека (их обычно называют стековыми), операнды перед обработкой помещаются в две верхних ячейки стековой памяти. Результат операции заносится в стек. Принцип действия стековой машины поясним на примере вычисления выражения $(a + b) * (c + d) - e$.

При описании вычислений с использованием стека обычно используется иная форма записи математических выражений, известная как обратная польская запись (обратная польская нотация), которую предложил польский математик Я. Лукашевич. Особенность ее в том, что в выражении отсутствуют скобки, а знак операции располагается не между операндами, а следует за ними (постфиксная форма).

Преобразование традиционной записи выражения в постфиксную удобно выполнять с помощью вспомогательного стека. Назовем его стеком последовательности операций (СПО), чтобы не путать со стеком вычислительной машины. Для работы с СПО зададим приоритеты операций исходного (традиционного) выражения (табл. 2.2). В табл. 2.2 нулем обозначен минимальный, а четверкой — максимальный приоритет.

Таблица 2.2. Приоритеты операций для работы с СПО

Операция	**Символ операции**	**Приоритет**
Открывающая скобка	(	0
Закрывающая скобка	)	1
Сложение | вычитание	+ | –	2
Умножение | деление	* | /	3
Возведение в степень	**	4

По мере просмотра традиционной строки в СПО помещаются встречающиеся символы операций, которые в дальнейшем (по определенным правилам) переносятся в постфиксную строку. Формирование строки с выражением в обратной польской нотации осуществляется в соответствии со следующим алгоритмом:

1. Традиционная (входная) строка с выражением просматривается слева направо.
2. Если очередной символ входной строки — операнд, он сразу переписывается в постфиксную строку.
3. Левая скобка, а также символ математической операции в случае, если СПО пуст, всегда заносится в СПО.
4. Когда при просмотре встречается правая скобка, символ, находящийся на вершине СПО, выталкивается из СПО и заносится в постфиксную строку. Процедура повторяется вплоть до появления на вершине СПО левой скобки. Когда это происходит, обе скобки взаимно уничтожаются, при этом левая скобка из СПО удаляется.
5. Если СПО пуст или символ операции во входной строке имеет более высокий приоритет, чем символ на вершине СПО, символ операции из входной строки заталкивается в СПО.
6. Если приоритет символа во входной строке равен или ниже приоритета символа на вершине СПО, символ из вершины СПО выталкивается в постфиксную строку. Процедура повторяется до тех пор, пока на вершине СПО не появится символ с меньшим приоритетом, либо левая скобка, либо СПО станет пустым. После этого символ из входной строки заносится в СПО.
7. При достижении последнего символа входной строки содержимое СПО последовательно выталкивается в постфиксную строку.

Процесс получения обратной польской записи для выражения $(a + b) * (c + d) - e$ представлен в табл. 2.3.

Таблица 2.3. Формирование обратной польской записи для выражения (a + b) * (c + d) – e

Просматриваемый символ	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Входная строка	(	*a*	+	*b*	)	*	(	*c*	+	*d*	)	–	*e*	
Состояние стека последовательности операций	(	((	+ (	+ (		*	(*	(*	+ (*	+ (*	*	–	–	
Постфиксная строка		*a*		*b*	+			*c*		*d*	+	*	*e*	–

Таким образом, рассмотренное выше выражение в польской записи имеет вид: *ab* + *cd* + * *e* –. Данная форма записи однозначно определяет порядок загрузки операндов и операций в стек (рис. 2.5).

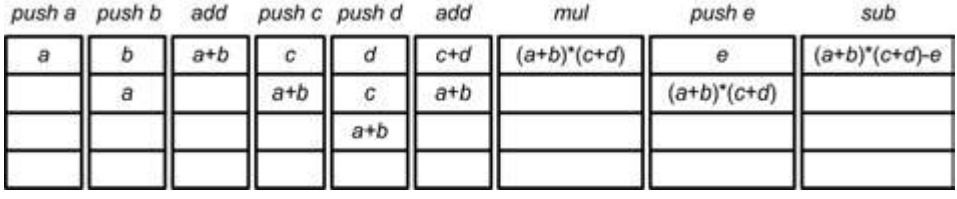


Рис. 2.5. Последовательность вычисления выражения *ab*+*cd*+**e*– на вычислительной машине со стековой архитектурой

Основные узлы и информационные тракты одного из возможных вариантов ВМ на основе стековой АСК показаны на рис. 2.6.

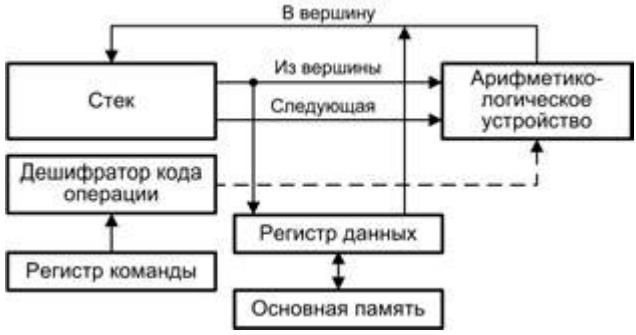


Рис. 2.6. Архитектура вычислительной машины на базе стека

Информация может быть занесена в вершину стека из памяти или из АЛУ. Для записи в стек содержимого ячейки памяти с адресом *x* выполняется команда *push x*, по которой информация считывается из ячейки памяти, заносится в регистр данных, а затем проталкивается в стек. Результат операции из АЛУ заносится в вершину стека автоматически.

Сохранение содержимого вершины стека в ячейке памяти с адресом *x* производится командой *pop x*. По этой команде содержимое верхней ячейки стека подается на шину, с которой и производится запись в ячейку *x*, после чего все содержимое стека проталкивается на одну позицию вверх.

Для выполнения арифметической или логической операции на вход АЛУ подается информация, считанная из двух верхних ячеек стека (при этом содержимое стека продвигается на две позиции вверх, то есть операнды из стека удаляются). Результат операции заталкивается в вершину стека. Возможен вариант, когда результат сразу же переписывается в память с помощью автоматически выполняемой операции *pop x*.

Верхние ячейки стековой памяти, где хранятся операнды и куда заносится результат операции, как правило, делают более быстродействующими и размещают в процессоре, в то время как остальная часть стека может располагаться в основной памяти и частично даже на магнитном диске.

К достоинствам АСК на базе стека следует отнести возможность сокращения адресной части команд, поскольку все операции производятся через вершину стека, то есть адреса операндов и результата в командах арифметической и логической обработки информации указывать не нужно. Код программы получается компактным. Достаточно просто реализуется декодирование команд.

С другой стороны, стековая АСК по определению не предполагает произвольного доступа к памяти, из-за чего компилятору трудно создать эффективный программный код, хотя создание самих компиляторов упрощается. Кроме того, стек становится «узким местом» ВМ в плане повышения производительности. В силу

упомянутых причин, данный вид АСК долгое время считался неперспективным и встречался, главным образом, в вычислительных машинах 1960-х годов.

Последние события в области вычислительной техники свидетельствуют о возрождении интереса к стековой архитектуре ВМ. Связано это с популярностью языка Java и расширением сферы применения языка Forth, семантике которых наиболее близка именно стековая архитектура. Среди современных ВМ со стековой АСК особо следует отметить стековую машину IGNITE компании Patriot Scientist, которую ее авторы считают представителем нового вида АСК — *архитектурой с безоперандным набором команд*. Для обозначения таких ВМ они предлагают аббревиатуру ROSC (Removed Operand Set Computer). ROSC-архитектура заложена и в некоторые российские проекты, например разработки ИТФ «Технофорт». Строго говоря, по своей сути ROSC мало отличается от традиционной архитектуры на базе стека, и выделение ее в отдельный вид представляется не вполне обоснованным.

Аккумуляторная архитектура

Архитектура на базе аккумулятора исторически возникла одной из первых. В ней для хранения одного из операндов арифметической или логической операции в процессоре имеется выделенный регистр — *аккумулятор*. В этот же регистр заносится и результат операции. Поскольку адрес одного из операндов предопределен, в командах обработки достаточно явно указать местоположение только второго операнда. Изначально оба операнда хранятся в основной памяти, и до выполнения операции один из них нужно загрузить в аккумулятор. После выполнения команды результат заносится в аккумулятор, и если этот результат не является операндом для последующей команды, то его требуется сохранить в ячейке памяти.

Типичная архитектура ВМ на базе аккумулятора показана на рис. 2.7.

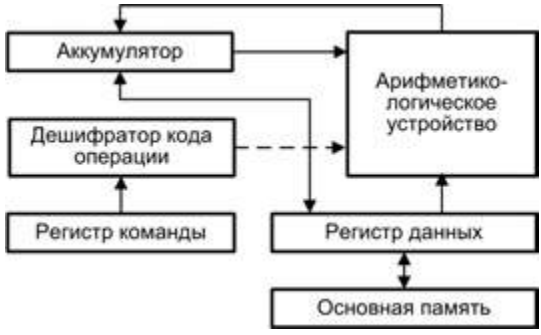


Рис. 2.7. Архитектура вычислительной машины на базе аккумулятора

Для загрузки в аккумулятор содержимого ячейки *x* предусмотрена команда загрузки *load x*. По этой команде информация считывается из ячейки памяти *x*, выход памяти подключается ко входам аккумулятора, и происходит занесение считанных данных в аккумулятор.

Запись содержимого аккумулятора в ячейку *x* осуществляется командой сохранения *store x*, при выполнении которой выходы аккумулятора подключаются к шине, после чего информация с шины записывается в память.

Для выполнения операции в АЛУ производится считывание одного из операндов из памяти в регистр данных. Второй операнд находится в аккумуляторе. Выходы регистра данных и аккумулятора подключаются к соответствующим входам АЛУ. По окончании предписанной операции результат с выхода АЛУ заносится в аккумулятор.

Достоинствами аккумуляторной АСК можно считать короткие команды и простоту декодирования команд. Однако наличие всего одного регистра порождает многократные обращения к основной памяти.

АСК на базе аккумулятора была популярна в ранних ВМ, таких, например, как IBM 7090, DEC PDP-8.

Регистровая архитектура

В машинах данного типа процессор включает в себя массив регистров общего назначения (РОН). Разрядность регистров обычно совпадает с размером машинного слова. К любому регистру можно обратиться, указав его номер. Количество РОН в архитектурах типа CISC обычно невелико (от 8 до 32), и для представления номера конкретного регистра необходимо не более пяти разрядов, благодаря чему в адресной части команд обработки допустимо одновременно указать номера двух, а зачастую и трех регистров (двух регистров операндов и регистра результата). RISC-архитектура предполагает использование существенно большего числа РОН (до нескольких сотен), однако типичная для таких ВМ длина команды (обычно 32 разряда) позволяет определить в команде до трех регистров.

Регистровая архитектура допускает расположение операндов как в регистрах, так и в основной памяти, поэтому в рамках данной АСК выделяют три формата команд обработки:

- регистр-регистр;
- регистр-память;
- память-память.

В формате «регистр-регистр» операнды могут находиться только в регистрах. В них же засылается и результат. Формат «регистр-память» предполагает, что один из операндов размещается в регистре, а второй в основной памяти. Результат обычно замещает один из операндов. В командах формата «память-память» оба операнда хранятся в основной памяти. Результат заносится в память. Каждому из форматов соответствуют свои достоинства и недостатки (табл. 2.4).

В выражениях вида (*m*, *n*), приведенных в первом столбце таблицы, *m* означает количество операндов, хранящихся в основной памяти, а *n* — общее число операндов в команде арифметической или логической обработки.

Формат «регистр-регистр» является основным в вычислительных машинах типа RISC. Команды формата «регистр-память» характерны для CISC-машин. Наконец,

формат «память-память» считается неэффективным, хотя и остается в наиболее сложных моделях машин класса CISC.

Таблица 2.4. Сравнительная оценка вариантов размещения операндов

Вариант	Достоинства	Недостатки
Регистр-регистр (0, 3)	Простота реализации, фиксированная длина команд, простая модель формирования объектного кода при компиляции программ, возможность выполнения всех команд за одинаковое количество тактов	Большая длина объектного кода, из-за фиксированной длины команд часть разрядов в коротких командах не используется
Регистр-память (1, 2)	Данные могут быть доступны без загрузки в регистры процессора, простота кодирования команд, объектный код получается достаточно компактным	Потеря одного из операндов при записи результата, длинное поле адреса памяти в коде команды сокращает место под номер регистра, что ограничивает общее число РОН. CPI зависит от места размещения операнда
Память-память (3, 3)	Компактность объектного кода, малая потребность в регистрах для хранения промежуточных данных	Разнообразие форматов команд и времени их исполнения, низкое быстродействие из-за обращения к памяти

Возможную структуру и информационные тракты вычислительной машины с регистровой архитектурой системы команд иллюстрирует рис. 2.8.

Рис. 2.8. Архитектура вычислительной машины на базе регистров общего назначения

Операции загрузки регистров из памяти и сохранения содержимого регистров в памяти идентичны таким же операциям с аккумулятором. Отличие состоит в этапе выбора нужного регистра, обеспечиваемого соответствующими селекторами.

Выполнение операции в АЛУ включает в себя:

- определение местоположения первого операнда (регистр или память);
- выбор регистра первого операнда или считывание первого операнда из памяти;

- определение местоположения второго операнда (регистр или память);
- выбор регистра второго операнда или считывание второго операнда из памяти;
- подачу на вход АЛУ операндов и выполнение операции;
- определение местоположения результата (регистр или память);
- выбор регистра результата или ячейки памяти и занесение результата операции из АЛУ.

Обратим внимание на то, что между АЛУ и регистровым файлом должны быть три шины. Две из трех шин, расположенных между массивом РОН и АЛУ, обеспечивают передачу в арифметико-логическое устройство операндов, хранящихся в регистрах общего назначения или ячейках памяти. Третья служит для занесения результата в выделенный для этого регистр или ячейку памяти. Эти же шины позволяют загрузить в регистры содержимое ячеек основной памяти и сохранить в ОП информацию, находящуюся в РОН.

К достоинствам регистровых АСК следует отнести: компактность получаемого кода, высокую скорость вычислений за счет замены обращений к основной памяти на обращения к быстрым регистрам. С другой стороны, данная архитектура требует более длинных команд по сравнению с аккумуляторной архитектурой.

В наши дни этот вид архитектуры системы команд является преобладающим, в частности такую АСК имеют современные персональные компьютеры.

Архитектура с выделенным доступом к памяти

В архитектуре с выделенным доступом к памяти обращение к основной памяти возможно только с помощью двух специальных команд: *load* и *store*. В английской транскрипции данную архитектуру называют *Load/Store architecture*. Команда *load* (загрузка) обеспечивает считывание значения из основной памяти и занесение его в регистр процессора (в команде обычно указывается адрес ячейки памяти и номер регистра). Пересылка информации в противоположном направлении производится командой *store* (сохранение). Операнды во всех командах обработки информации могут находиться только в регистрах процессора (чаще всего в регистрах общего назначения). Результат операции также заносится в регистр.

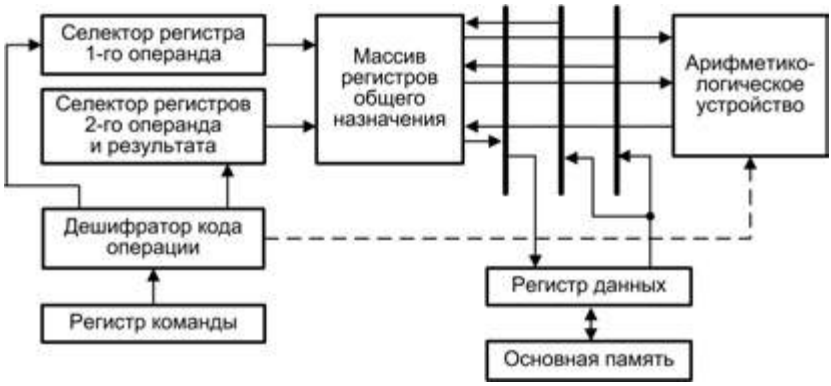


Рис. 2.9. Архитектура вычислительной машины с выделенным доступом к памяти

В архитектуре отсутствуют команды обработки, допускающие прямое обращение к основной памяти. Допускается наличие в АСК ограниченного числа команд, где операнд является частью кода команды.

Состав и информационные тракты ВМ с выделенным доступом к памяти показаны на рис. 2.9.

АСК с выделенным доступом к памяти характерна для всех вычислительных машин с RISC-архитектурой. К достоинствам архитектуры следует отнести простоту декодирования и исполнения команды.

Типы и форматы операндов

Машинные команды оперируют данными, которые в этом случае принято называть *операндами*. К наиболее общим (базовым) типам операндов можно отнести: числа, символы и логические данные. Помимо них, ВМ обеспечивает обработку и более сложных информационных единиц: графических изображений, аудио-, видео- и анимационной информации. Такая информация является производной от базовых типов данных и хранится в виде файлов на внешних запоминающих устройствах. Для каждого типа данных в ВМ предусмотрены определенные форматы.

Числовая информация

Такого рода данные можно классифицировать следующим образом:

- числа с фиксированной запятой (целые, дробные, смешанные);
- числа с плавающей запятой;
- логические значения.

Числа в форме с фиксированной запятой

Представление числа A в *форме с фиксированной запятой* (ФЗ), которую иногда называют также *естественной формой*, включает в себя знак числа и его модуль в q-ричном коде. Здесь q — *основание системы счисления* или *база*. Для современных ВМ характерна двоичная система ($q = 2$), но иногда используются также восьмеричная ($q = 8$) или шестнадцатеричная ($q = 16$) системы счисления. Запятую в записи числа называют соответственно двоичной, восьмеричной или шестнадцатеричной. Знак положительного числа кодируется двоичной цифрой 0, а знак отрицательного числа — цифрой 1.

Числам с ФЗ соответствует запись вида $A = \pm a_{n-1} \ldots a_1 a_0 a_{-1} a_{-2} \ldots a_{-r}$. Отрицательные числа обычно представляются в дополнительном коде. Разряд кода числа, в котором размещается знак, называется *знаковым разрядом кода*. Разряды, где располагаются значащие цифры числа, называются *цифровыми разрядами кода*. Знаковый разряд размещается левее старшего цифрового разряда. Положение запятой одинаково для всех чисел и в процессе решения задач не меняется. Хотя запятая и фиксируется, в коде числа она никак не выделяется, а только подразумевается. В общем случае разрядная сетка ВМ для размещения чисел в форме с ФЗ имеет вид, представленный на рис. 2.10, где m разрядов используются для записи целой части числа и r разрядов — для дробной части.

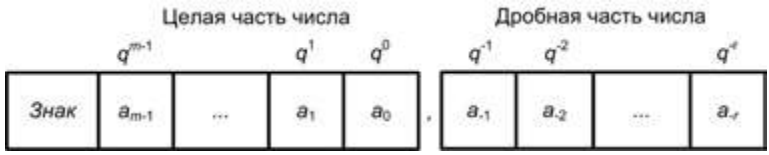


Рис. 2.10. Формат представления чисел с фиксированной запятой

Если число является смешанным (содержит целую и дробную части), оно обрабатываются как целое, хотя и не является таковым (в этом случае применяют термин *масштабируемое целое*). Обработка смешанных чисел в ВМ встречается крайне редко. Как правило, используются ВМ с дробной ($m = 0$) либо целочисленной ($r = 0$) арифметикой.

При фиксации запятой перед старшим цифровым разрядом (рис. 2.11) могут быть представлены только правильные дроби. Для ненулевых чисел возможны два варианта представления (нулевому значению соответствуют нули во всех разрядах): со знаком и без знака. Фиксация запятой перед старшим разрядом используется при обработке мантисс чисел в форме с плавающей запятой (рассматривается ниже).

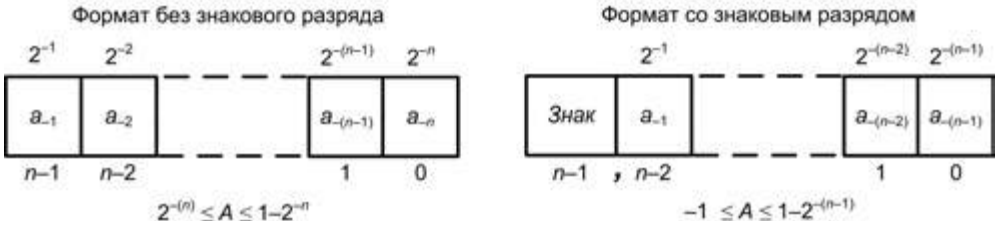


Рис. 2.11. Представление дробных чисел в формате ФЗ

При фиксации запятой после младшего разряда представимы лишь целые числа. Это наиболее распространенный способ. Здесь также возможны числа со знаком и без знака (рис. 2.12).

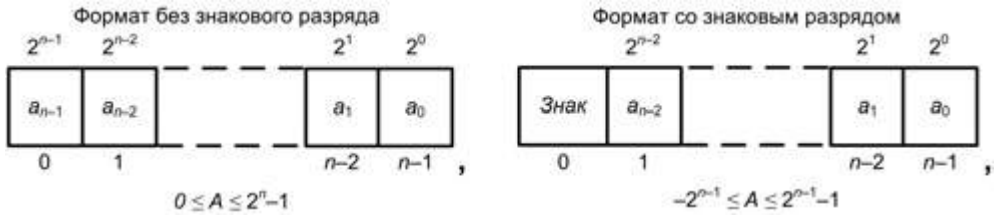


Рис. 2.12. Представление целых чисел в формате ФЗ

На рис. 2.13 приведены целочисленные форматы с фиксированной запятой, принятые в 64-разрядных процессорах.

Представление чисел в формате ФЗ упрощает аппаратурную реализацию ВМ и сокращает время выполнения машинных операций, однако при решении задач необходимо постоянно следить за тем, чтобы все исходные данные, промежуточные и окончательные результаты не выходили за допустимый диапазон формата, иначе возможно переполнение разрядной сетки, и результат вычислений будет неверным.

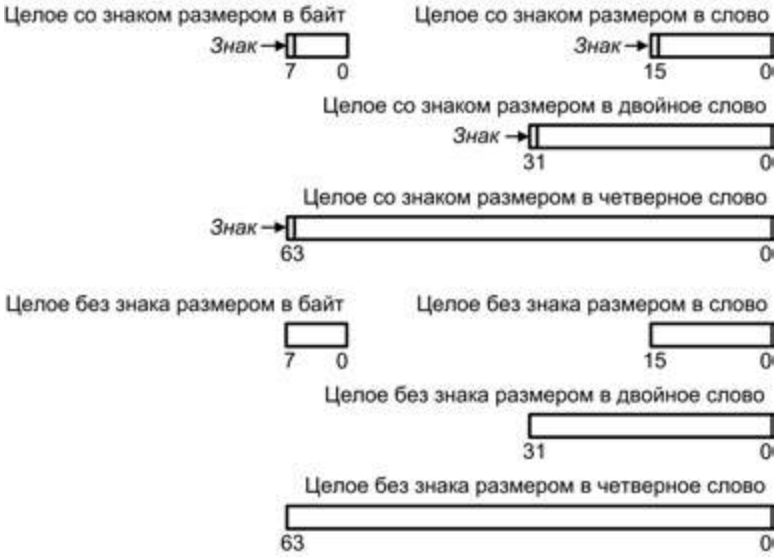


Рис. 2.13. Целочисленные форматы 64-разрядных процессоров

Упакованные целые числа

В АСК современных микропроцессоров имеются команды, оперирующие целыми числами, представленными в упакованном виде. Связано это с обработкой мультимедийной информации. Упаковка предполагает объединение в пределах достаточно длинного слова нескольких целых чисел меньшей разрядности, а соответствующие команды обрабатывают все эти числа параллельно. Такие команды появились в рамках технологии MMX (MultiMedia eXtension) фирмы Intel и похожей технологии 3DNow! фирмы AMD. Со временем сформировалась единая технология, известная под аббревиатурой SSE (Streaming SIMD Extensions). В микропроцессорах последних поколений — это версия SSE4, но уже ведется разработка варианта SSE5. В SSE4 за основу принято 128-разрядное слово и предусмотрены четыре формата упакованных целых чисел (рис. 2.14).

Байты в формате упакованных байтов нумеруются от 0 до 15, причем байт 0 располагается в младших разрядах 128-разрядного слова. Аналогичная система нумерации и размещения упакованных чисел применяется для 16-разрядных (номера 0–7), 32-разрядных (номера 0–3) и 64-разрядных (номера 0–1) целых чисел.

Рис. 2.14. Форматы упакованных целых чисел в технологии SSE4

Десятичные числа

В ряде задач, главным образом, учетно-статистического характера, приходится иметь дело с хранением, обработкой и пересылкой десятичной информации. Особенность таких задач состоит в том, что обрабатываемые числа могут состоять из различного и весьма большого количества десятичных цифр. Традиционные методы обработки с переводом исходных данных в двоичную систему счисления и обратным преобразованием результата зачастую сопряжены с существенными накладными расходами. По этой причине в ВМ применяются иные специальные формы представления десятичных данных. В их основу положен принцип кодирования каждой десятичной цифры эквивалентным двоичным числом из четырех битов (тетрадой), то есть так называемым двоично-десятичным кодом (BCD — Binary Coded Decimal). Обычно используется стандартная кодировка 8421, где цифры в обозначении кодировки означают веса разрядов. Десятичные цифры 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 кодируются двоичными комбинациями 0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1000, 1001 соответственно. Оставшиеся шесть комбинаций (1010, 1011, 1100, 1101, 1110, 1111) используются для представления знаков «+» и «–», а также возможных служебных символов. Знак «+» принято обозначать как 1100 (C_{16}), а знак «–» — как 1101 (D_{16}). Такое соглашение принято исходя из того, что обозначение соответствующих шестнадцатеричных цифр можно рассматривать как аббревиатуру бухгалтерских терминов «кредит» (Credit) и «дебет» (Debit), а именно в бухгалтерских расчетах чаще всего используется рассматриваемая форма представления информации. Другие допустимые обозначения знаков — это 1010 (A_{16}) или 1110 (E_{16}) для «+» и 1011 (B_{16}) для «–». Иногда допускается представление десятичных чисел без знака, и тогда в позиции, отводимой под знак, записывается комбинация 1111 (F_{16}).

В вычислительных машинах нашли применение два формата представления десятичных чисел (все числа рассматриваются как целые): упакованный и *зонный*. В обоих форматах каждая десятичная цифра представляется двоичной тетрадой, то есть заменяется двоично-десятичным кодом.

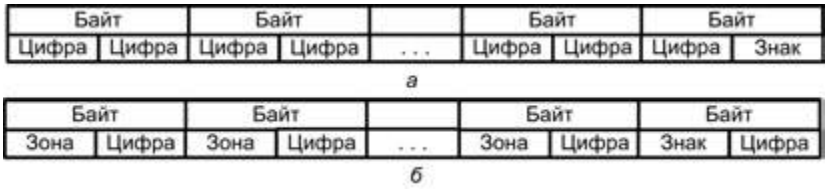


Рис. 2.15. Форматы десятичных чисел: *а* — упакованный; *б* — зонный

Наиболее распространен упакованный формат (рис. 2.15, *а*), позволяющий не только хранить десятичные числа, но и производить над ними арифметические операции. В данном формате запись числа имеет вид цепочки байтов, где каждый байт содержит коды двух десятичных цифр. Правая тетрада младшего байта предназначается для записи знака числа. Десятичное число должно занимать целое количество байтов. Если это условие не выполняется, то четыре старших двоичных разряда левого байта заполняются нулями. Так, представление числа –7396 в упакованном формате имеет вид, приведенный на рис 2.16.

Байт		Байт		Байт	
0	7	3	9	6	Минус
0000	0111	0011	1001	0110	1101

Рис. 2.16. Представление числа –7396 в упакованном формате

Зонный формат (рис. 2.15, *б*) распространен, главным образом, в больших универсальных ВМ семейства IBM 360/370/390. В нем под каждую цифру выделяется один байт, где младшие четыре разряда отводятся под код цифры, а в старшую тетраду (поле зоны) записывается специальный код «зона», не совпадающий с кодами цифр и знаков. В IBM 360/370/390 это код $1111_2 = F_{16}$. Исключение составляет байт, содержащий младшую цифру десятичного числа, где в поле зоны хранится знак числа. На рис. 2.17 показана запись числа –7396 в зонном формате. В некоторых ВМ принят вариант зонного формата, где поле зоны заполняется нулями.

Байт		Байт		Байт		Байт	
Зона	7	Зона	3	Зона	9	Минус	6
1111	0111	1111	0011	1111	1001	1101	0110

Рис. 2.17. Представление числа –7396 в зонном формате

Размещение знака в младшем байте, как в зонном, так и в упакованном представлениях, позволяет задавать десятичные числа произвольной длины и передавать их в виде цепочки байтов. В этом случае знак указывает, что байт, в котором он содержится, является последним байтом данного числа, а следующий байт последовательности — это старший байт очередного числа.

Рассмотренный вариант двоично-кодированного представления десятичных цифр с весами 8421 наиболее распространен, но не является единственным. Возможные иные схемы кодирования приведены в табл. 2.5.

Таблица 2.5. Варианты двоично-кодированного представления десятичных цифр

Цифра	BCD 8 4 2 1	Excess-3 (код Стибица)	BCD 2 4 2 1 (код Айкена)	BCD 8 4 -2 -1	BCD 8 4 2 1 (IBM 702, 705)
0	0000	0011	0000	0000	1010
1	0001	0100	0001	0111	0001
2	0010	0101	0010	0110	0010
3	0011	0110	0011	0101	0011
4	0100	0111	0100	0100	0100
5	0101	1000	1011	1011	1011
6	0110	1001	1100	1010	0110
7	0111	1010	1101	1001	0111
8	1000	1011	1110	1000	1000
9	1001	1100	1111	1111	1001

Использование 4-х двоичных цифр для представления одной десятичной цифры по своей сути избыточно. В то же время при иной системе кодирования для представления десятичного числа из трех цифр достаточно 10 двоичных разрядов. Два наиболее известных варианта такого «экономичного» кодирования — код Чен-Хо (Tien Chi Chen, Irving T. Ho) и плотно упакованный десятичный код (DPD — Densely Packed Decimal).

В обоих вариантах каждая десятичная цифра классифицируется по значению старшего бита в ее представлении в коде BCD на «маленькую» 0–7 (0000_2–0111_2) или «большую» 8–9 (1000_2–1001_2). Для идентификации маленькой цифры (М) достаточно 3 бита, а большой (Б) — одного бита.

При таком представлении возможны следующие комбинации из трех десятичных цифр:

- М + М + М (требуется 9 битов для цифр и остается 1 бит для идентификации этой комбинации);
- М + М + Б, или М + Б + М, или Б + М + М (требуется 7 битов для цифр и остается 3 бита для идентификации этих комбинаций);
- М + Б + Б, или Б + М + Б, или Б + Б + М (требуется 5 битов для цифр и остается 5 битов для идентификации этих комбинаций);
- Б + Б + Б (3 бита для цифр и 7 битов для индикации этой комбинации, хотя нужно только 5).

Для идентификации конкретной комбинации используются те из 10 битов, которые остались свободными после представления закодированных значений цифр. Различие в методах кодирования Чен-Хо и DPD состоит в способе формирования идентификатора комбинации. Более детально с данными видами кодировки десятичных чисел можно ознакомиться в [64, 72]. В табл. 2.6 приведены примеры кодировки троек десятичных чисел в кодах BCD, Чен-Хо и DPD.

Таблица 2.6. Примеры представления десятичных чисел в кодировках BCD, Чен-Хо и DPD

Десятичное число	BCD 8421	Чен-Хо	DPD
005	0000 0000 0101	000 000 0101	000 000 0101
009	0000 0000 1001	110 000 0001	000 000 1001
055	0000 0101 0101	000 010 1101	000 101 0101
079	0000 0111 1001	110 011 1001	000 111 1001
080	0000 1000 0000	101 000 0000	000 000 1010
099	0000 1001 1001	111 000 1001	000 101 1111
555	0101 0101 0101	010 110 1101	101 101 0101
999	1001 1001 1001	111 111 1001	001 111 1111

Числа в форме с плавающей запятой

От недостатков ФЗ в значительной степени свободна форма представления чисел с *плавающей запятой* (ПЗ), известная также под названиями *нормальной* или *полулогарифмической* формы. В данном варианте каждое число разбивается на две группы цифр. Первая группа цифр называется *мантиссой*, вторая — *порядком*. Число представляется в виде произведения $X = \pm mq^{\pm p}$, где m — мантисса числа X, p — порядок числа, q — основание системы счисления.

Для представления числа в форме с ПЗ требуется задать знаки мантиссы и порядка, их модули в q-ричном коде, а также основание системы счисления (рис. 2.18). Нормальная форма неоднозначна, так как взаимное изменение m и p приводит к «плаванию» запятой, чем и обусловлено название этой формы.

Рис. 2.18. Форма представления чисел с плавающей запятой

Диапазон и точность представления чисел с ПЗ зависят от числа разрядов, отводимых под порядок и мантиссу. На рис. 2.19 показаны диапазоны разрядностей порядка и мантиссы, характерные для известных ВМ.

Помимо разрядности порядка и мантиссы, диапазон представления чисел зависит и от основания используемой системы счисления, которое может быть отличным от 2. Например, в универсальных ВМ (мэйнфреймах) фирмы IBM используется база 16. Это позволяет при одинаковом количестве битов, отведенных под порядок, представлять числа в большем диапазоне. Так, если поле порядка равно 7 битам, максимальное значение q^p, на которое умножается мантисса, равно 2^{128} (при $q = 2$) или 16^{128} (при $q = 16$), а диапазоны представления чисел соответственно составят $10^{-19} < |X| < 10^{+19}$ и $10^{-76} < |X| < 10^{+76}$. Известны также случаи использования базы 8, например, в ВМ B-5500 фирмы Burroughs.

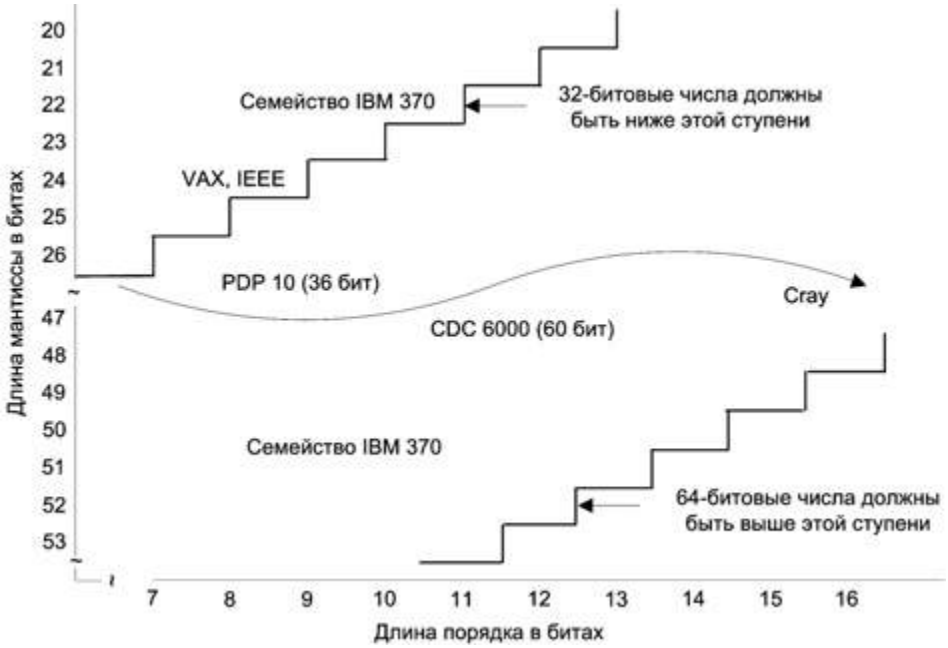


Рис. 2.19. Типовые разрядности полей порядка и мантиссы

В большинстве вычислительных машин для упрощения операций над порядками последние приводят к целым положительным числам, применяя так называемый *смещенный порядок*. Для этого к истинному порядку добавляется целое положительное число — смещение (рис. 2.20). Например, в системе со смещением 128, порядок –3 представляется как 125 (–3 + 128). Обычно смещение выбирается равным половине представимого диапазона порядков. Отметим, что смещенный порядок занимает все биты поля порядка, в том числе и тот, который ранее предназначался для записи знака порядка.

Рис. 2.20. Формат числа с ПЗ со смещенным порядком

Мантисса в числах с ПЗ обычно представляется в *нормализованной форме*. Это означает, что на мантиссу налагаются такие условия, чтобы она по модулю была меньше единицы ($|q| < 1$), а первая цифра после запятой отличалась от нуля. Полученная таким образом мантисса называется *нормализованной*. Для применяемых в ВМ систем счисления можно записать:

- двоичная: $X = m \times 2^p$, $(1 > |m| \geqslant ½)$;
- восьмеричная: $X = m \times 8^p$, $(1 > |m| \geqslant ⅛)$;
- шестнадцатеричная: $X = m \times 16^p$, $(1 > |m| \geqslant 1/16)$.

Если первые *i* цифр мантиссы равны нулю, для нормализации ее нужно сдвинуть относительно запятой на *i* разрядов влево с одновременным уменьшением порядка на *i* единиц. В результате такой операции число не изменяется:

База	**До нормализации**		**После нормализации**	
	Порядок	Мантисса	Порядок	Мантисса
2	100	0,000110	001	0,110000
16	8	0,001010	6	0,101000

В примере для шестнадцатеричной системы после нормализации старшая цифра в двоичном представлении содержит впереди три нуля (0001). Это несколько уменьшает точность представления чисел по сравнению с двоичной системой при одинаковом числе двоичных разрядов, отведенных под мантиссу.

Если для записи числа с ПЗ используется база 2 ($q = 2$), то часто применяют еще один способ повышения точности представления мантиссы, называемый *приемом скрытой единицы*. Суть его в том, что в нормализованной мантиссе старшая цифра всегда равна единице (для представления нуля используется специальная кодовая комбинация), следовательно, эта цифра может не записываться, а подразумеваться. Запись мантиссы начинают с ее второй цифры, и это позволяет задействовать дополнительный значащий бит для более точного представления числа. Следует отметить, что значение порядка в данном случае не меняется. Скрытая единица перед выполнением арифметических операций восстанавливается, а при записи результата — удаляется. Таким образом, нормализованная мантисса 0,101000(1) при использовании способа «скрытой единицы» будет иметь вид 0,010001 (в скобках указана цифра, не поместившаяся в поле мантиссы при стандартной записи).

Для более существенного увеличения точности вычислений под число отводят несколько машинных слов, например два. Дополнительные биты, как правило, служат для увеличения разрядности мантиссы, однако в ряде случаев часть из них может отводиться и для расширения поля порядка. В процессе вычислений может получаться ненормализованное число. В таком случае ВМ, если это предписано командой, автоматически нормализует его.

Рассмотренные принципы представления чисел с ПЗ поясним на примере [143]. На рис. 2.21 представлен типичный 32-битовый формат числа с ПЗ. Старший (левый) бит содержит знак числа. Значение смещенного порядка хранится в разрядах с 30-го по 23-й и может находиться в диапазоне от 0 до 255.

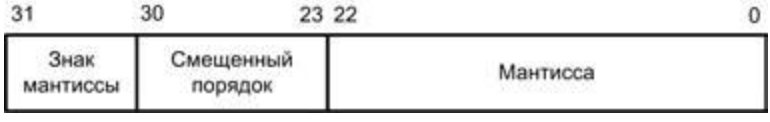


Рис. 2.21. Типичный 32-битовый формат числа с плавающей запятой

Для получения фактического значения порядка из содержимого этого поля нужно вычесть фиксированное значение, равное 128. С таким смещением фактические

значения порядка могут лежать в диапазоне от –128 до +127. В примере предполагается, что основание системы счисления равно 2. Третье поле слова содержит нормализованную мантиссу со скрытым разрядом (единицей). Благодаря такому приему 23-разрядное поле позволяет хранить 24-разрядную мантиссу в диапазоне от 0,5 до 1,0.

На рис. 2.22 приведены диапазоны чисел, которые могут быть записаны с помощью 32-разрядного слова.

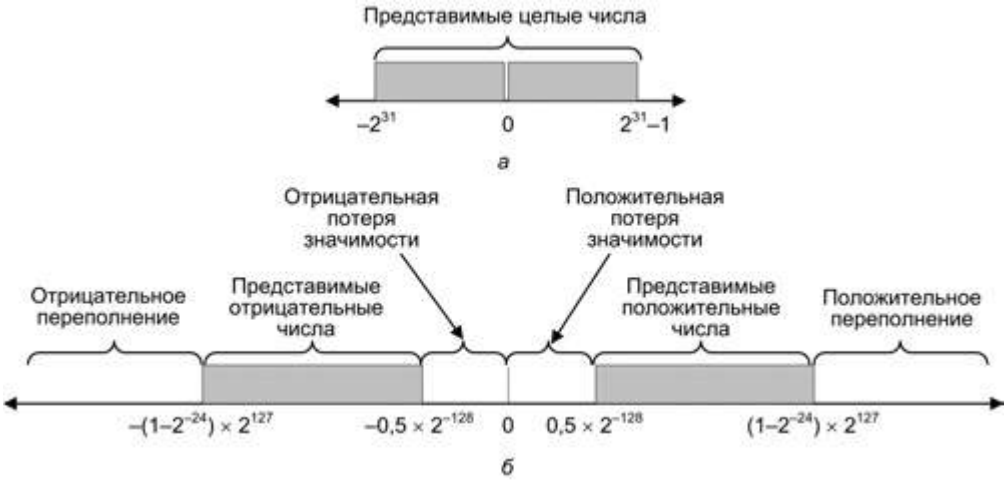


Рис. 2.22. Числа, представимые в 32-битовых форматах: *а* — целые числа с фиксированной запятой; *б* — числа с плавающей запятой

В варианте с ФЗ для целых чисел в дополнительном коде могут быть представлены все целые числа от -2^{31} до $2^{31} - 1$, то есть всего 2^{32} различных чисел (рис. 2.22, *а*). Для случая ПЗ возможны следующие диапазоны чисел (рис. 2.22, *б*):

- отрицательные числа между $-(1 - 2^{-24}) \times 2^{127}$ и $-0,5 \times 2^{-128}$;
- положительные числа между $0,5 \times 2^{-128}$ и $(1 - 2^{-24}) \times 2^{127}$.

В эту область не включены участки:

- отрицательные числа, меньшие чем $-(1 - 2^{-24}) \times 2^{127}$ — *отрицательное переполнение*;
- отрицательные числа, большие чем $-0,5 \times 2^{-128}$ — *отрицательная потеря значимости*;
- положительные числа, меньшие чем $0,5 \times 2^{-128}$ — *положительная потеря значимости*;
- положительные числа, большие чем $(1 - 2^{-24}) \times 2^{127}$ — *положительное переполнение*.

Показанная запись числа с ПЗ не учитывает нулевого значения. Для этой цели используется специальная кодовая комбинация. Переполнения возникают, когда в результате арифметической операции получается значение большее, чем можно

представить порядком 127 ($2^{120} \times 2^{100} = 2^{230}$). Потеря значимости — это когда результат представляет собой слишком маленькое дробное значение ($2^{-120} \times 2^{-100} = 2^{-230}$). Потеря значимости является менее серьезной проблемой, поскольку такой результат обычно рассматривают как нулевой.

Следует также отметить, что числа в формате ПЗ, в отличие от чисел в форме с ФЗ, размещены на числовой оси неравномерно. Возможные значения в начале числовой оси расположены плотнее, а по мере движения вправо — все реже (рис. 2.23). Это означает, что многие вычисления приводят к результату, который не является точным, то есть представляет собой округление до ближайшего значения, представимого в данной форме записи.

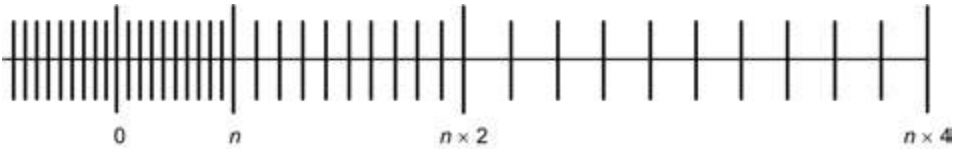


Рис. 2.23. Плотность чисел с плавающей запятой на числовой оси

Для формата, изображенного на рис 2.21, имеет место противоречие между диапазоном и точностью. Если увеличить число битов, отведенных под порядок, расширяется диапазон представимых чисел. Однако, поскольку может быть представлено только фиксированное число различных значений, уменьшается плотность и тем самым точность. Единственный путь увеличения как диапазона, так и точности — увеличение количества разрядов, поэтому в большинстве ВМ предлагается использовать числа в одинарном и двойном форматах. Например, число одинарного формата может занимать 32 бита, а двойного — 64 бита.

Числа с плавающей запятой в разных ВМ имеют несколько различных форматов. В табл. 2.7 приводятся основные параметры для нескольких систем представления чисел в форме с ПЗ. В настоящее время для всех ВМ рекомендован стандарт, разработанный общепризнанным международным центром стандартизации IEEE (Institute of Electrical and Electronics Engineers).

Таблица 2.7. Варианты форматов чисел с плавающей запятой[1]

Параметр	**IBM 390**	**VAX**	**IEEE 754**
Длина слова, бит	О: 32; Д: 64	О: 32; Д: 64	О: 32; Д: 64
Порядок, бит	7	8	О: 8; Д: 11
Мантисса, *m*	О: 6 цифр Д: 14 цифр	О: (1) + 23 бита Д: (1) + 55 битов	О: (1) + 23 бита Д: (1) + 52 бита
Смещение порядка	64	128	К: 127, Д: 1023
Основание системы счисления	16	2	2
Скрытая 1	Нет	Да	Да

продолжение ⇗

[1] О — одинарный формат; Д — двойной формат.

Таблица 2.7 (*продолжение*)

Параметр	IBM 390	VAX	IEEE 754
Запятая	Слева от мантиссы	Слева от скрытой 1	Справа старшего бита мантиссы
Диапазон мантиссы	$(1/16) \le m < 1$	$(1/2) \le m < 1$	$1 \le m < 2$
Представление мантиссы	Величина со знаком	Величина со знаком	Величина со знаком
Максимальное положительное число	$16^{63} \cong 10^{76}$	$2^{126} \cong 10^{38}$	$2^{1024} \cong 10^{308}$ (Д)
Точность	О: $16^{-6} \cong 10^{-7}$ Д: $16^{-14} \cong 10^{-17}$	О: $2^{-24} \cong 10^{-7}$ Д: $2^{-564} \cong 10^{-17}$	О: $2^{-23} \cong 10^{-7}$ Д: $2^{-524} \cong 10^{-16}$

Стандарт IEEE 754

Рекомендуемый для всех ВМ формат представления чисел с плавающей запятой определен стандартом IEEE 754. Этот стандарт был разработан с целью облегчить перенос программ с одного процессора на другие и нашел широкое применение практически во всех процессорах и арифметических сопроцессорах.

Стандарт определяет 32-битовый (одинарный) и 64-битовый (двойной) форматы (рис. 2.24) с 8- и 11-разрядным порядком соответственно. Основанием системы счисления является 2. В дополнение, стандарт предусматривает два расширенных формата, одинарный и двойной, фактический вид которых зависит от конкретной реализации. Расширенные форматы предусматривают дополнительные биты для порядка (увеличенный диапазон) и мантиссы (повышенная точность). Таблица 2.8 содержит описание основных характеристик всех четырех форматов.

Рис. 2.24. Основные форматы IEEE 754: *а* — одинарный; *б* — двойной

Не все кодовые комбинации в форматах IEEE интерпретируются обычным путем — некоторые комбинации используются для представления специальных значений. Предельные значения порядка, содержащие все нули (0) и все единицы (255 — в одинарном формате и 2047 — в двойном формате), определяют специальные значения.

Описываются следующие классы чисел.

- Порядки в диапазоне от 1 до 254 для одинарного формата и от 1 до 2036 — для двойного формата, используются для представления ненулевых нормализо-

ванных чисел. Порядки смещены так, что их диапазон составляет от –126 до +127 для одинарного формата и от –1022 до +1023 — для двойного формата. Нормализованное число требует, чтобы слева от двоичной запятой был единичный бит. Этот бит подразумевается, благодаря чему обеспечивается эффективная ширина мантиссы, равная 24 битам для одинарного и 53 битам — для двойного форматов.

- Нулевой порядок совместно с нулевой мантиссой представляют положительный или отрицательный 0, в зависимости от состояния бита знака мантиссы.
- Порядок, содержащий единицы во всех разрядах, плюс нулевая мантисса соответствуют бесконечности (положительной или отрицательной, в зависимости от состояния бита знака), что позволяет пользователю самому решить, считать ли это ошибкой или продолжать вычисления со значением, равным бесконечности.
- Нулевой порядок в сочетании с ненулевой мантиссой обозначают ненормализованное число. В этом случае бит слева от двоичной точки равен 0 и фактический порядок равен –126 или –1022. Число является положительным или отрицательным в зависимости от значения знакового бита.
- Кодовая комбинация, в которой порядок содержит все единицы, а мантисса не равна 0, используется как признак «не числа» (NAN — Not a Number) и служит для предупреждения о различных исключительных ситуациях, например о делении 0/0.

Таблица 2.8. Параметры форматов стандарта IEEE 754

Параметр	**Формат**			
	одинарный	**одинарный расширенный**	**двойной**	**двойной расширенный**
Разрядность слова, бит	32	≥ 43	64	≥ 79
Поле порядка, бит	8	≥ 11	11	≥ 15
Смещение порядка	127	Не оговорено	1023	Не оговорено
Поле мантиссы, бит	23	≥ 31	52	≥ 63
Максимальное значение порядка	127	≥ 1023	1023	≥ 16383
Минимальное значение порядка	–126	≤ –1022	–1022	≤ –16382
Диапазон чисел	10^{-38}, 10^{+38}	Не оговорен	10^{-308}, 10^{-308}	Не оговорен

Упакованные числа с плавающей запятой

В рамках уже упоминавшейся технологии SSE4 имеются команды, служащие для увеличения производительности систем при обработке мультимедийной информации, описываемой числами с ПЗ. Каждая такая команда работает с четырьмя операндами с плавающей запятой одинарной точности или двумя операндами двойной точности. Операнды упаковываются в 128-разрядные группы, как это показано на рис. 2.25.

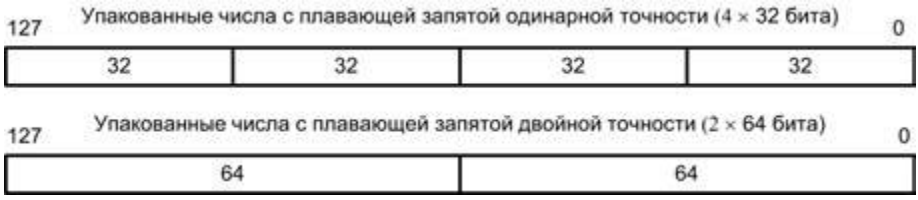


Рис. 2.25. Формат упакованных чисел с плавающей запятой в технологии SSE4

Разрядность основных форматов числовых данных

Данные, представляющие в ВМ числовую информацию, могут иметь *фиксированную* или *переменную* длину. Операционные устройства вычислительных машин (целочисленные арифметико-логические устройства, блоки обработки чисел с плавающей запятой, устройства десятичной арифметики и т. п.), как правило, рассчитаны на обработку кодов фиксированной длины. Общепринятые величины разрядности кодов чисел: бит, полубайт, байт, слово, двойное слово, счетверенное слово, двойное счетверенное слово.

Наименьшей единицей данных в ВМ служит бит (BIT, BInary digiT — двоичная цифра). В большинстве случаев эта единица информации слишком мала. Однобитовые операционные устройства использовались в ВМ с последовательной обработкой информации, а в современных машинах с параллельной обработкой разрядов они практически не применяются. Побитовую работу с данными скорее можно встретить в многопроцессорных вычислительных системах, построенных из одноразрядных процессоров.

Следующая по величине единица состоит из четырех битов и называется полубайтом или тетрадой, или реже «ниблом» (nibble). Она также редко имеет самостоятельное значение и заслуживает упоминания как единица представления отдельных десятичных цифр при их двоично-десятичной записи.

Реально наименьшей обрабатываемой единицей считается байт (BYTE, BinarY TErm — двоичный элемент), состоящий из восьми битов. На практике эта единица информации также оказывается недостаточной, и значительно чаще применяются числа, представленные двумя (слово — 16 битов), четырьмя (двойное слово — 32 бита), восемью (счетверенное слово — 64 бита) или шестнадцатью (двойное счетверенное слово — 128 битов) байтами.

Блоки операций с плавающей запятой обычно согласованы со стандартом IEEE 754 и рассчитаны на обработку чисел в формате двойной длины (64 бита). В большинстве ВМ реальная разрядность таких блоков даже больше (80 битов). Таким образом, наилучшим вариантом при проведении вычислений с плавающей запятой можно считать формат двойного слова. При выборе формата меньшей длины (32 разряда) вычисления все равно ведутся с большей точностью, после чего результат округляется. Таким образом, использование короткого формата чисел с плавающей запятой, как и в случае целых чисел с фиксированной запятой, помимо экономии памяти никаких иных преимуществ также не дает.

В приложениях, оперирующих десятичными числами, где количество цифр в числе может варьироваться в широком диапазоне, что характерно для задач из области экономики, более удобными оказываются форматы переменной длины. В этом случае числа не переводятся в двоичную систему, а записываются в виде последовательности двоично-кодированных десятичных цифр. Длина подобной цепочки может быть произвольной, а для указания ее границы обычно используют символ-ограничитель, код которого не совпадает с кодами цифр. Длина цифровой последовательности может быть задана явно в виде количества цифр числа и храниться в первом байте записи числа, однако этот прием более характерен для указания длины строки символов.

Размещение числовых данных в памяти

В современных ВМ разрядность ячейки памяти, как правило, равна одному байту (8 битов), а реальная длина кодов чисел составляет 2, 4, 8 или 16 байтов. При хранении таких чисел в памяти их байты размещают в нескольких ячейках со смежными адресами, при этом для доступа к числу указывается только наименьший из адресов. При разработке архитектуры системы команд необходимо определить порядок размещения байтов в памяти, то есть какому из байтов (старшему или младшему) будет соответствовать этот наименьший адрес[1]. На рис. 2.26 показаны оба варианта размещения 32-разрядного числа в четырех смежных ячейках памяти, начиная с адреса *A*.

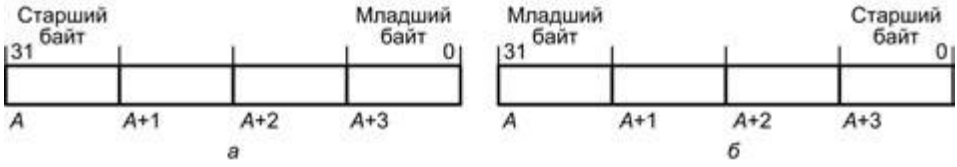


Рис. 2.26. Размещение в памяти 32-разрядного числа: *а* — начиная со старшего байта; *б* — начиная с младшего байта

В вычислительном плане оба способа записи равноценны. Так, фирмы DEC и Intel отдают предпочтения размещению в первой ячейке младшего байта, а IBM и Motorola ориентируются на противоположный вариант. Выбор обычно связан с некими иными соображениями разработчиков ВМ. В настоящее время в большинстве машин предусматривается использование обоих вариантов, причем выбор может быть произведен программным путем за счет соответствующей установки регистра конфигурации.

Помимо порядка размещения байтов, существенным бывает и выбор адреса, с которого может начинаться запись числа. Связано это с физической реализацией полупроводниковых запоминающих устройств, где обычно предусматривается возможность считывания (записи) четырех байтов подряд. Причем данная операция выполняется быстрее, если адрес первого байта A отвечает условию $A \bmod S = 0$

[1] В англоязычной литературе систему записи числа начиная со старшего байта обозначают термином «big endian», а с младшего байта — термином «little endian». Оба названия происходят от названия племен («тупоконечники» и «остроконечники»), упоминаемых в книге Джонатана Свифта «Путешествия Гулливера». Там описывается религиозная война между этими племенами, по причине разногласий в вопросе, с какого конца следует разбивать яйцо — тупого или острого.

(S = 2, 4, 8, 16). Числа, размещенные в памяти в соответствии с этим правилом, называются *выравненными* (рис. 2.27).

Рис. 2.27. Размещение чисел в памяти с выравниванием

На рис. 2.28 показаны варианты размещения 32-разрядного двойного слова без выравнивания. Такое размещение приводит к увеличению времени доступа к памяти.

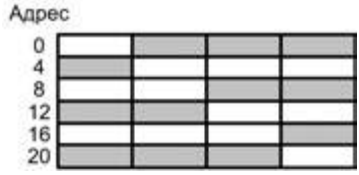


Рис. 2.28. Размещение 32-разрядного слова без соблюдения правила выравнивания

Большинство компиляторов генерируют код, в котором предусмотрено выравнивание чисел в памяти.

Символьная информация

В общем объеме вычислительных действий все большая доля приходится на обработку символьной информации, содержащей буквы, цифры, знаки препинания, математические и другие символы. Каждому символу ставится в соответствие определенная двоичная комбинация. Совокупность возможных символов и назначенных им двоичных кодов образует *таблицу кодировки*. В настоящее время применяются множество различных таблиц кодировки. Объединяет их весовой принцип, при котором веса кодов цифр возрастают по мере увеличения цифры, а веса символов увеличиваются в алфавитном порядке. Так вес буквы «Б» на единицу больше веса буквы «А». Это способствует упрощению обработки в ВМ.

До недавнего времени наиболее распространенными были кодовые таблицы, в которых символы кодируются с помощью восьмиразрядных двоичных комбинаций (байтов), позволяющих представить 256 различных символов:

- расширенный двоично-кодированный код EBCDIC (Extended Binary Coded Decimal Interchange Code);
- американский стандартный код для обмена информацией ASCII (American Standard Code for Information Interchange).

Код EBCDIC используется в качестве внутреннего кода в универсальных ВМ фирмы IBM. Он же известен под названием ДКОИ (двоичный код для обработки информации).

Стандартный код ASCII — 7-разрядный, восьмая позиция отводится для записи бита четности. Это обеспечивает представление 128 символов, включая все латинские буквы, цифры, знаки основных математических операций и знаки пунктуации. Позже появилась европейская модификация ASCII, называемая Latin 1 (стандарт ISO 8859-1). В ней «полезно» используются все 8 разрядов. Дополнительные комбинации (коды 128–255) в новом варианте отводятся для представления специфических букв алфавитов западноевропейских языков, символов псевдографики, некоторых букв греческого алфавита, а также ряда математических и финансовых символов. Именно эта кодовая таблица считается мировым стандартом де-факто, который применяется с различными модификациями во всех странах. В зависимости от использования кодов 128–255 различают несколько вариантов стандарта ISO 8859 (табл. 2.9).

Таблица 2.9. Варианты стандарта ISO 8859

Стандарт	Характеристика
ISO 8859-1	Западноевропейские языки
ISO 8859-2	Языки стран центральной и восточной Европы
ISO 8859-3	Языки стран южной Европы, мальтийский и эсперанто
ISO 8859-4	Языки стран северной Европы
ISO 8859-5	Языки славянских стран с символами кириллицы
ISO 8859-6	Арабский язык
ISO 8859-7	Современный греческий язык
ISO 8859-8	Языки иврит и идиш
ISO 8859-9	Турецкий язык
ISO 8859-10	Языки стран северной Европы (лапландский, исландский)
ISO 8859-11	Тайский язык
ISO 8859-13	Языки балтийских стран
ISO 8859-14	Кельтский язык
ISO 8859-15	Комбинированная таблица для европейских языков
ISO 8859-16	Содержит специфические символы ряда языков: албанского, хорватского, английского, финского, французского, немецкого, венгерского, ирландского, итальянского, польского, румынского и словенского

В распространенной в свое время операционной системе MS-DOS стандарт ISO 8859 реализован в форме *кодовых страниц* OEM (Original Equipment Manufacturer). Каждая OEM-страница имеет свой идентификатор (табл. 2.10).

Таблица 2.10. Наиболее распространенные кодовые страницы OEM

Идентификатор кодовой страницы	Страны
CP437	США, страны западной Европы и Латинской Америки
CP708	Арабские страны
CP737	Греция

продолжение ⇩

Таблица 2.10 (*продолжение*)

Идентификатор кодовой страницы	Страны
CP775	Латвия, Литва, Эстония
CP852	Страны восточной Европы
CP853	Турция
CP855	Страны с кириллической письменностью
CP860	Португалия
CP862	Израиль
CP865	Дания, Норвегия
CP866	Россия
CP932	Япония
CP936	Китай

Хотя код ASCII достаточно удобен, он все же слишком тесен и не вмещает множества необходимых символов. По этой причине в 1993 году консорциумом компаний Apple Computer, Microsoft, Hewlett–Packard, DEC и IBM был разработан 16-битовый стандарт ISO 10646, определяющий универсальный набор символов (UCS, Universal Character Set). Новый код, известный под названием Unicode, позволяет задать до 65 536 символов, то есть дает возможность одновременно представить символы всех основных «живых» и «мертвых» языков. Для букв русского языка выделены коды 1040–1093.

Все символы в Unicode логически разделяют на 17 плоскостей по 65536 (2^{16}) кодов в каждой[1].

- Плоскость 0 (0000–FFFF): BMP, Basic Multilingual Plane — основная многоязычная плоскость. Эта плоскость охватывает коды большинства основных символов используемых в настоящее время языков. Каждый символ представляется 16-разрядным кодом.
- Плоскость 1 (10000–1FFFF): SMP, Supplementary Multilingual Plane — дополнительная многоязычная плоскость.
- Плоскость 2 (20000–2FFFF): SIP, Supplementary Ideographic Plane — дополнительная идеографическая плоскость.
- Плоскости с 3 по 13 (30000–DFFF) пока не используются.
- Плоскость 14 (E0000–EFFFF): SSP, Supplementary Special-purpose Plane — специализированная дополнительная плоскость.
- Плоскость 15 (F0000–FFFFF): PUA, Private Use Area — область для частного использования.
- Плоскость 16 (100000–10FFFF): PUA, Private Use Area — область для частного использования.

[1] Предусмотрено увеличение длины кода до 21 бита.

В настоящее используется порядка 10% потенциального пространства кодов.

В «естественном» варианте кодировки Unicode, известном как UCS-2, каждый символ описывается двумя последовательными байтами m и n, так что номеру символа соответствует численное значение $256 \times m + n$. Таким образом, кодовый номер представлен 16-разрядным двоичным числом.

Наряду с UCS-2 в рамках Unicode существуют еще несколько вариантов кодировки Unicode (UTF, Unicode Transformation Formats), основные из которых UTF-32, UTF-16, UTF-8 и UTF-7.

В кодировке UTF-32 каждая кодовая позиция представлена 32-разрядным двоичным числом. Это очень простая и очевидная система кодирования, хотя и неэффективная в плане разрядности кода. Кодировка UTF-32 используется редко, главным образом, в операциях обработки строк для внутреннего представления данных.

UTF-16 каждую кодовую позицию для символов из плоскости BMP представляет двумя байтами. Кодовые позиции из других плоскостей представлены так называемой суррогатной парой. Представление основных символов одинаковым числом байтов очень удобно в том плане, что можно прямо адресоваться к любому символу строки.

В кодировке UTF-8 коды символов меньшие, чем 128, представляются одним байтом. Все остальные коды формируются по более сложным правилам. В зависимости от символа, его код может занимать от двух до шести байтов, причем старший бит каждого байта всегда имеет единичное значение. Иными словами, значение байта лежит в диапазоне от 128 до 255. Ноль в старшем бите байта означает, что код занимает один байт и совпадает по кодировке с ASCII. Схема формирования кодов UTF-8 показана в табл. 2.11.

Таблица 2.11. Структура кодов UTF-8

Число байтов	Двоичное представление	Число свободных битов
1	0xxxxxxx	7
2	110xxxxx 10xxxxxx	11 (5 + 6)
3	1110xxxx 10xxxxxx 10xxxxxx	16 (4 + 6 × 2)
4	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx	21 (3 + 6 × 3)
5	111110xx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx	26 (2 + 6 × 4)
6	1111110x 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx	31 (1 + 6 × 5)

В UTF-7 код символа также может занимать один или более байтов, однако в каждом из байтов значение не превышает 127 (старший бит байта содержит ноль). Многие символы кодируются одним байтом, и их кодировка совпадает с ASCII, однако некоторые коды зарезервированы для использования в качестве преамбулы, характеризующей последующие байты многобайтового кода.

Стандарт Unicode обратно совместим с кодировкой ASCII, однако, если в ASCII для представления схожих по виду символов (минус, тире, знак переноса) применялся общий код, в Unicode каждый из этих символов имеет уникальную кодировку.

Впервые Unicode был использован в операционной системе Windows NT. Распределение кодов в Unicode иллюстрирует табл. 2.12.

Таблица 2.12. Блоки символов в стандарте Unicode

Коды	Символы
0–8191	Алфавиты — английский, европейские, фонетический, кириллица, армянский, иврит, арабский, эфиопский, бенгали, деванагари, гур, гуджарати, ория, телугу, тамильский, каннада, малайский, сингальский, грузинский, тибетский, тайский, лаосский, кхмерский, монгольский
8192–12287	Знаки пунктуации, математические операторы, технические символы, орнаменты и т. п.
12288–16383	Фонетические символы китайского, корейского и японского языков
16384–59391	Китайские, корейские, японские идеографы. Единый набор символов каллиграфии хань
59392–65024	Блок для частного использования
65025–65536	Блок обеспечения совместимости с программным обеспечением

Логические данные

Элементом логических данных является логическая (булева) переменная, которая может принимать лишь два значения: «истина» или «ложь». Кодирование логического значения принято осуществлять битом информации: единицей кодируют истинное значение, нулем — ложное. Как правило, в ВМ оперируют наборами логических переменных длиной в машинное слово. Обрабатываются такие слова с помощью команд логических операций (И, ИЛИ, НЕ и т. д.), при этом все биты обрабатываются одинаково, но независимо друг от друга, то есть никаких переносов между разрядами не возникает.

Строки

Строки — это непрерывная последовательность битов, байтов, слов или двойных слов. *Битовая строка* может начинаться в любой позиции байта и содержать до 2^{32} битов. *Байтовая строка* может состоять из байтов, слов или двойных слов. Длина такой строки варьируется от нуля до $2^{32} - 1$ байтов (4 Гбайт). Приведенные цифры характерны для 32-разрядных ВМ.

Если байты байтовой строки представляют собой коды символов, то говорят о *текстовой строке*. Поскольку длина текстовой строки может меняться в очень широких пределах, то для указания конца строки в последний байт заносится код-ограничитель — обычно это нули во всех разрядах байта. Иногда вместо ограничителя длину строки указывают числом, расположенным в первом или двух первых байтах строки.

Прочие виды информации

Представляемая в ВМ информация может быть статической или динамической [25]. Так, числовая, символьная и логическая информация является статической — ее

значение не связано со временем. Напротив, аудиоинформация имеет динамический характер — существует только в режиме реального времени и не может быть остановлена для более подробного изучения. Если изменить масштаб времени, аудиоинформация искажается, что используется, например, для создания звуковых эффектов.

Видеоинформация

Видеоинформация бывает как статической, так и динамической. Статическая видеоинформация включает в себя текст, рисунки, графики, чертежи, таблицы и др. Рисунки делятся также на плоские — двумерные и объемные — трехмерные.

Динамическая видеоинформация — это видео-, мульт- и слайд-фильмы. В их основе лежит последовательное экспонирование на экране в реальном масштабе времени отдельных кадров в соответствии со сценарием. Динамическая информация используется либо для передачи движущихся изображений (анимация), либо для последовательной демонстрации отдельных кадров (слайд-фильмы).

Для демонстрации анимационных и слайд-фильмов опираются на различные принципы. Анимационные фильмы демонстрируются так, чтобы зрительный аппарат человека не мог зафиксировать отдельных кадров (для получения качественной анимации кадры должны сменяться порядка 70 раз/с). При демонстрации слайд-фильмов каждый кадр экспонируется на экране столько времени, сколько необходимо для восприятия его человеком (обычно от 30 с до 1 мин). Слайд-фильмы можно отнести к статической видеоинформации.

В вычислительной технике существует два способа представления графических изображений: *матричный* (*растровый*) и *векторный*. Матричные (bitmap) форматы хорошо подходят для изображений со сложными гаммами цветов, оттенков и форм, таких как фотографии, рисунки, отсканированные данные. Векторные форматы более приспособлены для чертежей и изображений с простыми формами, тенями и окраской.

В матричных форматах изображение представляется прямоугольной матрицей точек — *пикселов* (picture element), положение которых в матрице соответствует координатам точек на экране. Помимо координат, каждый пиксел характеризуется своим цветом, цветом фона или градацией яркости. Количество битов, выделяемых для указания цвета пиксела, изменяется в зависимости от формата. В высококачественных изображениях цвет пиксела описывают 32 битами, что дает около 4096 млн цветов. Основной недостаток матричной (растровой) графики заключается в большой емкости памяти, требуемой для хранения изображения, из-за чего для описания изображений прибегают к различным методам сжатия данных. Сжатие изображения обычно включает два этапа. На первом этапе изображение делится на блоки пикселей, в каждом из которых устраняется избыточность. На втором этапе производится кодирование длинных последовательностей нулей и единиц и представление их кодами переменной длины. В случае динамической видеоинформации возможен и третий этап сжатия за счет сравнения каждого изображения с предыдущим и сохранением лишь изменившейся его части. В настоящее время существует множество форматов графических файлов, различающихся алгоритмами сжатия

и способами представления матричных изображений, а также сферой применения. Некоторые из распространенных форматов матричных графических файлов перечислены в табл. 2.13.

Таблица 2.13. Матричные графические форматы

Обозначение	**Полное название**
BMP	Windows и OS\2 Bitmap
GIF	Graphics Interchange Format
PCX	PC Paintbrush File Format
JPEG	Joint Photographic Experts Group
TIFF	Tagged Image File Format
PNG	Portable Network Graphics

Векторное представление, в отличие от матричной графики, задает изображение не пикселами, а графическими примитивами, которые могут быть описаны математически. В качестве примитивов могут выступать:

- линии и ломаные линии;
- многоугольники;
- окружности и эллипсы;
- сплайны;
- безигоны.

Прежде всего следует пояснить некоторые пункты в вышеприведенном списке, в частности понятия сплайна и безигона.

Сплайн — это гладкая кривая, которая проходит через две или более опорных точки, управляющие формой сплайна. В векторной графике наиболее распространены сплайны на основе кривых Безье (Bezie). Кривая Безье описывается парой касательных в точках ее концов. Касательные задаются как отрезки прямой путем указания координат концов этих отрезков, то есть четырьмя координатами P_0, P_1, P_2 и P_3. Эти отрезки можно рассматривать как «рычаги», с помощью которых кривую изгибают нужным образом. Форма кривой зависит не только от наклона касательной, но и от длины отрезка, представляющего касательную (рис. 2.29).

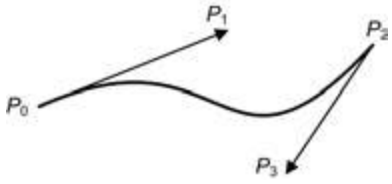


Рис. 2.29. Пример кривой Безье

Термин безигон образованный сочетанием слов Безье и полигон (многоугольник), обозначает фигуру, напоминающую многоугольник, но в котором вершины соединены не прямой линией, а кривыми Безье.

Так как все примитивы в векторной графике определены соответствующими математическими выражениями, для их описания в рамках изображения достаточно указать лишь несколько параметров, а соответствующее графическое представление получить путем «вычислений». Так, для описания окружности достаточно указать ее радиус и координаты центра, а также цвет и толщину контура и цвет заполнения (если не предполагается сделать контур или окружность прозрачными). Сравнительно небольшое число параметров, характеризующих графические примитивы, позволяет существенно сократить объем файла с изображением. Еще одно достоинство векторной графики — при масштабировании объектов качество изображения не ухудшается.

Недостатком векторных изображений является их некоторая искусственность, заключающаяся в том, что любое изображение необходимо разбить на конечное множество составляющих его примитивов. Кроме того, с увеличением числа графических примитивов возрастает и объем соответствующего файла. Как и для матричной графики, существует несколько форматов графических векторных файлов. Некоторые из них приведены в табл. 2.14.

Таблица 2.14. Векторные графические форматы

Обозначение	**Полное название**
DXF	Drawing Interchange Format
CDR	Corel Drawing
AI	Adobe Illustrator
PS	PostScript
SVG	Scalable Vector Graphics
VSD	Microsoft Visio format

Матричная и векторная графика существуют не обособленно друг от друга. Так, векторные рисунки могут включать в себя и матричные изображения. Кроме того, векторные и матричные изображения могут быть преобразованы друг в друга. Графические форматы, позволяющие сочетать матричное и векторное описание изображения, называются *метафайлами*. Метафайлы обеспечивают достаточную компактность файлов с сохранением высокого качества изображения. Примеры наиболее распространенных форматов метафайлов даны в табл. 2.15.

Таблица 2.15. Форматы метафайлов

Обозначение	**Полное название**
EPS	Encapsulated PostScript
WMF	Windows Metafile
CGM	Computer Graphics Metafile

Рассмотренные формы представления статической видеоинформации используются, в частности, для отдельных кадров, образующих анимационные фильмы. Для хранения анимационных фильмов применяются различные методы сжатия информации, большинство из которых стандартизовано.

Аудиоинформация

Понятие *аудио* связано со звуками, которые способно воспринимать человеческое ухо. Частоты аудиосигналов лежат в диапазоне от 15 Гц до 20 кГц, а сигналы по своей природе являются непрерывными (аналоговыми). Прежде чем быть представленной в ВМ, аудиоинформация должна быть преобразована в цифровую форму (оцифрована). Для этого значения звуковых сигналов (выборки, samples), взятые через малые промежутки времени, с помощью аналого-цифровых преобразователей (АЦП) переводятся в двоичный код. Обратное действие выполняется цифро-аналоговыми преобразователями (ЦАП). Чем чаще производятся выборки, тем выше может быть точность последующего воспроизведения исходного сигнала, но тем большая емкость памяти требуется для хранения оцифрованного звука. Для высококачественного представления аудиоинформации рекомендуется 16-разрядное представление амплитуды сигнала (2^{16} градаций уровня звука) и частота выборки порядка 40 кГц (промежуток времени между последовательными выборками не более 25 мкс).

Цифровой эквивалент аудиосигналов обычно хранится в виде файлов, причем широко используются различные методы сжатия такой информации. Как правило, к методам сжатия аудиоинформации предъявляется требование возможности восстановления непрерывного сигнала без заметного ухудшения его качества. В настоящее время распространен целый ряд форматов хранения аудиоинформации. Некоторые из них перечислены в табл. 2.16.

Таблица 2.16. Форматы аудиофайлов

Обозначение	Полное название
AVI	Audio Video Interleave
WAV	WAVeform Extension
MIDI	Musical Instrument Digital Interface
AIF	Audio Interchange Format
MPEG	Motion Picture Expert Group Audio
RA	Real Audio

Типы команд

Несмотря на различие в системах команд разных ВМ, некоторые основные типы операций могут быть найдены в любой из них. Для описания этих типов примем следующую классификацию:

- команды пересылки данных;
- команды арифметической и логической обработки;
- команды работы со строками;
- команды SIMD;
- команды преобразования;
- команды ввода/вывода;
- команды управления потоком команд.

Команды пересылки данных

Это наиболее распространенный тип машинных команд. В таких командах должна содержаться следующая информация:

- адреса источника и получателя операндов — адреса ячеек памяти, номера регистров процессора или информация о том, что операнды расположены в стеке;
- длина подлежащих пересылке данных (обычно в байтах или словах), заданная явно или косвенно;
- способ адресации каждого из операндов, с помощью которого содержимое адресной части команды может быть пересчитано в физический адрес операнда.

Рассматриваемая группа команд обеспечивает передачу информации между процессором и ОП, внутри процессора и между ячейками памяти. Пересылочные операции внутри процессора задаются командами формата «регистр-регистр». Команды передачи между процессором и памятью относятся к формату «регистр-память», а команды пересылки в памяти — к формату «память-память».

Команды арифметической и логической обработки

В данную группу входят команды, обеспечивающие арифметическую и логическую обработку информации в различных формах ее представления. Для каждой формы представления чисел в АСК обычно предусматривается некий стандартный набор операций.

Помимо вычисления результата, выполнение арифметических и логических операций сопровождается формированием в АЛУ признаков (флагов), характеризующих этот результат. Наиболее часто фиксируются такие признаки, как: **Z** (**Z**ero) — нулевой результат; **N** (**N**egative) — отрицательный результат; **V** (o**V**erflow) — переполнение разрядной сетки; **C** (**C**arry) — наличие переноса.

Операции с целыми числами

К стандартному набору операций над целыми числами, представленными в форме с фиксированной запятой, следует отнести:

- двухместные арифметические операции (операции с двумя операндами): сложение, вычитание, умножение и деление;
- одноместные арифметические операции (операции с одним операндом): вычисление абсолютного значения (модуля) операнда, изменение знака операнда;
- операции сравнения, обеспечивающие сравнение двух целых чисел и выработку признаков, характеризующих соотношение между сопоставляемыми величинами (=, <>, >, <, <=, >=).

Часто этот перечень дополняют такими операциями, как вычисление остатка от целочисленного деления, сложение с учетом переноса, вычитание с учетом займа, увеличение значения операнда на единицу (инкремент), уменьшение значения операнда на единицу (декремент).

Отметим, что выполнение арифметических команд может дополнительно сопровождаться перемещением данных из устройства ввода в АЛУ или из АЛУ на устройство вывода.

Операции с числами в форме с плавающей запятой

Для работы с числами, представленными в форме с плавающей запятой, в АСК большинства машин предусмотрены:

- основные арифметические операции: сложение, вычитание, умножение и деление;
- операции сравнения, обеспечивающие сравнение двух вещественных чисел с выработкой признаков: =, <>, >, <, <=, >=;
- операции преобразования: формы представления (между фиксированной и плавающей запятой), формата представления (с одинарной и двойной точностью).

Логические операции

Стандартная система команд ВМ содержит команды для выполнения различных логических операций над отдельными битами слов или других адресуемых единиц. Такие команды предназначены для обработки символьных и логических данных. Минимальный набор поддерживаемых логических операций — это «НЕ», «И», «ИЛИ» и «сложение по модулю 2».

Операции сдвигов

В дополнение к побитовым логическим операциям, практически во всех АСК предусмотрены команды для реализации операций логического, арифметического и циклического сдвигов (рис. 2.30).

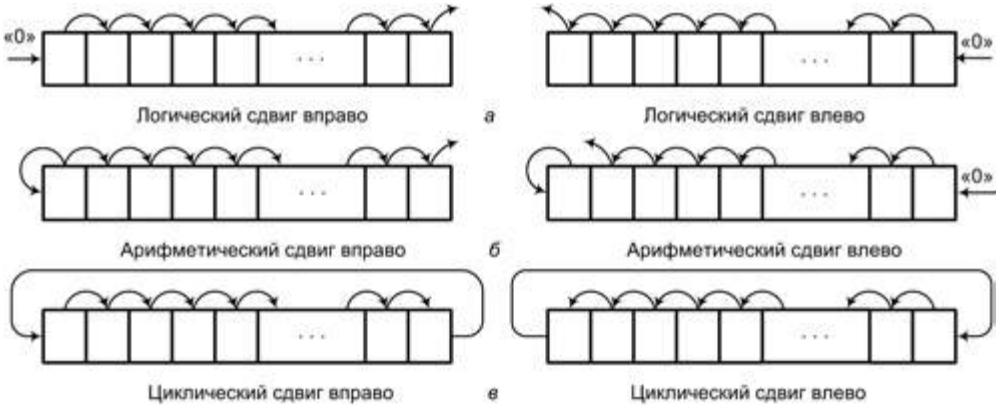


Рис. 2.30. Варианты операций сдвига

При *логическом* сдвиге влево или вправо (рис. 2.30, *а*), сдвигаются все разряды слова. Биты, вышедшие за пределы разрядной сетки, теряются, а освободившиеся позиции заполняются нулями.

При *арифметическом* сдвиге (рис. 2.30, *б*) данные трактуются как целые числа со знаком, причем бит знака не изменяет положения. При сдвиге вправо освободившиеся позиции заполняются значением знакового разряда, а при сдвиге влево — нулями. Арифметические сдвиги позволяют ускорить выполнение некоторых арифметических операций. Так, если числа представлены двоичным дополнительным

кодом, то сдвиги влево и вправо эквивалентны соответственно умножению и делению на 2.

При *циклическом* сдвиге (рис. 2.30, *в*) смещаются все разряды слова, причем значение разряда, выходящего за пределы слова, заносится в позицию, освободившуюся с противоположной стороны, то есть потери информации не происходит. Одно из возможных применений циклических сдвигов — это перемещение интересующего бита в крайнюю левую (знаковую) позицию, где он может быть проанализирован как знак числа.

Операции с десятичными числами

Десятичные числа представляются в ВМ в двоично-кодированной форме. В вычислительных машинах первых поколений для обработки таких чисел предусматривались специальные команды, обеспечивавшие выполнение основных арифметических операций (сложение, вычитание, умножение и деление). В АСК современных машин подобных команд обычно нет, а соответствующие вычисления имитируются с помощью команд целочисленной арифметики с последующей коррекцией полученного результата.

SIMD-команды

Название данного типа команд представляет собой аббревиатуру от Single Instruction Multiple Data — буквально «одна команда — много данных». В отличие от обычных команд, оперирующих двумя числами, SIMD-команды обрабатывают сразу две группы чисел (в принципе, их можно называть групповыми командами). Операнды таких команд обычно представлены в одном из упакованных форматов.

Идея SIMD-обработки была выдвинута в Институте точной механики и вычислительной техники им. С. А. Лебедева в 1978 году в рамках проекта «Эльбрус-1». С 1992 года команды типа SIMD становятся неотъемлемым элементом АСК микропроцессоров фирм Intel и AMD. Поводом послужило широкое распространение мультимедийных приложений. Видео, трехмерная графика и звук в ВМ представляются большими массивами данных, элементы которых чаще всего обрабатываются идентично. Так, при сжатии видео и преобразовании его в формат MPEG один и тот же алгоритм применяется к тысячам битов данных. В трехмерной графике часто встречаются операции, которые можно выполнить за один такт: интерполирование и нормировка векторов, вычисление скалярного произведения векторов, интерполяция компонентов цвета и т. д. Включение SIMD-команд в АСК позволяет существенно ускорить подобные вычисления.

Первой на мультимедийный бум отреагировала фирма Intel, добавив в систему команд своего микропроцессора Pentium MMX 57 SIMD-команд. Название MMX (MultiMedia eXtention — мультимедийное расширение) разработчики обосновывали тем, что при выборе состава новых команд были проанализированы алгоритмы, применяемые в различных мультимедийных приложениях. Команды MMX обеспечивали параллельную обработку упакованных целых чисел. При выполнении арифметических операций каждое из чисел, входящих в группу, рассматривается как самостоятельное, без связи с соседними числами. Учитывая специфику

обрабатываемой информации, команды MMX реализуют так называемую арифметику с насыщением: если в результате сложения образуется число, выходящее за пределы отведенных под него позиций, оно заменяется наибольшим двоичным числом, которое в эти позиции вмещается. На рис. 2.31 показано сложение двух групп четырехразрядных целых чисел, упакованных в 32-разрядные слова.

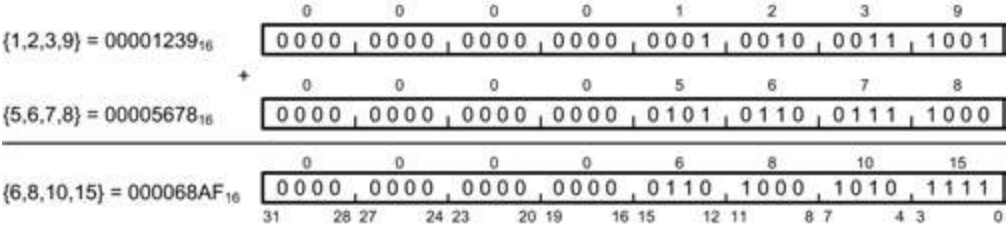


Рис. 2.31. Сложение с насыщением упакованных целых чисел

Следующим шагом стало создание новых наборов SIMD-команд, работающих также с операндами, представленными в виде упакованных чисел с плавающей запятой. Такие команды в соответствующих приложениях повышают производительность процессора примерно вдвое. Первой подобную технологию в середине 1998 года предложила фирма AMD в микропроцессоре K6-2. Это мультимедийное расширение включало в себя 21 SIMD-команду и получило название 3DNow!. Расширение 3DNow! в дополнение к SIMD-обработке целочисленной информации типа MMX позволяло оперировать парой упакованных чисел в формате с плавающей запятой.

В дальнейшем наборы SIMD-команд стали обозначать аббревиатурой SSE — Streaming SIMD Extension (потоковая обработка по принципу «одна команда — много данных»). В последних разработках фирм Intel и AMD технология обозначена как SSE4, при этом у каждой из фирм она имеет свои особенности. Так, в микропроцессорах Penryn фирмы Intel реализован набор из 47 SIMD-команд (SSE4.1), в микропроцессорах Nehalem (Core i7) той же фирмы — дополнительно еще 7 команд (SSE4.2). Микропроцессоры Phenom фирмы AMD в настоящее время поддерживают лишь 4 команды из SSE4, но дополнительно реализуют две новые команды, обозначаемые как SSE4a. В 2007 году фирма AMD анонсировала новый стандарт — SSE5, преподносимый как 128-разрядное расширение SSE. Предлагается набор из 170 команд, который предполагается реализовать на процессорах Bulldozer, производство которых запланировано на 2011 год.

Еще один вариант архитектуры системы команд с SIMD-командами воплощен фирмой IBM в процессорах серии PowerPC. Эта реализация носит название AltiVec.

Команды для работы со строками

Для работы со строками в АСК обычно предусматриваются команды, обеспечивающие перемещение, сравнение и поиск строк. В большинстве машин перечисленные операции просто имитируются за счет других команд.

Команды преобразования

Команды преобразования осуществляют изменение формата представления данных. Примером может служить преобразование из десятичной системы счисления в двоичную или перевод 8-разрядного кода символа из кодировки ASCII в кодировку EBCDIC, и наоборот.

Команды ввода/вывода

Команды этой группы могут быть подразделены на команды управления периферийным устройством (ПУ), проверки его состояния, ввода и вывода.

Команды *управления периферийным устройством* служат для запуска ПУ и указания ему требуемого действия. Например, накопителю на магнитной ленте может быть предписана перемотка ленты или ее продвижение вперед на одну запись. Трактовка подобных инструкций зависит от типа ПУ.

Команды *проверки состояния ввода/вывода* применяются для тестирования различных признаков, характеризующих состояние модуля В/ВЫВ и подключенных к нему ПУ. Благодаря этим командам центральный процессор может выяснить, включено ли питание ПУ, завершена ли предыдущая операция ввода/вывода, возникли ли в процессе ввода/вывода какие-либо ошибки и т. п.

Собственно обмен информацией с ПУ обеспечивают команды ввода и вывода. Команды *ввода* предписывают модулю В/ВЫВ получить элемент данных (байт или слово) от ПУ и поместить его на шину данных, а команды *вывода* — заставляют модуль В/ВЫВ принять элемент данных с шины данных и переслать его на ПУ.

Команды управления системой

Команды, входящие в эту группу, являются привилегированными и могут выполняться только когда центральный процессор ВМ находится в привилегированном состоянии или выполняет программу, находящуюся в привилегированной области памяти (обычно привилегированный режим используется лишь операционной системой). Так, лишь эти команды способны считывать и изменять состояние ряда регистров устройства управления.

Команды управления потоком команд

Концепция фон-неймановской вычислительной машины предполагает, что команды программы, как правило, выполняются в порядке их расположения в памяти. Для получения адреса очередной команды достаточно увеличить содержимое счетчика команд на длину текущей команды. В то же время основные преимущества ВМ заключаются именно в возможности изменения хода вычислений в зависимости от возникающих в процессе счета результатов. С этой целью в АСК вычислительной машины включаются команды, позволяющие нарушить естественный порядок следования и передать управление в иную точку программы. В адресной части таких команд содержится адрес точки перехода (адрес той команды, которая должна быть выполнена следующей). Переход реализуется путем загрузки адреса

точки перехода в счетчик команд (вместо увеличения содержимого этого счетчика на длину команды).

В системе команд ВМ можно выделить три разновидности команд, способных изменить последовательность вычислений:

- безусловные переходы;
- условные переходы (ветвления);
- вызовы процедур и возвраты из процедур.

Статистика программирования показывает, что среди команд рассматриваемой группы доминируют условные переходы.

Несмотря на то что присутствие в программе большого числа *команд безусловного перехода* считается признаком плохого стиля программирования, такие команды обязательно входят в АСК любой ВМ. Для их обозначения в языке ассемблера обычно используется английское слово *jump* (прыжок). Команда безусловного перехода обеспечивает переход по заданному адресу без проверки каких-либо условий.

Условный переход происходит только при соблюдении определенного условия, в противном случае выполняется следующая по порядку команда программы. Большинство производителей ВМ в своих ассемблерах обозначают подобные команды словом *branch* (ветвление).

Условием, на основании которого осуществляется переход, чаще всего выступают признаки результата предшествующей арифметической или логической операции. Каждый из признаков фиксируется в своем разряде регистра признаков процессора. Возможен и иной подход, когда решение о переходе принимается в зависимости от состояния одного из регистров общего назначения, куда предварительно помещается результат операции сравнения. Третий вариант — это объединение операций сравнения и перехода в одной команде.

В системе команд ВМ для каждого признака результата предусматривается своя команда ветвления (иногда — две: переход при наличии признака и переход при его отсутствии). Большая часть условных переходов связана с проверкой взаимного соотношения двух величин или с равенством (неравенством) некоторой величины нулю. Последний вид проверок используется в программах наиболее интенсивно.

Одной из форм команд условного перехода являются *команды пропуска*. В них адрес перехода отсутствует, а при выполнении условия происходит пропуск следующей команды, то есть предполагается, что отсутствующий в команде адрес следующей команды эквивалентен адресу текущей команды, увеличенному на длину пропускаемой команды. Такой прием позволяет сократить длину команд передачи управления.

Для всех языков программирования характерно интенсивное использование механизма процедур. Процедура может быть вызвана в любой точке программы. Для ВМ такой вызов означает, что в этой точке необходимо выполнить процедуру, после чего вернуться в точку, непосредственно следующую за местом вызова.

Процедурный механизм базируется на *командах вызова процедуры*, обеспечивающих переход из текущей точки программы к начальной команде процедуры,

и *командах возврата из процедуры*, для возврата в точку, непосредственно расположенную за командой вызова. Такой режим предполагает наличие средств для сохранения текущего состояния содержимого счетчика команд в момент вызова (запоминание адреса точки возврата) и его восстановления при выходе из процедуры.

Форматы команд

Типовая команда, в общем случае, должна указывать:

- подлежащую выполнению операцию;
- адреса исходных данных (операндов), над которыми выполняется операция;
- адрес, по которому должен быть помещен результат операции.

В соответствии с этим команда состоит из двух частей: операционной и адресной (рис. 2.32).

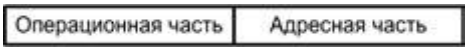


Рис. 2.32. Структура команды

Формат команды определяет ее структуру, то есть количество двоичных разрядов, отводимых под всю команду, а также количество и расположение отдельных полей команды. *Полем* называется совокупность двоичных разрядов, кодирующих составную часть команды. При создании ВМ выбор формата команды влияет на многие характеристики будущей машины. Оценивая возможные форматы, нужно учитывать следующие факторы:

- общее число различных команд;
- общую длину команды;
- тип полей команды (фиксированной или переменной длины) и их длина;
- простоту декодирования;
- адресуемость и способы адресации;
- стоимость оборудования для декодирования и исполнения команд.

Длина команды

Это важнейшая характеристика, влияющая на организацию и емкость памяти, структуру шин, сложность и быстродействие ЦП. С одной стороны, удобно иметь в распоряжении мощный набор команд, то есть как можно больше кодов операций, операндов, способов адресации, и максимальное адресное пространство. Однако все это требует выделения большего количества разрядов под каждое поле команды, что приводит к увеличению ее длины. Вместе с тем, для ускорения выборки из памяти желательно, чтобы команда была как можно короче, а ее длина была равна или кратна ширине шины данных. Для упрощения аппаратуры и повышения быстродействия ВМ длину команды обычно выбирают кратной байту, поскольку в большинстве ВМ основная память организована в виде 8-битовых ячеек. В рамках системы команд одной ВМ могут использоваться разные форматы команд. Обычно это связано с применением различных способов адресации. В таком случае в состав

кода команды вводится поле для задания способа адресации (СА), и обобщенный формат команды приобретает вид, показанный на рис 2.33.

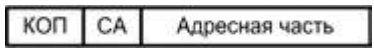


Рис. 2.33. Обобщенный формат команды

Общая длина команды $R_К$ может быть описана следующим соотношением:

$$R_K = \sum_{i=1}^{l} R_{A_i} + R_{КОП} + R_{СА}, \qquad (2.1)$$

где l — количество адресов в команде; R_{A_i} — количество разрядов для записи i-го адреса; $R_{КОП}$ — разрядность поля кода операции; $R_{СА}$ — разрядность поля способа адресации.

В большинстве ВМ одновременно уживаются несколько различных форматов команд.

Разрядность полей команды

Как уже говорилось, в любой команде можно выделить операционную и адресную части. Длины соответствующих полей определяются различными факторами.

Разрядность поля кода операции

Количество двоичных разрядов, отводимых под код операции, выбирается так, чтобы можно было представить любую из операций. Если система команд предполагает $N_{КОП}$ различных операций, то минимальная разрядность поля кода операции $R_{КОП}$ определяется следующим образом:

$$R_{КОП} = \lceil \log_2 N_{КОП} \rceil, \qquad (2.2)$$

где $\lceil\ \rceil$ означает округление в большую сторону до целого числа.

При заданной длине кода команды приходится искать компромисс между разрядностью поля кода операции и адресного поля. Большее количество возможных операций предполагает длинное поле кода операции, что ведет к сокращению адресного поля, то есть к сужению адресного пространства. Для устранения этого противоречия иногда длину поля кода операции варьируют. Изначально под код операции отводится некое фиксированное число разрядов, однако для отдельных команд это поле расширяется за счет нескольких битов, отнимаемых у адресного поля. Так, например, может быть увеличено число различных команд пересылки данных. Необходимо отметить, что «урезание» части адресного поля ведет к сокращению возможностей адресации, и такой прием рекомендуется только в тех командах, где подобное сокращение оправдано.

Разрядность адресной части

В адресной части команды содержится информация о местонахождении исходных данных и месте сохранения результата операции. Обычно местонахождение

каждого из операндов и результата задается в команде путем указания адреса соответствующей ячейки основной памяти или номера регистра процессора. Принципы использования информации из адресной части команды определяет *система адресации*. Система адресации задает *число адресов в команде* команды и принятые *способы адресации*.

Разрядности полей R_{A_i} и R_{CA} рассчитываются по формулам:

$$R_{A_i} = \lceil \log_2 N_i \rceil, \tag{2.3}$$

$$R_{CA} = \lceil \log_2 N_{CA} \rceil, \tag{2.4}$$

где N_i — количество ячеек памяти, к которому можно обратиться с помощью i-го адреса; N_{CA} — количество способов адресации.

Количество адресов в команде

Для определения количества адресов, включаемых в адресную часть, будем использовать термин *адресность*. В «максимальном» варианте необходимо указать три компонента: адрес первого операнда, адрес второго операнда и адрес ячейки, куда заносится результат операции. В принципе, может быть добавлен еще один адрес, указывающий место хранения следующей команды. В итоге имеет место *четырехадресный формат команды* (рис. 2.34). Такой формат поддерживался в ВМ EDVAC, разработанной в 1940-х годах [91].

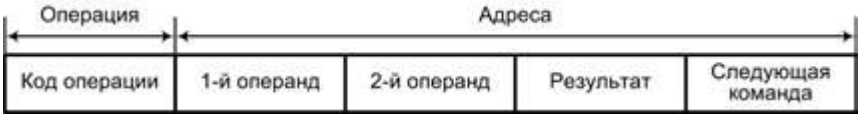


Рис. 2.34. Четырехадресный формат команды

В фон-неймановских ВМ надобность в четвертом адресе отпадает, поскольку команды располагаются в памяти в порядке их выполнения, и адрес очередной команды может быть получен за счет простого увеличения адреса текущей команды в счетчике команд. Это позволяет перейти к *трехадресному формату команды* (рис. 2.35). Требуется только добавить в систему команд ВМ команды, способные изменять порядок вычислений.

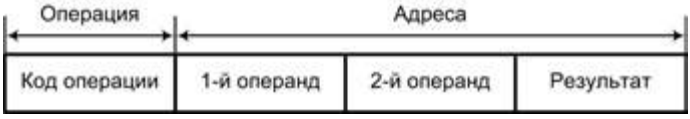


Рис. 2.35. Трехадресный формат команды

К сожалению, и в трехадресном формате длина команды может оказаться весьма большой. Так, если адрес ячейки основной памяти имеет длину 32 бита, а длина кода операции — 8 битов, то длина команды составит 104 бита (13 байтов).

Если по умолчанию взять в качестве адреса результата адрес одного из операндов (обычно второго), то можно обойтись без третьего адреса, и в итоге получаем *двухадресный формат команды* (рис. 2.36). Естественно, что в этом случае соответствующий операнд после выполнения операции теряется.

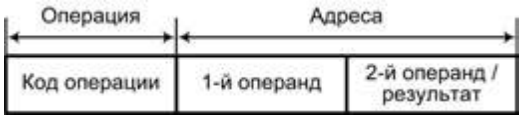


Рис. 2.36. Двухадресный формат команды

Команду можно еще более сократить, перейдя к *одноадресному формату* (рис. 2.37), что возможно при выделении определенного стандартного места для хранения первого операнда и результата. Обычно для этой цели используется специальный регистр центрального процессора (ЦП), известный под названием *аккумулятора*, поскольку здесь аккумулируется результат.

Рис. 2.37. Одноадресный формат команды

Применение единственного регистра для хранения одного из операндов и результата является ограничивающим фактором, поэтому помимо аккумулятора часто используют и другие регистры ЦП. Так как число регистров в ЦП невелико, для указания одного из них в команде достаточно иметь сравнительно короткое адресное поле. Соответствующий формат носит название *полутораадресного* или *регистрового формата* (рис. 2.38).

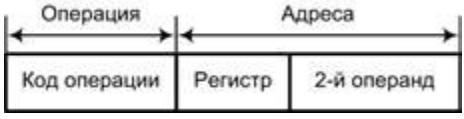


Рис. 2.38. Полутораадресный формат команды

Наконец, если для обоих операндов указать четко заданное местоположение, а также для команд, не требующих операнда, можно получить *нульадресный формат команды* (рис. 2.39).

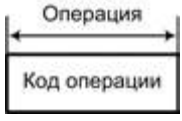


Рис. 2.39. Нульадресный формат команды

В таком варианте адресная часть команды вообще отсутствует или не задействуется.

Выбор адресности команд

При выборе количества адресов в адресной части команды обычно руководствуются следующими критериями:

- емкостью запоминающего устройства, требуемой для хранения программы;
- временем выполнения программы;
- эффективностью использования ячеек памяти при хранении программы.

Для оценки влияния адресности на каждый из перечисленных элементов воспользуемся методикой и выводами, изложенными в [20].

Адресность и емкость запоминающего устройства

Емкость запоминающего устройства для хранения программы E_A можно оценить из соотношения

$$E_A = N_A \times R_{K_A},$$

где N_A — количество команд в программе; R_{K_A} — разрядность команды, определяемая в соответствии с формулой (2.1); A — индекс, указывающий адресность команд программы. Оптимальная адресность команды определяется путем решения уравнения $\frac{\partial E_A}{\partial A} = 0$ при условии, что найденное значение обеспечивает минимум E_A. В [24] показано, что в среднем E_A монотонно возрастает с увеличением A. Таким образом, при выборе количества адресов по критерию «емкость ЗУ» предпочтение следует отдавать одноадресным командам.

Адресность и время выполнения программы

Время выполнения одной команды складывается из времени выполнения операции и времени обращения к памяти.

Для трехадресной команды последнее суммируется из четырех составляющих времени:

- выборки команды;
- выборки первого операнда;
- выборки второго операнда;
- записи в память результата.

Одноадресная команда требует двух обращений к памяти:

- выборки команды;
- выборки операнда.

Как видно, на выполнение одноадресной команды затрачивается меньше времени, чем на обработку трехадресной команды, однако для реализации одной трехадресной команды, как правило, нужно три одноадресных. Этих соображений тем не менее недостаточно, чтобы однозначно отдать предпочтение тому или иному варианту адресности. Определяющим при выборе является тип

алгоритмов, на преимущественную реализацию которых ориентирована конкретная ВМ.

В самой общей постановке время выполнения алгоритма T_A можно определить выражением:

$$T_A = N_a \tau_{a_A} + N_н \tau_{н_A}, \qquad (2.5)$$

в котором N_a — количество арифметических и логических команд в программе; τ_{a_A} — время выполнения одной арифметической или логической команды; $N_Н$ — количество неарифметических команд; $\tau_{н_A}$ — время выполнения одной неарифметической команды; $A = \{1, 2, 3\}$ — индекс, определяющий количество адресов в команде. В свою очередь, $N_Н$ можно определить как $N_н = N_у + N_{в_A}$, где $N_у$ — количество команд передачи управления (их число в программе не зависит от адресности), а $N_{в_A}$ — количество вспомогательных команд пересылок данных в регистр сумматора и из него.

Время выполнения как арифметической (τ_{a_A}), так и неарифметической ($\tau_{н_A}$) команды складывается из времени выборки команды из памяти τ_0 (τ_0 — время, затрачиваемое на одно обращение к памяти) и времени считывания/записи данных $A\tau_0$. В случае арифметической команды следует учесть также время на исполнение арифметической операции τ_a. Таким образом, имеем:

$$\tau_{a_A} = \tau_a + \tau_0 + A\tau_0 = \tau_a + (A+1)\tau_0, \qquad (2.6)$$

$$\tau_{н_A} = \tau_0 + A\tau_0 = (A+1)\tau_0, \qquad (2.7)$$

и выражение (2.5) принимает вид:

$$T_A = N_a[\tau_a + (A+1)\tau_0] + (N_у + N_{в_A})(A+1)\tau_0 . \qquad (2.8)$$

Подставляя в (2.8) значения $A = 1$ и $A = 3$, можно определить разность времен ΔT реализации алгоритма с помощью одноадресных и трехадресных команд, принимая во внимание, что для трехадресных команд $N_{2_3} = 0$:

$$\Delta T = T_1 - T_3 = 2\tau_o(N_{в_1} - N_a - N_у). \qquad (2.9)$$

Теперь проанализируем «выгодность» той или иной адресности команды в зависимости от типа целевого алгоритма. Возможные типы алгоритмов условно разделим на три группы:

- последовательные;
- параллельные;
- комбинированные.

Для последовательного алгоритма результат предшествующей команды используется в последующей. Здесь $N_{в_1} = 2$, так как требуется всего одна команда предварительной засылки числа в аккумулятор в начале вычисления и одна команда пересылки результата в память в конце вычислений. Если обозначить количество арифметических и логических команд в последовательном алгоритме как $N_{a_{посл}}$

($N_a = N_{a_{\text{посл}}}$), то выигрыш во времени для подобного алгоритма ($\Delta T_{\text{посл}}$), согласно выражению (2.9), составит

$$\Delta T_{\text{посл}} = 2\tau_0(2 - N_{a_{\text{посл}}} - N_{y_1}), \tag{2.10}$$

где $N_{\text{у}_1}$ — количество команд передачи управления. Таким образом, в последовательных алгоритмах чем больше $N_{a_{\text{посл}}}$, тем выгоднее оказываются одноадресные команды.

В параллельном алгоритме результат предыдущей команды не используется в последующей и должен быть отослан в память. В этом случае

$$N_a = N_{a_{\text{пар}}},\ N_{\text{в}_1} = 2N_{a_{\text{пар}}},$$

и выигрыш по времени определяется как

$$\Delta T_{\text{пар}} = 2\tau_0(N_{a_{\text{пар}}} - N_{\text{у}_2}), \tag{2.11}$$

где $N_{\text{у}_2}$ — количество операций передачи управления. Таким образом, при $N_{a_{\text{пар}}} > N_{\text{у}_2}$ целесообразно ориентироваться на трехадресные команды.

В комбинированном алгоритме вычислительный процесс образуют как последовательные, так и параллельные части, при этом

$$N_a = N_{a_{\text{посл}}} + N_{a_{\text{пар}}},$$

и выигрыш во времени $\Delta T_{\text{комб}}$ с учетом (2.10) и (2.11) можно оценить как

$$\Delta T_{\text{комб}} = \Delta T_{\text{пар}} + \Delta T_{\text{посл}} = 2\tau_0(2 + N_{a_{\text{пар}}} - N_{a_{\text{посл}}} - N_{\text{у}}), \tag{2.12}$$

где $N_{\text{у}} = N_{\text{у}_1} + N_{\text{у}_2}$ — количество команд передачи управления в обеих частях алгоритма.

Из (2.12) следует, что при $N_{a_{\text{посл}}} + N_{\text{у}} > 2 + N_{a_{\text{пар}}}$ предпочтение следует отдать одноадресным командам.

Двухадресные команды в плане времени реализации алгоритмов занимают промежуточное положение между одноадресными и трехадресными. Несколько лучшие показатели дают полуторадресные команды, в которых, с одной стороны, сохраняются преимущества одноадресных команд для последовательных алгоритмов, а с другой — повышается эффективность реализации параллельных и комбинированных алгоритмов.

Способы адресации операндов

Вопрос о том, каким образом в адресном поле команды может быть указано местоположение операндов, считается одним из центральных при разработке АСК. С точки зрения сокращения аппаратурных затрат очевидно стремление разработчиков уменьшить длину адресного поля при сохранении возможностей доступа ко всему адресному пространству. С другой стороны, способ задания адресов должен способствовать максимальному сближению операторов языков

программирования высокого уровня и машинных команд. Все это привело к тому, что в архитектуре системы команд любой ВМ предусмотрены различные способы адресации операндов.

Приступая к рассмотрению способов адресации, вначале определим понятия «исполнительный адрес» и «адресный код».

Исполнительным адресом операнда ($A_{\text{ИСП}}$) называется двоичный код номера ячейки памяти, служащей источником или приемником операнда. Этот код подается на адресные входы запоминающего устройства (ЗУ) и по нему происходит фактическое обращение к указанной ячейке. Если операнд хранится не в основной памяти, а в регистре процессора, его исполнительным адресом будет номер регистра.

Адресный код команды ($A_{\text{К}}$) — это двоичный код в адресном поле команды, из которого необходимо сформировать исполнительный адрес операнда.

В современных ВМ исполнительный адрес и адресный код, как правило, не совпадают, и для доступа к данным требуется соответствующее преобразование. *Способ адресации* — это способ формирования исполнительного адреса операнда по адресному коду команды. Способ адресации существенно влияет на параметры процесса обработки информации. Одни способы позволяют увеличить емкость адресуемой памяти без удлинения команды, но снижают скорость выполнения операции, другие — ускоряют операции над массивами данных, третьи — упрощают работу с подпрограммами и т. д. В сегодняшних ВМ обычно имеется возможность приложения нескольких различных способов адресации операндов к одной и той же операции.

Чтобы устройство управления вычислительной машины могло определить, какой именно способ адресации принят в данной команде, в разных ВМ используются различные приемы. Часто разным способам адресации соответствуют и разные коды операции. Другой подход — это добавление в состав команды специального *поля способа адресации*, содержимое которого определяет, какой из способов адресации должен быть применен. Иногда в команде имеется нескольких полей — по одному на каждый адрес. Отметим, что возможен также вариант, когда в команде вообще отсутствует адресная информация, то есть имеет место *неявная адресация*. При неявной адресации адресного поля либо просто нет, либо оно содержит не все необходимые адреса — отсутствующий адрес подразумевается кодом операции. Так, при исключении из команды адреса результата подразумевается, что результат помещается на место второго операнда. Неявная адресация применяется достаточно широко, поскольку позволяет сократить длину команды.

Выбор способов адресации является одним из важнейших вопросов разработки системы команд и всей ВМ в целом, при этом существенное значение имеет не только удобство программирования, но и эффективность способа. Для оценки эффективности различных способов адресации обратимся к методике и выводам, изложенным в [20]. Согласно предложенной методике, эффективность способа адресации можно характеризовать двумя показателями: затратами оборудования C и затратами времени T на доступ к адресуемым данным. Затраты оборудования определяются суммой:

$$C = C_{\text{ВА}} + C_{\text{ЗУ}}, \tag{2.13}$$

где $C_{ВА}$ — затраты аппаратных средств, обеспечивающих вычисление исполнительных адресов; $C_{ЗУ}$ — затраты памяти на хранение адресных кодов команд. Обычно $C_{ЗУ} >> C_{ВА}$, поэтому при оценке затрат оборудования ограничиваются учетом величины $C_{ЗУ}$. Затраты времени T определяются суммой времени $t_{ФИА}$ формирования исполнительного адреса и времени $t_{ЗУ}$ выборки или записи операнда:

$$T = t_{ФИА} + t_{ЗУ}. \quad (2.14)$$

В настоящее время используются различные способы адресации, наиболее распространенные из которых рассматриваются ниже.

Непосредственная адресация

При *непосредственной адресации* (НА) в адресном поле команды вместо адреса содержится непосредственно сам операнд (рис. 2.40). Этот способ может применяться при выполнении арифметических операций, операций сравнения, а также для загрузки констант в регистры.

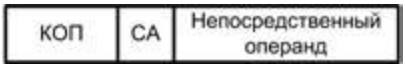


Рис. 2.40. Непосредственная адресация

При записи в регистр, имеющий разрядность, превышающую длину непосредственного операнда, операнд размещается в младшей части регистра, а оставшиеся свободными позиции заполняются значением знакового бита операнда.

Помимо того, что в адресном поле могут быть указаны только константы, еще одним недостатком данного способа адресации является то, что размер непосредственного операнда ограничен длиной адресного поля команды, которое в большинстве случаев меньше длины машинного слова.

В 50–60% команд с непосредственной адресацией длина операнда не превышает 8 битов, а в 75–80% — 16 битов и лишь в 20–25% случаев непосредственный операнд имеет длину более 16 битов. По оценке Э. Таненбаума [152], в 98% случаев непосредственный операнд укладывается в 13 битов. Таким образом, в подавляющем числе случаев шестнадцати разрядов вполне достаточно, хотя для вычисления адресов могут потребоваться и более длинные константы.

Исследования показывают, что средний процент использования непосредственной адресации по всем командам составляет 35% для целочисленных вычислений и 10% — в программах, ориентированных на обработку чисел с плавающей запятой. Наиболее интенсивно данный вид адресации используется в арифметических операциях и командах сравнения. В то же время загрузка констант в большинстве программ не такая частая операция.

Непосредственная адресация сокращает время выполнения команды, так как не требуется обращение к памяти за операндом. Кроме того, экономится память, поскольку отпадает необходимость в ячейке для хранения операнда. В плане эффективности этот способ можно считать «идеальным» ($C_{НА} = 0$, $T_{НА} = 0$) и его можно рекомендовать к использованию во всех ситуациях, когда тому не препятствуют вышеупомянутые ограничения.

Прямая адресация

При прямой или абсолютной адресации (ПА) адресный код прямо указывает номер ячейки памяти, к которой производится обращение (рис. 2.41), то есть адресный код совпадает с исполнительным адресом.

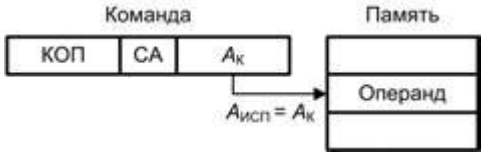


Рис. 2.41. Прямая адресация

При всей простоте использования способ имеет существенный недостаток — ограниченный размер адресного пространства, так как для обращения к памяти большой емкости нужно «длинное» адресное поле. Однако более существенным несовершенством можно считать то, что адрес, указанный в команде, не может быть изменен в процессе вычислений (во всяком случае, такое изменение не рекомендуется). Это ограничивает возможности по произвольному размещению программы (и данных) в памяти.

Прямую адресацию характеризуют следующие показатели эффективности: $C_{ПА} = \lceil \log_2 N_i \rceil$, $T_{ПА} = t_{ЗУ}$, где N_i — количество адресуемых операндов.

Косвенная адресация

Одним из путей преодоления проблем, свойственных прямой адресации, может служить прием, когда с помощью ограниченного адресного поля команды указывается адрес ячейки, в свою очередь, содержащей полноразрядный адрес операнда (рис. 2.42). Этот способ известен как *косвенная адресация* (КА). Запись $(A_К)$ означает содержимое ячейки, адрес которой указан в скобках.

При косвенной адресации содержимое адресного поля команды остается неизменным, в то время как косвенный адрес в процессе выполнения программы можно изменять. Это позволяет проводить вычисления, когда адреса операндов заранее неизвестны и появляются лишь в процессе решения задачи. Дополнительно такой прием упрощает обработку массивов и списков, а также передачу параметров подпрограммам.

Рис. 2.42. Косвенная адресация

Недостатком косвенной адресации является необходимость в двухкратном обращении к памяти: сначала для извлечения адреса операнда, а затем для обращения к операнду ($T_{КА} = 2t_{ЗУ}$). Сверх того, задействуется лишняя ячейка памяти для хранения исполнительного адреса операнда. Способу свойственны следующие затраты оборудования:

$$C_{КА} = R_{ЯЧ} + \lceil \log_2 N_A \rceil \geqslant \lceil \log_2 (N_i + N_A) \rceil, \qquad (2.15)$$

где $R_{ЯЧ}$ — разрядность ячейки памяти, хранящей исполнительный адрес; N_A — количество ячеек для хранения исполнительных адресов; N_i — количество адресуемых операндов. Здесь выражение $\lceil \log_2 N_A \rceil$ определяет разрядность сокращенного адресного поля команды (обычно $N_A << N_i$).

В качестве варианта косвенной адресации, правда, достаточно редко используемого, можно упомянуть *многоуровневую* или *каскадную косвенную адресацию*: $A_{ИСП} = (...(A_К)...)$, когда к исполнительному адресу ведет цепочка косвенных адресов. В этом случае один из битов в каждом адресе служит признаком косвенной адресации. Состояние бита указывает, является ли содержимое ячейки очередным адресом в цепочке адресов или это уже исполнительный адрес операнда. Особых преимуществ у такого подхода нет, но в некоторых специфических ситуациях он оказывается весьма удобным, например, при обработке многомерных массивов. В то же время очевиден и его недостаток — для доступа к операнду требуется три и более обращений к памяти.

Регистровая адресация

Регистровая адресация (РА) напоминает прямую адресацию. Различие состоит в том, что адресное поле команды указывает не на ячейку памяти, а на регистр процессора (рис. 2.43). Адрес регистра в дальнейшем будем обозначать буквой *R*. Обычно размер адресного поля в данном случае составляет три или четыре бита, что позволяет указать соответственно на один из 8 или 16 регистров общего назначения (РОН).

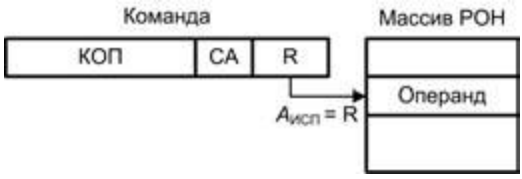


Рис. 2.43. Регистровая адресация

Двумя основными преимуществами регистровой адресации являются: короткое адресное поле в команде и исключение обращений к памяти. Малое число РОН позволяет сократить длину адресного поля команды, то есть $C_{РА} << C_{ПА}$. Кроме того, $T_{РА} = t_{РОН}$, где $t_{РОН}$ — время выборки операнда из регистра общего назначения, причем $t_{РОН} << t_{ЗУ}$. К сожалению, возможности по использованию регистровой адресации ограничены малым числом РОН в составе процессора.

Косвенная регистровая адресация

Косвенная регистровая адресация (КРА) представляет собой косвенную адресацию, где исполнительный адрес операнда хранится не в ячейке основной памяти, а в регистре процессора. Соответственно, адресное поле команды указывает не на ячейку памяти, а на регистр (рис. 2.44).

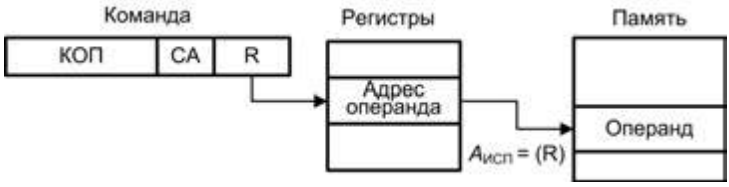


Рис. 2.44. Косвенная регистровая адресация

Достоинства и ограничения косвенной регистровой адресации те же, что и у обычной косвенной адресации, но благодаря тому что косвенный адрес хранится не в памяти, а в регистре, для доступа к операнду требуется на одно обращение к памяти меньше. Эффективность косвенной регистровой адресации можно оценить по формулам:

$$T_{КРА} = t_{РОН} + t_{ЗУ};\ C_{КРА} = R_{РОН} + \lceil \log_2 N_A \rceil \geqslant \lceil \log_2 (N_i + N_A) \rceil, \qquad (2.16)$$

где $R_{РОН}$ — разрядность регистров общего назначения.

Адресация со смещением

При адресации со смещением исполнительный адрес формируется в результате суммирования содержимого адресного поля команды с содержимым одного или нескольких регистров процессора (рис. 2.45).

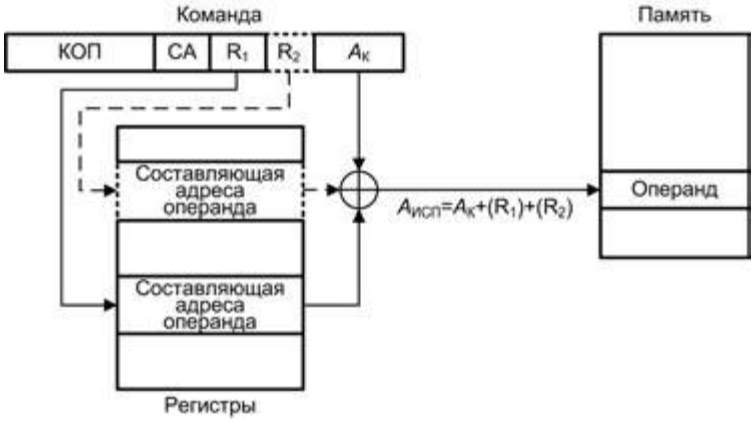


Рис. 2.45. Адресация со смещением

Адресация со смещением предполагает, что адресная часть команды включает в себя, как минимум, одно поле ($A_К$). В нем содержится константа, смысл которой в разных

вариантах адресации со смещением может меняться. Константа может представлять собой некий базовый адрес, к которому добавляется хранящееся в регистре смещение. Допустим и прямо противоположный подход: базовый адрес находится в регистре процессора, а в поле $A_К$ указывается смещение относительно этого адреса. В некоторых процессорах для реализации определенных вариантов адресации со смещением предусмотрены специальные регистры, например базовый или индексный. Использование таких регистров предполагается по умолчанию, поэтому адресная часть команды содержит только поле $A_К$. Если же составляющая адреса может располагаться в произвольном регистре общего назначения, то для указания конкретного регистра в команду включается дополнительное поле R (при составлении адреса более чем из двух составляющих в команде будет несколько таких полей). Еще одно поле R может появиться в командах, где смещение перед вычислением исполнительного адреса умножается на масштабный коэффициент. Такой коэффициент заносится в один из РОН, на который и указывает это дополнительное поле. В наиболее общем случае адресация со смещением подразумевает наличие двух адресных полей: $A_К$ и R.

В рамках адресации со смещением имеется еще один вариант, при котором исполнительный адрес вычисляется не суммированием, а конкатенацией (присоединением) составляющих адреса. Здесь одна составляющая представляет собой старшую часть исполнительного адреса, а вторая — младшую.

Ниже рассматриваются основные способы адресации со смещением, каждый из которых, впрочем, имеет собственное название.

Относительная адресация

При *относительной адресации* для получения исполнительного адреса операнда содержимое поля $A_К$ команды складывается с содержимым счетчика команд (рис. 2.46). Таким образом, адресный код в команде представляет собой смещение относительно адреса текущей команды. Следует отметить, что в момент вычисления исполнительного адреса операнда в счетчике команд может уже быть сформирован адрес следующей команды, что нужно учитывать при выборе величины смещения. Обычно поле $A_К$ трактуется как двоичное число в дополнительном коде.

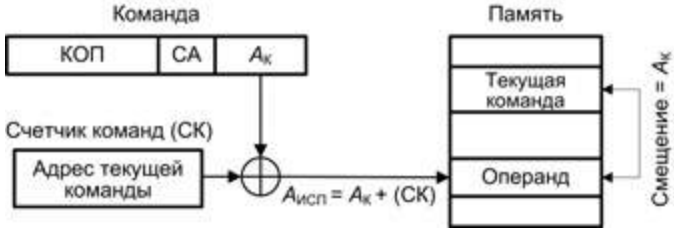


Рис. 2.46. Относительная адресация

Адресация относительно счетчика команд базируется на свойстве локальности, выражающемся в том, что большая часть обращений происходит к ячейкам, расположенным в непосредственной близости от выполняемой команды. Это позволяет

сэкономить на длине адресной части команды, поскольку разрядность поля $A_К$ может быть небольшой. Главное достоинство данного способа адресации состоит в том, что он делает программу перемещаемой в памяти: независимо от текущего расположения программы в адресном пространстве взаимное положение команды и операнда остается неизменным, поэтому адресация операнда остается корректной.

Эффективность данного способа адресации (обозначим его СА — «относительно Счетчика Адресация») можно описать выражениями:

$$T_{СА} = t_{РОН} + t_{СЛ} + t_{ЗУ};\ C_{СА} = \lceil \log_2 N_i - R_{СК} \rceil, \qquad (2.17)$$

где $R_{СК}$ — разрядность счетчика команд; $t_{СЛ}$ — время сложения составляющих исполнительного адреса.

Базовая регистровая адресация

В случае *базовой регистровой адресации* (БРА) регистр, называемый базовым, содержит полноразрядный адрес, а поле $A_К$ — смещение относительно этого адреса. Ссылка на базовый регистр может быть явной или неявной. В некоторых ВМ имеется специальный базовый регистр и его использование является неявным, то есть поле R в команде отсутствует (рис. 2.47, *а*).

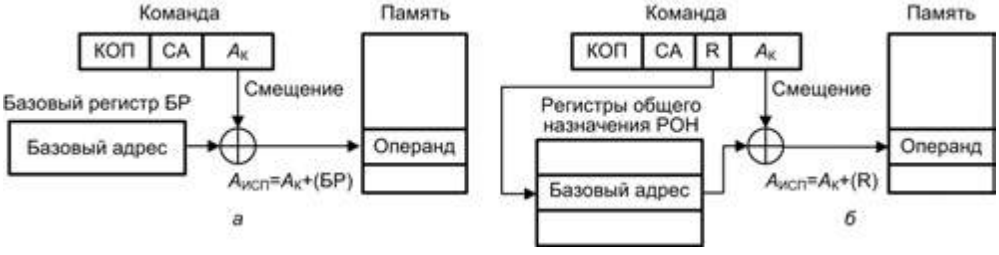


Рис. 2.47. Базовая регистровая адресация: *а* — с базовым регистром; *б* — с использованием одного из РОН

Более типичен случай, когда в роли базового регистра выступает один из регистров общего назначения (РОН), тогда его номер явно указывается в поле R команды (рис. 2.47, *б*).

Базовую регистровую адресацию обычно используют для доступа к элементам массива, положение которого в памяти в процессе вычислений может меняться. В базовый регистр заносится начальный адрес массива, а адрес элемента массива указывается в поле $A_К$ команды в виде смещения относительно начального адреса массива. Достоинство данного способа адресации в том, что смещение имеет меньшую длину, чем полный адрес, и это позволяет сократить длину адресного поля команды. Короткое смещение расширяется до полной длины исполнительного адреса путем добавления слева битов, совпадающих со значением знакового разряда смещения.

Разрядность смещения $R_{СМ}$ и, соответственно, затраты оборудования определяются из условия $R_{СМ} = C_{БРА} = \left\lceil \log_2(\max_i(N_{ОП_i})) \right\rceil$, где $N_{ОП_i}$ — количество операндов i-й программы.

Затраты времени составляют: $T_{БРА} = t_{РОН} + t_{СЛ} + t_{ЗУ}$.

Индексная адресация

При *индексной адресации* (ИА) поле $A_К$ содержит адрес ячейки памяти, а регистр (указанный явно или неявно) — смещение относительно этого адреса. Как видно, этот способ адресации похож на базовую регистровую адресацию. Поскольку при индексной адресации в поле $A_К$ находится полноразрядный адрес ячейки памяти, играющий роль базы, длина этого поля больше, чем при базовой регистровой адресации. Тем не менее вычисление исполнительного адреса операнда производится идентично (рис. 2.48).

Рис. 2.48. Индексная адресация: *а* — с индексным регистром; *б* — с использованием одного из РОН

Индексная адресация предоставляет удобный механизм для организации итеративных вычислений. Пусть, например, имеется массив чисел, расположенных в памяти последовательно, начиная с адреса N, и мы хотим увеличить на единицу все элементы данного массива. Для этого требуется извлечь каждое число из памяти, прибавить к нему 1 и вернуть обратно, а последовательность исполнительных адресов будет следующей: $N, N+1, N+2$ и т. д., вплоть до последней ячейки, занимаемой рассматриваемым массивом. Значение N берется из поля $A_К$ команды, а в выбранный регистр, называемый *индексным регистром*, сначала заносится 0. После каждой операции содержимое индексного регистра увеличивается на 1.

Так как это довольно типичный случай, в большинстве ВМ увеличение или уменьшение содержимого индексного регистра до или после обращения к нему осуществляется автоматически как часть машинного цикла. Такой прием называется *автоиндексированием*. Если для индексной адресации используются специально выделенные регистры, автоиндексирование может производиться неявно и автоматически. При задействовании для хранения индексов регистров общего назначения необходимость автоиндексирования должна указываться в команде специальным битом.

Автоиндексирование с увеличением содержимого индексного регистра носит название *автоинкрементной адресации* и может быть описано следующим образом:

$$A_{ИСП} = A_К + (R),\ R \leftarrow (R) + 1 \text{ или } R \leftarrow (R) + 1,\ A_{ИСП} = A_К + (R), \qquad (2.18)$$

где (R) — содержимое индексного регистра с адресом R.

В первом варианте увеличение содержимого индексного регистра происходит после формирования исполнительного адреса, и этот способ называется *постин-*

крементным автоиндексированием. Во втором случае сначала производится увеличение содержимого индексного регистра, и уже новое значение используется для формирования исполнительного адреса. Тогда говорят о *преинкрементном автоиндексировании*.

Автоиндексирование с уменьшением содержимого индексного регистра носит название *автодекрементной адресации* и может быть описано так:

$$A_{\text{ИСП}} = A_{\text{К}} + (R),\ R \leftarrow (R) - 1 \text{ или } R \leftarrow (R) - 1,\ A_{\text{ИСП}} = A_{\text{К}} + (R). \qquad (2.19)$$

Здесь также возможны два варианта, отличающиеся последовательностью выполнения операций уменьшения содержимого индексного регистра и вычисления исполнительного адреса: *постдекрементное автоиндексирование* и *предекрементное автоиндексирование*.

Интересным и весьма полезным является еще один вариант индексной адресации — *индексная адресация с масштабированием и смещением*: содержимое индексного регистра умножается на масштабный коэффициент и суммируется с $A_{\text{К}}$. Масштабный коэффициент может принимать значения 1, 2, 4 или 8, для чего в адресной части команды выделяется дополнительное поле. Описанный способ адресации реализован, например, в микропроцессорах фирмы Intel.

Следует особо отметить, что система команд многих ВМ предоставляет возможность различным образом сочетать базовую и индексную адресации в качестве дополнительных способов адресации.

Страничная адресация

Страничная адресация (СТА) предполагает разбиение адресного пространства на страницы. Страница определяется своим начальным адресом, выступающим в качестве базы. Старшая часть этого адреса хранится в специальном регистре — *регистре адреса страницы* (РАС). В адресном коде команды указывается смещение внутри страницы, рассматриваемое как младшая часть исполнительного адреса. Исполнительный адрес образуется конкатенацией (присоединением) $A_{\text{К}}$ к содержимому РАС, как показано на рис. 2.49. На рисунке символ точки • обозначает операцию конкатенации.

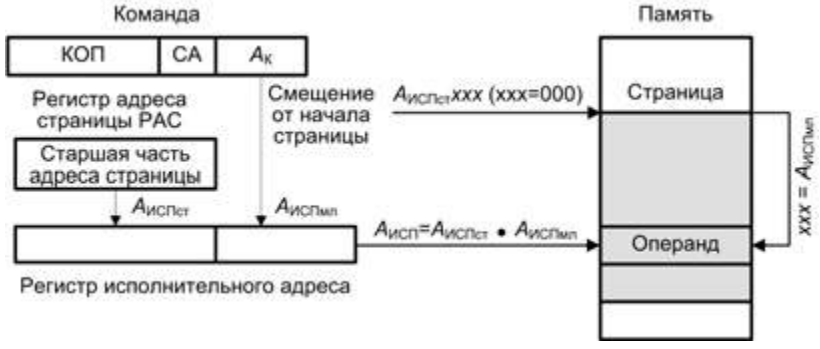


Рис. 2.49. Страничная адресация

Показатели эффективности страничной адресации имеют вид:

$$C_{\text{СТА}} = \lceil \log_2 N_i - \log_2 M \rceil,\ T_{\text{СТА}} = T_{\text{РОН}} + T_{\text{ЗУ}}, \tag{2.20}$$

где M — количество страниц в памяти.

Блочная адресация

Блочная адресация используется в командах, для которых единицей обработки служит блок данных, расположенных в смежных ячейках памяти. Этот способ очень удобен при работе с внешними запоминающими устройствами и в операциях с векторами. Для описания блока обычно берется адрес ячейки, где хранится первый или последний элемент блока, и общее количество элементов блока, заданное числом байтов или ячеек. Вместо длины блока может использоваться специальный признак «конец блока», помещаемый за последним элементом блока.

Стековая адресация

Данный вид адресации был рассмотрен при описании стековой архитектуры системы команд.

Распространенность различных видов адресации

Частота использования различных способов адресации существенно зависит от типа АСК. Для машин со стековой архитектурой очевидно, что основным способом адресации является стековая адресация. Для ВМ с аккумуляторной АСК главные способы адресации — это прямая и непосредственная.

Достаточно ясна и ситуация с RISC-архитектурой. Из самой идеи этого подхода вытекает, что преимущественный способ адресации здесь — регистровая адресация.

Более сложным является вопрос о частоте использования различных видов адресации в регистровых ВМ. В рамках этой архитектуры существует множество машин с самыми разнообразными списками команд и различными сочетаниями способов адресации, в силу чего дать однозначный ответ относительно наиболее распространенных вариантов практически невозможно.

Единственное общее замечание — интенсивность применения конкретных способов адресации ощутимо зависит от характера решаемой задачи. Это обстоятельство обязательно должно учитываться пользователями при выборе ВМ под конкретное применение.

Способы адресации в командах управления потоком команд

Основными способами адресации в командах управления потоком команд являются прямая и относительная.

Для команд безусловного и условного перехода (ветвления) наиболее типична относительная адресация, когда в адресной части команды указывается смещение адреса точки перехода относительно текущей команды, то есть смещение относительно текущего содержимого счетчика команд. Использование данного способа адресации позволяет программе выполняться в любом месте памяти — программы

становятся перемещаемыми. Среди команд безусловного перехода доля относительной адресации составляет около 90%.

Для команд перехода чрезвычайно важно, насколько далеко адрес перехода отстоит от адреса команды перехода, иными словами, какова типичная величина смещения. Исследования показывают, длина смещения в основном не превышает 8 битов, что соответствует смещению в пределах ±128 относительно команды ветвления. В подавляющем большинстве случаев переход идет в пределах 3–7 команд относительно команды перехода.

Система операций

Системой операций называется список операций, непосредственно выполняемых техническими средствами вычислительной машины. Система операций ВМ определяется областью ее применения, требованиями к стоимости, производительности и точности вычислений.

Связь системы операций с алгоритмами решаемых задач проявляется в степени ее приспособленности для записи программ реализации этих алгоритмов. Степень приспособленности характеризуется близостью списка операций системы команд и операций, используемых на каждом шаге выполнения алгоритмов. Простоту программирования алгоритма часто определяют термином «программируемость вычислительной машины». Чем меньше команд требуется для составления программы реализации какого-либо алгоритма, тем программируемость выше. В архитектурах типа CISC улучшения программируемости добиваются введением в систему операций большого количества операций, в том числе и достаточно сложных. Это может приводить и к повышению производительности ВМ, хотя в любом случае увеличивает аппаратурные затраты.

Обоснованный выбор системы операций (СО) возможен лишь исходя из анализа подлежащих реализации алгоритмов. Для этого определяется частотный вектор используемых в алгоритме операторов (q_1, ..., q_n). Изучив вектор, составляют список основных, наиболее часто встречающихся операторов. Операторы основного списка реализуются системой машинных операций ВМ (каждому оператору сопоставляется своя машинная операция). Остальные операторы алгоритма получают путем их разложения на операторы основного списка.

Показатели эффективности системы операций

Качество системы операций (СО) можно характеризовать двумя свойствами: функциональной полнотой и эффективностью.

Функциональная полнота — это достаточность системы операций для описания любых алгоритмов. Системы операций ВМ включают в себя большое количество машинных операций и практически всегда являются функционально полными.

Эффективность системы операций показывает степень ее соответствия заданному классу алгоритмов и требованиям к производительности ВМ. Количественно эффективность характеризуется затратами оборудования, затратами времени на реализацию алгоритмов и вероятностью правильного выполнения программ.

Затраты оборудования C можно описать выражением

$$C = C_{\text{ПР}} + C_{\text{ЗУ}},$$

где $C_{\text{ПР}}$ — затраты в процессоре на реализацию системы операций, $C_{\text{ЗУ}}$ — затраты памяти на размещение данных и программ, представляющих алгоритм в терминах заданной системы операций.

Величина $C_{\text{ПР}}$ пропорциональна количеству и сложности машинных операций, а $C_{\text{ЗУ}}$ — емкости памяти, необходимой для хранения закодированного алгоритма. Усложнение машинных операций приводит к сокращению количества операций (команд), требуемых для описания алгоритма, и, следовательно, к уменьшению необходимой емкости памяти.

Затраты времени на реализацию алгоритма (T) пропорциональны количеству команд (операций) в программе. Введение в СО более сложных операций позволяет программировать сложные действия одной командой, в результате чего уменьшается количество команд программы.

Выбор системы операций

Выбор оптимальной системы операций является сложной задачей. Основные трудности в ее решении связаны с установлением точной функциональной зависимости показателей эффективности C, T от состава СО. Поэтому найти чисто формальный метод выбора оптимальной СО пока не удается, а существующие подходы к ее решению основываются на комбинации формальных и эвристических приемов.

В качестве примера рассмотрим один из способов решения этой задачи — выбор системы операций на основе структурирования алгоритмов.

Метод структурирования алгоритмов предполагает следующую формулировку задачи выбора системы операций: необходимо выбрать такую систему F_n, которая обеспечивает реализацию алгоритма A за заданное время $T = T_{\text{ДОП}}$ при минимальной стоимости ВМ.

Иерархия операций $F_1, ..., F_p$, функционально полных для алгоритма A, может быть определена процедурой структурирования [17], сводящейся к следующему. Алгоритм A рассматривается как один оператор, реализующий операцию f_1 над исходными данными с целью получения требуемых результатов, то есть $F_1 = \{f_1\}$. Затем оператор разделяется на части — программируется последовательностью более простых операторов. Последовательное применение процедуры структурирования (разделения оператора на более простые операторы) позволяет выявить системы операций $F_1 = \{f_1\}$, $F_2 = \{f_2, f_3\}$, $F_3 = \{f_3, f_4, f_5\}$, $F_4 = \{f_4, f_5, f_6, f_7\}$, ... , и тем самым построить иерархию операций $F_1, ..., F_p$.

Для каждой операции в F_i можно определить количество n_g ее выполнений при одной реализации алгоритма. Тогда сумма

$$N_i = \sum_{f_g \in F_i}^{G} n_g \qquad (2.21)$$

будет представлять количество операций, выполняемых при одной реализации алгоритма, запрограммированного в терминах F_i. Характеристики элементной базы позволяют задать приближенное значение средней длительности τ_i операции в ВМ. С учетом этого время выполнения алгоритма на основе F_i составит $T_i = \tau_i N_i$, что дает возможность поставить в соответствие иерархии систем операций $F_1, ..., F_p$ затраты времени на реализацию алгоритма $T_1, ..., T_p$, причем $T_1 < ... < T_p$. Можно предположить, что минимум аппаратных затрат достигается при $F_n \in \{F_1, ..., F_p\}$, обеспечивающей время реализации алгоритма T_n максимально близкое к заданному значению $T_{\text{ДОП}}$. В силу сказанного, выбор системы операций сводится к нахождению такой системы F_n, для которой разность $(T_{\text{ДОП}} - T_n)$ имеет минимальное положительное значение.

Контрольные вопросы

1. Какие характеристики вычислительной машины охватывает понятие «архитектура системы команд»?
2. Охарактеризуйте эволюцию архитектур системы команд вычислительных машин.
3. В чем состоит проблема семантического разрыва?
4. Поясните различия в подходах по преодолению семантического разрыва, применяемых в ВМ с CISC- и RISC-архитектурами.
5. Какая форма записи математических выражений наиболее соответствует стековой архитектуре системы команд и почему?
6. Какие средства используются для ускорения доступа к вершине стека в ВМ со стековой архитектурой?
7. Чем обусловлено возрождение интереса к стековой архитектуре?
8. Какие особенности аккумуляторной архитектуры можно считать ее достоинствами и недостатками?
9. Какие доводы можно привести за и против увеличения числа регистров общего назначения в ВМ с регистровой архитектурой системы команд?
10. Почему для ВМ с RISC-архитектурой наиболее подходящей представляется АСК с выделенным доступом к памяти?
11. Какую позицию запятой в формате с фиксированной запятой можно считать общепринятой?
12. Чем в формате с фиксированной запятой заполняются избыточные старшие разряды?
13. Какое минимальное количество полей должен содержать формат с плавающей запятой?
14. Как в формате с плавающей запятой решается проблема работы с порядками, имеющими разные знаки?
15. В чем состоит особенность трактовки нормализованной мантиссы в стандарте IEEE 754?
16. От чего зависят точность и диапазон представления чисел в формате с плавающей запятой?

17. Чем обусловлено появление форматов с упакованными числами в современных микропроцессорах?
18. Какие факторы влияют на выбор разрядности целых чисел?
19. Сказывается ли на производительности ВМ порядок следования в памяти байтов «длинного» числа и выбор адреса, с которого начинается запись числа?
20. По какому признаку при передаче потока десятичных чисел можно определить окончание одного числа и начало следующего?
21. Какой общий принцип лежит в основе различных таблиц кодировки символов?
22. Чем обусловлен переход от кодировки ASCII к кодировке Unicode?
23. В чем состоит особенность обработки логических данных?
24. Какие трактовки включает в себя понятие «строка»?
25. Перечислите способы представления графической информации и охарактеризуйте особенности каждого из них.
26. Каким образом в вычислительной машине представляется аудиоинформация?
27. Какой вид команд пересылки данных характерен для ВМ с RISC-архитектурой?
28. Чем вызвана необходимость заполнения освободившихся разрядов значением знакового разряда при арифметическом сдвиге вправо?
29. В чем состоит особенность SIMD-команд и в каком формате должны быть представлены операнды?
30. Что такое «арифметика с насыщением» и где она применяется?
31. Какие виды команд относят к командам ввода/вывода?
32. Какие виды команд условного перехода обычно доминируют в реальных программах?
33. Какие факторы определяют выбор формата команд?
34. Перечислите возможные пути сокращения длины кода команды.
35. Какая особенность фон-неймановской архитектуры позволяет отказаться от указания в команде адреса очередной команды?
36. Какие факторы необходимо учитывать при выборе оптимальной адресности команд?
37. С какими ограничениями связано использование непосредственной адресации?
38. В каких случаях может быть удобна многоуровневая косвенная адресация?
39. Какие преимущества дает адресация относительно счетчика команд?
40. В чем проявляются сходство и различия между базовой и индексной адресацией?
41. В чем состоит сущность автоиндексирования и в каких ситуациях оно применяется?
42. С какой целью применяется адресация с масштабированием?
43. Какие способы адресации переходов используются в командах управления потоком команд?
44. Как можно оценить эффективность системы операций при разработке архитектуры системы команд?

Глава 3

Функциональная организация фон-неймановской ВМ

Данная глава посвящена рассмотрению базовых принципов построения и функционирования фон-неймановских вычислительных машин.

Функциональная схема фон-неймановской вычислительной машины

Чтобы получить более детальное представление о структуре и функциях устройств ВМ, представим фон-неймановскую ВМ в виде гипотетической машины с аккумуляторной архитектурой (рис. 3.1).

Примем, что гипотетическая ВМ имеет следующие особенности:

- *Одноадресные команды*. Адресная часть команды (АЧ) содержит только один адрес. При выполнении операций с двумя операндами предполагается, что операнд, адрес которого в команде не указан, находится в специальном регистре АЛУ — аккумуляторе, а также, что результат остается в аккумуляторе.
- *Единство форматов*. Длина команд и данных совпадает с разрядностью ячеек памяти, то есть любая команда или операнд занимают только одну ячейку памяти. В этом случае адрес очередной команды в памяти может быть получен путем прибавления единицы к адресу текущей команды, а для извлечения из памяти любой команды или любого операнда достаточно одного обращения к памяти.

Список команд, выполняемых гипотетической ВМ, приведен в табл. 3.1.

На функциональной схеме (рис. 3.1) показаны типовые узлы гипотетической ВМ, а также сигналы, инициирующие выполнение отдельных операций по пересылке информации и ее обработке.

Устройство управления

Назначение устройства управления (УУ) было определено ранее при рассмотрении структурной схемы ВМ, где отмечалось, что эта часть ВМ организует

автоматическое выполнение программ и функционирование ВМ как единой системы. Теперь остановимся на описании узлов, входящих в состав УУ.

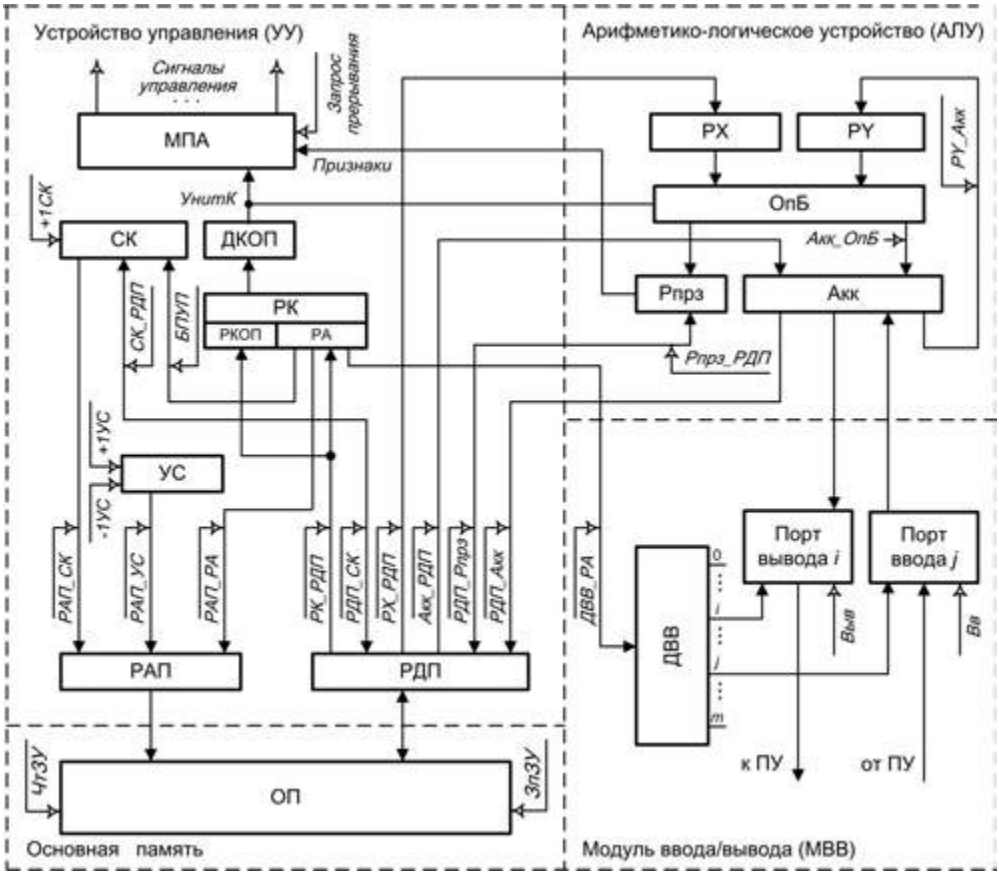


Рис. 3.1. Функциональная схема гипотетической фон-неймановской ВМ

Таблица 3.1. Команды гипотетической вычислительной машины

Мнемоническое обозначение	КОП[1]	АЧ	Описание
LDA	1	ADR	Загрузка в аккумулятор содержимого ячейки основной памяти (ОП) с адресом ADR
STA	2	ADR	Запись содержимого аккумулятора в ячейку ОП с адресом ADR
ADD	3	ADR	Сложение содержимого аккумулятора и ячейки ОП, имеющей адрес ADR. Результат остается в аккумуляторе

продолжение

Таблица 3.1 (*продолжение*)

Мнемоническое обозначение	КОП[1]	АЧ	Описание
SUB	4	ADR	Вычитание из содержимого аккумулятора числа, хранящегося в ячейке ОП, имеющей адрес ADR. Результат остается в аккумуляторе
INP	5	IPRT	Ввод в аккумулятор информации с периферийного устройства, подключенного к порту ввода с номером IPRT
OUT	6	OPRT	Вывод содержимого аккумулятора на периферийное устройство, подключенное к порту вывода с номером OPRT
JMP	7	ADR	Безусловный переход к команде, хранящейся по адресу ADR
BRZ	8	ADR	Переход к команде, хранящейся по адресу ADR, при условии, что результат предыдущей арифметической операции равен 0, иначе естественный порядок вычислений не нарушается
	9-E		Прочие возможные команды
HLT	F		Останов вычислений

[1] Шестнадцатеричный код операции.

Счетчик команд

Счетчик команд (СК) — неотъемлемый элемент устройства управления любой фон-неймановской ВМ. В соответствии с принципом программного управления команды программы хранятся в смежных ячейках памяти ВМ, то есть в ячейках со смежными адресами. Выполняются команды в естественной последовательности, в порядке их следования в программе (если изменение этой последовательности не предусмотрено программой). Из этого вывод: адрес очередной команды может быть вычислен путем добавления единицы к адресу текущей команды (в нашем примере любая команда занимает одну ячейку). Такое вычисление обеспечивает счетчик команд. Перед началом вычислений в СК заносится адрес ячейки основной памяти, где хранится первая команда. К моменту завершения очередной команды, не изменяющей естественную последовательность вычислений, содержимое СК увеличивается на единицу (по сигналу управления +1СК). Таким образом, адрес следующей команды программы всегда берется из счетчика команд. Для изменения естественного порядка вычислений (перехода в иную точку программы) вместо прибавления единицы нужно занести в СК адрес точки перехода.

Хотя термин «счетчик команд» считается общепринятым, его нельзя признать удачным из-за того, что он может создать неверное впечатление о задачах данного узла. По этой причине разработчики ВМ используют и иные названия, в частности *программный счетчик* (PC, Program Counter) или *указатель команды* (IP, Instruction Pointer). Последнее определение представляется наиболее удачным, поскольку точнее отражает назначение рассматриваемого узла УУ.

В заключение добавим, что в ряде ВМ роль СК выполняет не двоичный счетчик, а обычный регистр, увеличение содержимого которого обеспечивается специальной внешней схемой (схемой инкремента/декремента).

Регистр команды

Счетчик команд определяет лишь местоположение команды в памяти, но не хранит информации о ее содержании. Чтобы приступить к выполнению команды, ее необходимо извлечь из памяти и разместить в *регистре команды* (РК). Этот этап носит название *выборки команды*. Только с момента загрузки команды в РК она становится «видимой» для процессора. В РК команда хранится в течение всего времени ее выполнения. Как уже отмечалось ранее, любая команда содержит два поля: поле кода операции и поле адресной части. Учитывая это обстоятельство, регистр команды иногда рассматривают как совокупность двух регистров — *регистра кода операции* (РКОП) и *регистра адреса* (РА), в которых хранятся соответствующие составляющие команды.

Если команда занимает несколько смежных ячеек, то код операции всегда находится в том слове команды, которое извлекается из памяти первым. Это позволяет по коду операции определить, требуется ли считывание из памяти и загрузка в РК остальных слов команды. Собственно выполнение команды начинается только после занесения в РК ее полного кода.

Указатель стека

Указатель стека (УС) — это регистр, где хранится адрес вершины стека. В реальных вычислительных машинах стек реализуется в виде участка основной памяти, при этом вершина стека — это ячейка, куда при работе со стеком была произведена последняя по времени запись. Для хранения адреса такой ячейки и предназначен УС. При занесении информации в стек (операция *push*) содержимое УС с помощью сигнала –1УС сначала уменьшается на единицу, после чего используется в качестве адреса, по которому производится запись. Соответствующая ячейка становится новой вершиной стека. Считывание из стека (операция *pop*) происходит из ячейки, на которую указывает текущий адрес в УС, после чего содержимое указателя стека сигналом +1УС увеличивается на единицу, то есть вершина стека опускается, а считанное слово считается удаленным из стека. Хотя физически считанное слово и осталось в ячейке памяти, при следующей записи в стек оно будет заменено новой информацией.

Регистр адреса памяти

Регистр адреса памяти (РАП) предназначен для хранения исполнительного адреса ячейки основной памяти, вплоть до завершения операции (считывания или записи) с этой ячейкой. Наличие РАП позволяет компенсировать различия в быстродействии ОП и прочих устройств машины.

Регистр данных памяти

Регистр данных памяти (РДП) призван компенсировать разницу в быстродействии запоминающих устройств и устройств, выступающих в роли источников и потребителей информации. При чтении из памяти в РДП заносится считанное содержимое ячейки ОП. В случае записи информация, подлежащая сохранению в ячейке ОП, должна быть предварительно помещена в этот регистр. Собственно момент считывания и записи в ячейку определяется сигналами ЧтЗУ и ЗпЗУ соответственно.

Дешифратор кода операции

Дешифратор кода операции (ДКОП) преобразует код операции в форму, требуемую для работы микропрограммного автомата (МПА). С этих позиций ДКОП правильнее было бы назвать не дешифратором, а преобразователем кодов. В рассматриваемой гипотетической ВМ код операции преобразуется в унитарный код УнитК, в котором каждой команде (каждому КОП) соответствует отдельный бит.

Микропрограммный автомат

Микропрограммный автомат (МПА) правомочно считать центральным узлом устройства управления. Именно МПА формирует последовательность сигналов управления, в соответствии с которыми производятся все действия, необходимые для выборки команд из памяти и их выполнения. Исходной информацией для МПА служат: преобразованный в ДКОП код операции, состояние признаков (флагов), характеризующих результат предшествующих вычислений, а также внешние запросы на прерывание текущей программы.

Арифметико-логическое устройство

Это устройство, как следует из его названия, предназначено для арифметической и логической обработки данных. В машине, изображенной на рис. 3.1, оно содержит следующие узлы.

Операционный блок

Операционный блок (ОПБ) представляет собой ту часть АЛУ, которая, собственно, и выполняет арифметические и логические операции над поданными на вход операндами. Выбор конкретной операции (из возможного списка операций для данного ОПБ) определяется кодом операции команды. В нашей ВМ код операции поступает непосредственно из регистра команды. В реальных машинах КОП зачастую преобразуется в МПА в иную форму и уже из микропрограммного автомата поступает в АЛУ. Операционные блоки современных АЛУ строятся как комбинационные схемы, то есть они не обладают внутренней памятью, и до момента сохранения результата операнды должны присутствовать на входе блока.

Регистры операндов

Регистры РХ и РУ обеспечивают сохранение операндов на входе операционного блока вплоть до получения результата операции и его записи (в нашем случае в аккумулятор).

Регистр признаков

Регистр признаков (Рпрз) предназначен для фиксации и хранения признаков, характеризующих результат последней выполненной арифметической или логической операции. Такие признаки могут информировать о равенстве результата нулю, о знаке результата, о возникновении переноса из старшего разряда, переполнении разрядной сетки и т. д. Содержимое Рпрз обычно используется устройством управления при выполнении команд условных переходов. Под каждый из возможных признаков отводится один разряд Рпрз.

Формирование признаков осуществляется специальным узлом АЛУ, который может быть либо частью ОПБ, либо внешней схемой, располагаемой между операционным блоком и Рпрз.

Аккумулятор

Аккумулятор (Акк) — это регистр, на который возлагаются самые разнообразные функции. Так, в него предварительно загружается один из операндов, участвующих в арифметической или логической операции. В аккумуляторе может храниться результат предыдущей команды и в него же заносится результат очередной операции. Через аккумулятор зачастую производятся операции ввода и вывода.

Строго говоря, аккумулятор в равной мере можно отнести как к АЛУ, так и к УУ, а в ВМ с регистровой архитектурой его можно рассматривать как один из регистров общего назначения.

Основная память

Вне зависимости от типа используемых микросхем, основная память (ОП) представляет собой массив запоминающих элементов (ЗЭ), организованных в виде ячеек, способных хранить некую единицу данных, обычно один байт. Каждая ячейка имеет уникальный адрес. Ячейки ОП образуют матрицу, а выбор ячейки осуществляется путем подачи разрешающих сигналов на соответствующие строку и столбец этой матрицы. Выбор обеспечивается дешифратором адреса памяти, преобразующим поступивший из РАП адрес ячейки в разрешающие сигналы, подаваемые в горизонтальную и вертикальную линии, на пересечении которых расположена адресуемая ячейка. Поскольку для современных ВМ характерна значительная емкость ОП, приходится использовать несколько микросхем запоминающих устройств (ЗУ). В этих условиях процесс обращения к ячейке состоит из выбора нужной микросхемы (на основании старших разрядов адреса) и выбора ячейки внутри микросхемы (определяется младшими разрядами адреса). Первая часть процедуры производится внешними схемами, а вторая — внутри микросхем ЗУ.

Модуль ввода/вывода

Структура приведенного на рис. 3.1 *модуля ввода/вывода* (МВВ) обеспечивает только пояснение логики работы системы ввода/вывода ВМ. В реальных ВМ реализация модуля ввода/вывода может существенно отличаться от рассматриваемой. Задачей МВВ является обеспечение возможности подключения к вычислительной машине различных периферийных устройств (ПУ) и обмена информацией с ними. В рассматриваемом варианте МВВ состоит из дешифратора номера порта ввода/вывода, множества портов ввода и множества портов вывода.

Порты ввода и порты вывода

Портом называют схему, ответственную за передачу информации из периферийного устройства ввода в аккумулятор АЛУ (порт ввода) или из аккумулятора на периферийное устройство вывода (порт вывода). Схема обеспечивает электрическое и логическое сопряжение ВМ с подключенным к нему периферийным устройством.

Дешифратор номера порта ввода/вывода

В модуле ввода-вывода рассматриваемой ВМ предполагается, что каждое ПУ подключается к своему порту. Каждый порт имеет уникальный номер, который указывается в адресной части команд ввода/вывода. *Дешифратор номера порта ввода/вывода* (ДВВ) обеспечивает преобразование номера порта в сигнал, разрешающий операцию ввода или вывода в соответствующем порту. Непосредственно ввод (вывод) происходит при поступлении из МПА сигнала Вв или Выв.

Микрооперации и микропрограммы

Для пояснения логики функционирования ВМ ее целесообразно представить в виде совокупности узлов, связанных между собой коммуникационной сетью (рис. 3.2).

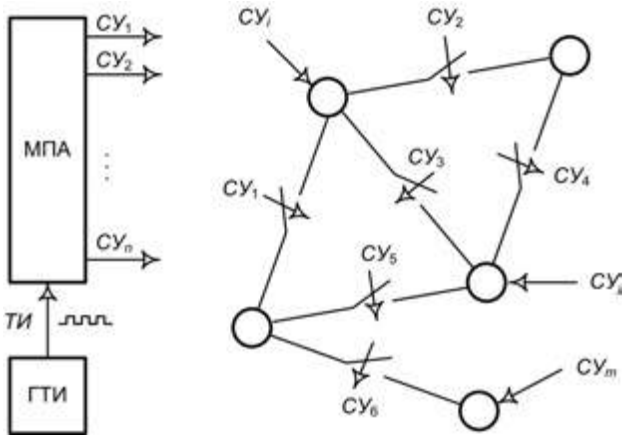


Рис. 3.2. Вычислительная машина с позиций микроопераций и сигналов управления

Функционирование вычислительной машины можно описать как последовательность пересылок информации между ее узлами и элементарных действий, выполняемых в узлах. Понятие узла здесь трактуется весьма широко: от регистра до АЛУ или основной памяти. Также широко следует понимать и термин «элементарное действие». Это может быть установка регистра в некоторое состояние или выполнение операции в АЛУ. Любое элементарное действие производится при поступлении из микропрограммного автомата УУ соответствующего сигнала управления (СУ). Возможная частота формирования сигналов на выходе автомата определяется синхронизирующими импульсами (ТИ), поступающими от генератора тактовых импульсов (ГТИ). Элементарные пересылки или преобразования информации, выполняемые в течение одного такта сигналов синхронизации, называются *микрооперациями*. В течение одного такта могут одновременно выполняться несколько микроопераций. Совокупность сигналов управления, порождающих микрооперации, выполняемые в одном такте, называют *микрокомандой*. Относительно сложные действия, осуществляемые вычислительной машиной в процессе ее работы,

реализуются как последовательность микроопераций и могут быть заданы последовательностью микрокоманд, называемой *микропрограммой*. Реализует микропрограмму (вырабатывает управляющие сигналы, задаваемые ее микрокомандами) *микропрограммный автомат* (МПА).

Способы записи микропрограмм

Для записи микропрограмм в компактной форме используются граф-схемы алгоритмов и языки микропрограммирования.

Граф-схемы алгоритмов

Граф-схема алгоритма (ГСА) имеет вид ориентированного графа. При построении графа оперируют пятью типами вершин (рис. 3.3).

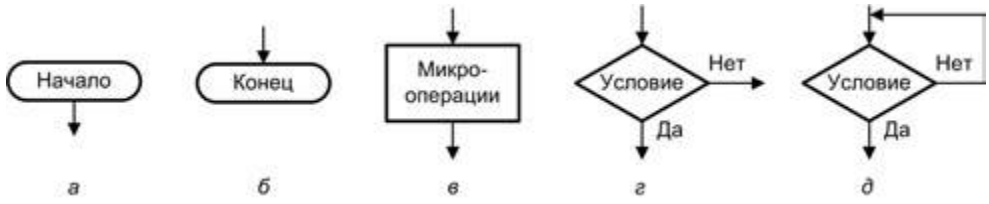


Рис. 3.3. Разновидности вершин граф-схемы алгоритма: *а* — начальная; *б* — конечная; *в* — операторная; *г* — условная; *д* — ждущая

Начальная вершина (см. рис. 3.3, *а*) определяет начало микропрограммы и не имеет входов. Конечная вершина (см. рис. 3.3, *б*) указывает конец микропрограммы, поэтому имеет только вход. В операторную вершину (см. рис. 3.3, *в*) вписывают микрооперации, выполняемые в течение одного машинного такта. С вершиной связаны один вход и один выход. Условная вершина (см. рис. 3.3, *г*) используется для ветвления вычислительного процесса. Она имеет один вход и два выхода, соответствующие положительному («Да») и отрицательному («Нет») исходам проверки условия, записанного в вершине. С помощью ждущей вершины (см. рис. 3.3, *д*) можно описывать ожидание в работе устройств. В этом случае выход «Да» соответствует снятию причины, вызвавшей ожидание.

Граф-схемы алгоритмов составляются в соответствии со следующими правилами.

1. ГСА должна содержать одну начальную, одну конечную и конечное множество операторных и условных вершин.
2. Каждый выход вершины ГСА соединяется только с одним входом.
3. Входы и выходы различных вершин соединяются дугами, направленными от выхода к входу.
4. Для любой вершины ГСА существует, по крайней мере, один путь из этой вершины к конечной вершине, проходящий через операторные и условные вершины в направлении соединяющих их дуг.
5. В каждой операторной вершине записываются микрооперации y, соответствующие одной микрокоманде Y. Сами микрокоманды указываются возле операторных вершин.

6. В каждой условной вершине записывается один из элементов множества логических условий x.
7. Начальной вершине ставится в соответствие фиктивный оператор y_0, а конечной — фиктивный оператор y_k.

На рис. 3.4 показан пример микропрограммы, записанной на языке ГСА.

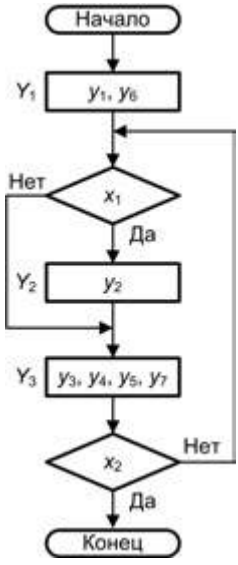


Рис. 3.4. Пример граф-схемы микропрограммы

В примере микрокоманда Y_1 инициирует микрооперации y_1 и y_6, микрокоманда Y_2 — микрооперацию y_2, а Y_3 — микрооперации y_3, y_4, y_5 и y_7.

Языки микропрограммирования

Языки микропрограммирования (ЯМП) обеспечивают описание функционирования ВМ в терминах микроопераций.

Если средства языка ориентированы на запись микропрограммы без привязки к конкретным средствам для их реализации, то такой ЯМП называют *языком функционального микропрограммирования*, а соответствующие микропрограммы — *функциональными микропрограммами* [15]. Подобные микропрограммы служат исходной формой для описания функционирования ВМ.

В случае, когда средства языка нацелены на описание микропрограмм, привязанных к конкретной реализующей их структуре, ЯМП называют *языком структурно-функционального микропрограммирования*.

В последующих разделах для описания функционирования ВМ будет использоваться язык микропрограммирования, предложенный в [20]. Ниже рассматриваются основные средства языка.

Описание слов, шин, регистров

Основным элементом данных, с которым оперирует микропрограмма, является *слово*.

Описание слова, шины (совокупности цепей, используемых для передачи слов), регистра состоит из названия (идентификатора) и разрядного указателя. Идентификатором может служить произвольная последовательность букв и цифр, начинающаяся с буквы. Разрядный указатель состоит из номеров старшего и младшего разрядов описываемого объекта, разделенных горизонтальной чертой (дефис) и заключенных в круглые скобки. Разрядный указатель может опускаться, если это не вызывает недоразумений (например, если объект уже был описан раньше). Ниже приводятся примеры описания слова, шины и регистра.

- Аисп(23–0) — описание слова, представляющего 24-разрядный исполнительный адрес $A_{\text{исп}} = a_{23}, a_{22}, ..., a_0$;
- ША(31–0) — описание 32-разрядной шины адреса;
- РК(31–0) — описание 32-разрядного регистра команды.

Если объект содержит несколько полей, то каждое из них может быть описано отдельно с сохранением общего идентификатора объекта. Так, пусть команда имеет длину 32 бита и состоит из 8-разрядного поля кода операции (КОП), 4-разрядного поля способа адресации (СА) и 20-разрядного поля адреса (А). Тогда описание отдельных полей регистра команды выглядит следующим образом: РК(31–24), РК(23–20), РК(19–0). Вместо номеров разрядов в разрядном указателе можно записывать наименование поля слова, и два первых поля регистра команды могут быть представлены так: РК(КОП), РК(СА). С другой стороны, описание объекта может быть представлено через описание его полей. Например, описание 32-разрядного регистра РПЗ для хранения чисел с плавающей запятой, где число состоит из трех полей: S (поле знака мантиссы, бит 31), Р (поле порядка, биты 30–23) и М (поле мантиссы, биты 22–0), может быть задано в виде РПЗ(31 • 30–23 • 22–0) или РПЗ(S • Р • М). Здесь точка обозначает операцию конкатенации (составления целого слова из его частей).

Описание памяти, слова памяти

Описание модуля памяти или отдельной его ячейки напоминает описание рассмотренных ранее объектов и отличается тем, что между идентификатором и разрядным указателем размещается адресный указатель, заключаемый в квадратные скобки. В случае описания модуля памяти он содержит два адреса, разделенных двоеточием (слева от двоеточия адрес первого, а справа — адрес последнего слова памяти). При описании ячейки адрес этой ячейки заключается в квадратные скобки. Ниже даны примеры описания модулей памяти и ячеек:

- ОЗУ[0000:1023](7–0) — описание модуля оперативного запоминающего устройства (ОЗУ) емкостью 1 Кбайт;
- ПЗУ[0000_{16}:$0FFF_{16}$](0–31) — описание модуля постоянного запоминающего устройства (ПЗУ) емкостью 8192 32-разрядных слова (адреса слов указаны в шестнадцатеричном коде, в каждом слове старший разряд имеет номер 0, а младший — 31);

- ОЗУ[211] — описание ячейки модуля ОЗУ, имеющей адрес 211.

При описании ячеек памяти допускается символическое представление адреса, например ОЗУ1[Аисп]. Если адрес ячейки дополнительно заключен в круглые скобки, это означает задание косвенного адреса. Так, в записи ОЗУ[(РАП)] указан косвенный адрес, значение которого содержится в регистре РАП.

Описание микроопераций

Здесь под микрооперацией понимается элементарная функциональная операция, выполняемая над словами под воздействием одного сигнала управления, который вырабатывается устройством управления ВМ. В зависимости от количества преобразуемых слов (операндов), различают одноместные, двухместные и трехместные микрооперации.

Описание микрооперации складывается из двух частей, разделяемых двоеточием

МЕТКА: МИКРООПЕРАТОР.

Микрооператор определяет содержимое производимого элементарного действия (микрооперации). Меткой служит обозначение сигнала управления, вызывающего выполнение данной микрооперации. Завершается метка двоеточием. Для повышения наглядности записей желательно, чтобы метка сигнала управления несла смысловую нагрузку. Например, для микроопераций выдачи информации идентификатор метки сигнала можно начинать с буквы «в», а для приема — с буквы «п». Метка для микрооперации пересылки в регистр РВ из регистра РА записывается в виде пРВвРА (прием в регистр РВ, выдача из регистра РА), или сокращенно РВ_РА. Как видим, метка сигнала управления отображает то же направление передачи данных, что и микрооператор, то есть справа налево.

Микрооператор по форме записи представляет собой оператор присваивания. Выражение справа от знака присваивания (:=) называется формулой микрооператора. Формула определяет вид преобразования, производимого микрооперацией, и местоположение преобразуемых операндов. Слева от знака присваивания в микрооператоре указывается приемник результата реализации формулы.

В соответствии с формулой микрооператора будем различать следующие классы микроопераций.

Микрооперация установки — присваивание слову значения константы.

Например, пРХ: РХ(S • М) := 0; пРС:РС(7–0) := 31_{10},

здесь запись 31_{10} означает, что речь идет о десятичной константе 31.

Микрооперации передачи — присваивание слову значения другого слова, в том числе с инверсией передаваемого слова.

Примером простого присваивания может служить микрооперация БпУп: СК := РА. Здесь микрооператор описывает занесение в счетчик команд содержимого регистра адреса (адресного поля регистра команды), то есть реализацию перехода в командах безусловного и условного перехода, что и отражает идентификатор сигнала управления.

Другие примеры микроопераций передачи:

пРУвРХ: РУ(15–0) := РХ(15–0); РевУХ: РУ(15–0) := РХ(0–15).

Первый микрооператор описывает пересылку 16-разрядного слова из регистра РХ в регистр РУ с сохранением расположения разрядов, а второй — с «разворотом» исходного слова.

Микрооперации передачи числа с плавающей запятой, имеющего поля знака S, порядка Р и мантиссы М, а также передачи знака с инвертированием, имеют вид:

пП32вП31: РгП32(S • Р • М) := РгП31 (S • Р • М); пЗИ: РгХ(S) := $\overline{\text{РгУ(S)}}$.

Если регистры связаны между собой не непосредственно, а через шину, которая используется многими источниками и приемниками данных, то передача слова между ними возможна при одновременном выполнении двух микроопераций, и описание принимает вид:

вРгВ: ША := РгВ, пРгА: РгА := ША

или

пРгА, вРгВ: РгА := ША := РгВ.

Здесь метки одновременно формируемых сигналов управления перечисляются через запятую и образуют микрокоманду.

Микрооперации составления слова — обеспечивают получение целого слова большой разрядности из нескольких малоразрядных слов.

Пусть в 16-разрядный регистр А нужно передать слово, старшие разряды которого содержатся в 8-разрядном регистре В, а младшие — в 8-разрядном регистре С. Соответствующую микрооперацию можно описать так:

пРгА: РгА(15–0) := РгВ(7–0) • РгС(7–0),

где точка (•) — знак операции составления.

Операция составления (конкатенации) позволяет присоединить значение слова, указанного справа от знака операции, к значению слова, расположенного слева от знака операции.

Микрооперации сдвига служат для изменения положения разрядов слова. Положение разрядов изменяется путем перемещения каждого разряда на несколько позиций влево или вправо.

Микрооперации сдвига слова в аккумуляторе, например, могут быть описаны в следующих формах:

- R2AK: AK(15–0) := PC(1–0) • AK(15–2) — сдвиг на два разряда вправо с введением в два старших освобождающихся разряда содержимого двух младших разрядов регистра PC;
- L1AK: AK(15–0) := AK(14–0) • 0 — сдвиг на один разряд влево с занесением в освобождающийся разряд нуля;
- R2AK(15–0): PC(15–0) := AK(15) • AK(15) • AK(15–2) — арифметический сдвиг слова вправо на два разряда с загрузкой в старшие освобождающиеся разряды знака.

Для сокращения записи микрооперации сдвига используются две процедуры:

- Rn(A) — удаление n младших правых разрядов из слова A, то есть сдвиг значения на n разрядов вправо;
- Ln(A) — удаление n старших левых разрядов из слова A, то есть сдвиг значения на n разрядов влево.

Использование этих процедур приводит к представлению ранее рассмотренных микрооператоров в форме:

- AK(15–0) := R2(PC(1–0) • AK);
- AK(15–0) := L1(AK • 0);
- PC (15–0) := R2(AK(s) • AK(s) • AK).

Микрооперация счета — обеспечивает изменение значения слова на единицу:

+1СК : СК := СК + 1.

Микрооперация сложения — служит для присваивания слову суммы слагаемых:

СлАД: РАП := ИР + РА.

Логические микрооперации — присваивают слову значение, полученное поразрядным применением функций И ($\wedge$), ИЛИ ($\vee$), исключающее ИЛИ ($\oplus$) к парам соответствующих разрядов операндов:

И: АК := РХ $\wedge$ РУ; М2: АК := РХ $\oplus$ РУ.

Микрооперация двоичного декодирования — состоит в преобразовании n-разрядного двоичного позиционного кода A в 2^n-разрядный унитарный код B. В унитарном коде только один разряд принимает единичное значение, а все остальные равны нулю. Номер разряда K, который принимает значение 1, определяется значением кода $A = a_{n-1}, a_{n-2}, ..., a_0$:

$$K = \sum_{i=0}^{n-1} a_i \times 2^i.$$

Принято следующее условное обозначение: B := DECOD (A).

Комментарии к микрооперациям

Микрооперации могут снабжаться произвольными комментариями. Комментарии записываются справа от микрооперации и заключаются в угловые скобки. Например:

+1СК := СК := СК + 1 <Увеличение содержимого СК на единицу>.

Совместимость микроопераций

Совместимостью называется свойство совокупности микроопераций, гарантирующее возможность их параллельного выполнения [15]. Различают функциональную и структурную совместимости. Пусть S_1, S_2, S_3, S_4 — подмножества слов из множества S. Тогда микрооперации $S_1 := \varphi_1(S_2)$ и $S_3 := \varphi_2(S_4)$ называются *функционально совместимыми*, если $S_1 \cap S2 = \varnothing$, то есть если микрооперации присваивают значения

разным словам. В функциональных микропрограммах, описывающих алгоритмы выполнения операций без учета структуры вычислительной машины, одновременно могут выполняться только функционально совместимые микрооперации.

Структура ВМ может внести дополнительные ограничения на возможность параллельного выполнения микроопераций. Микрооперации называются *структурно несовместимыми*, если из-за ограничений, обусловленных структурой ВМ, они не могут выполняться параллельно. Обычно структурная несовместимость связана с использованием микрооперациями одного и того же оборудования.

Цикл команды

Программа в фон-неймановской ЭВМ реализуется центральным процессором (ЦП) посредством последовательного исполнения образующих эту программу команд. Действия, требуемые для выборки (извлечения из основной памяти) и выполнения команды, называют *циклом команды*. В общем случае цикл команды включает в себя несколько составляющих (этапов):

- выборку команды (ВК);
- декодирование команды (ДК);
- вычисление исполнительных адресов (ВА);
- выборку операндов (ВО);
- исполнение операции (ИО);
- запись результата (ЗР);
- формирование адреса следующей команды (ФАСК).

Перечисленные этапы выполнения команды в дальнейшем будем называть *стандартным циклом команды*. Отметим, что не все из этапов присутствуют при выполнении любой команды (зависит от типа команды), тем не менее этапы выборки, декодирования, исполнения и формирования адреса следующей команды имеют место всегда.

В определенных ситуациях возможны еще два этапа:

- косвенная адресация (КА);
- реакция на прерывание (РП).

Стандартный цикл команды

Кратко охарактеризуем каждый из вышеперечисленных этапов стандартного цикла команды. При изучении данного материала следует учитывать, что приводимое описание лишь дает представление о сущности каждого из этапов. В то же время распределение функций по разным этапам цикла команды и последовательность выполнения этапов в реальных ВМ могут отличаться от излагаемых.

Этап выборки команды

Цикл любой команды начинается с того, что центральный процессор извлекает команду из памяти, используя адрес, хранящийся в счетчике команд (СК). Двоичный код команды помещается в регистр команды (РК) и с этого момента становится

«видимым» для процессора. Без учета промежуточных пересылок и сигналов управления это можно описать следующим образом: РК := ОП[(СК)].

Приведенная запись охватывает весь этап выборки, если длина команды совпадает с разрядностью ячейки памяти. В то же время система команд многих ВМ предполагает несколько форматов команд, причем в разных форматах команда может занимать 1, 2 или более ячеек, а этап выборки команды можно считать завершенным лишь после того, как в РК будет помещен полный код команды. Информация о фактической длине команды содержится в полях кода операции и способа адресации. Обычно эти поля располагают в первом слове кода команды, и для выяснения необходимости продолжения процесса выборки необходимо предварительное декодирование их содержимого.

Такое декодирование может быть произведено после того, как первое слово кода команды окажется в РК. В случае многословного формата команды процесс выборки продолжается вплоть до занесения в РК всех слов команды. Например, для 16-разрядной команды, занимающей две 8-разрядных ячейки памяти, выборку можно описать так:

```
ПСтРК: РК(15–8) := ОП[(СК)];      <Занесение в РК старшего байта команды>
+1СК: СК := СК + 1;
ПМлРК: РК(7–0) := ОП[(СК)].       <Занесение в РК младшего байта команды>
```

Этап декодирования команды

После выборки команды она должна быть декодирована, для чего ЦП расшифровывает находящийся в РК код команды. В результате декодирования выясняются следующие моменты:

- находится ли в РК полный код команды или требуется дозагрузка остальных слов команды;
- какие последующие действия нужны для выполнения данной команды;
- если команда использует операнды, то откуда они должны быть взяты (номер регистра или адрес ячейки основной памяти);
- если команда формирует результат, то куда этот результат должен быть направлен.

Ответы на два первых вопроса дает расшифровка кода операции, результатом которой может быть унитарный код, где каждый разряд соответствует одной из команд, что можно описать в виде УнитК := DECOD (КОП). На практике вместо унитарного кода могут встретиться самые разнообразные формы представления результатов декодирования, например адрес ячейки специальной управляющей памяти, где хранится первая микрокоманда микропрограммы для реализации указанной в команде операции.

Полное выяснение всех аспектов команды, помимо расшифровки кода операции, требует также анализа адресной части команды, включая поле способа адресации.

По результатам декодирования производится подготовка электронных схем ВМ к выполнению предписанных командой действий.

Этап вычисления исполнительных адресов

Как уже отмечалось ранее, для физического доступа к основной памяти необходимо подать на ее адресные входы исполнительный адрес ячейки. В то же время в адресной части команды обычно указывается лишь адресный код, который, в общем случае, отличается от исполнительного адреса. Поэтому для последующего исполнения команды адресный код предварительно должен быть преобразован в соответствующий исполнительный адрес. Такое преобразование и составляет содержание рассматриваемого этапа цикла команды. При преобразовании руководствуются способом адресации, указанным в одноименном поле кода команды. Так, в случае индексной адресации для получения исполнительного адреса производится суммирование содержимого адресной части команды и содержимого индексного регистра.

Этап выборки операндов

Данный этап характерен лишь для команд обработки операндов. Вычисленные на предыдущем этапе исполнительные адреса используются для считывания операндов из памяти и занесения в определенные регистры процессора. Например, в случае арифметической команды операнд после извлечения из памяти может быть загружен во входной регистр АЛУ. Однако чаще операнды предварительно заносятся в специальные вспомогательные регистры процессора, а их пересылка на вход АЛУ происходит на этапе исполнения операции.

Этап исполнения операции

На этом этапе реализуется указанная в команде операция. В силу различия сущности каждой из команд ВМ, содержание этого этапа также сугубо индивидуально. Этапы исполнения типовых команд будут рассмотрены ниже на примере команд гипотетической вычислительной машины, приведенной на рис. 3.1.

Этап записи результата

Этап записи результата присутствует в цикле тех команд, которые предполагают занесение результата в регистр или ячейку основной памяти. Фактически его можно считать частью этапа исполнения, особенно для тех команд, которые помещают результат сразу в несколько мест.

Этап формирования адреса следующей команды

Для фон-неймановских машин характерно размещение соседних команд программы в смежных ячейках памяти. Если извлеченная команда не нарушает естественного порядка выполнения программы, для вычисления адреса следующей выполняемой команды достаточно увеличить содержимое счетчика команд на длину текущей команды, представленную количеством занимаемых кодом команды ячеек памяти. Для однословной команды это описывается микрооперацией: +1СК: СК := СК + 1.

Длина команды, а также то, способна ли она изменить естественный порядок выполнения команд программы, выясняются на этапе декодирования. Если извлеченная команда способна изменить последовательность выполнения программы (команда условного или безусловного перехода, вызова процедуры и т. п.), процесс

формирования адреса следующей команды сводится к занесению в СК адресной части кода команды и фактически происходит уже на этапе исполнения.

Описание стандартных циклов команды для гипотетической машины

Для иллюстрации содержания стандартного цикла команд обратимся к гипотетической ВМ (см. рис. 3.1). Микропрограммы отдельных этапов исполнения ее команд приведены в табл. 3.2. В таблице также указаны номера тактовых периодов, в течение которых выполняется каждая микрокоманда.

Таблица 3.2. Микропрограммы отдельных этапов циклов команд (гипотетическая ВМ)

Этап	Команда		Номер такта	Микропрограмма
	КОп	АЧ		
ВК			T0	РАП_СК: РАП := СК, ЧтЗУ: РДП := ОП[(РАП)]
ВК+ДК			T1	РК_РДП: РК := РДП, МПА := УнитК := DECOD (РК (КОП))
ВО	LDA	ADR	T2	РАП_РА: РАП := РК(РА); ЧтЗУ: РДП := ОП[(РАП)]
ИО			T3	Акк_РДП: Акк := РДП
ИО	STA	ADR	T2	РАП_РА: РАП := РК(РА); РДП_Акк: РДП := Акк
ИО			T3	ЗпЗУ: ОП[(РАП)] := РДП
ВО	ADD	ADR	T2	РАП_РА: РАП := РК(РА), ЧтЗУ: РДП := ОП[(РАП)]
ИО			T3	РХ_РДП: РХ := РДП, РУ_Акк: РУ := Акк, ОпБ := РУ + РХ, Рпрз := Признаки
ИО			T4	Акк_ОпБ: Акк := ОпБ
ВО	SUB	ADR	T2	РАП_РА: РАП := РК(РА), ЧтЗУ: РДП := ОП[(РАП)]
ИО			T3	РХ_РДП: РХ := РДП, РУ_Акк: РУ := Акк, ОпБ := РУ – РХ, Рпрз: = Признаки
ИО			T4	Акк_ОпБ: Акк := ОпБ
ИО	INP	IPRT	T2	ДВВ_РА: ДВВ := РК(РА)
ИО			T3	Вв: Акк := Порт ввода IPRT
ИО	OUT	OPRT	T2	ДВВ_РА: ДВВ := РК(РА)
ИО			T3	Выв: Порт вывода OPRT := Акк
ИО	JMP	ADR	T2	БПУП: СК := РА
ИО	BRZ	ADR	T2	If Z=1 then БПУП: СК := РА else +1СК: СК := СК+1
ИО	HLT		T2	ОСТ:
ФАСК			T4 T5	+1СК: СК := СК+1; <Для команд LDA, STA, INP, OUT> +1СК: СК := СК+1; <Для команд ADD, SUB >

В силу важности рассматриваемого вопроса ниже дается словесное описание процессов, представленных микропрограммами табл. 3.2. Напомним, что все сигналы управления и управляющие коды формируются микропрограммным автоматом.

Этап выборки любой команды ВМ. На этом этапе происходит извлечение двоичного кода команды из ячейки основной памяти и его занесение в регистр команды. Этап реализуется в двух начальных тактах цикла команды (T0 и T1):

- Такт T0 — вырабатывается сигнал управления РАП_СК, инициирующий пересылку содержимого счетчика команд (СК) в регистр адреса памяти (РАП), после чего по сигналу ЧтЗУ содержимое ячейки, выбранной дешифратором адреса памяти (код команды), переписывается в регистр данных памяти (РДП).
- Такт T1 — формируется сигнал РК_РДП, по которому содержимое РДП передается в РК, при этом поле РКОП заполняется кодом операции, а поле РА — адресной частью команды.

Этап декодирования любой команды ВМ. Сразу же после размещения кода операции в РК производится его декодирование, в результате которого активным становится выход дешифратора кода операции (ДКОП), соответствующий коду операции:

- Такт T1 — ДКОП непосредственно подключен к соответствующему полю регистра команды, поэтому специальный сигнал управления не нужен, а декодирование происходит сразу же после заполнения РК, то есть параллельно второй фазе этапа ВК.

Этап вычисления исполнительных адресов. В гипотетической ВМ предусмотрена только прямая адресация, поэтому этап вычисления исполнительных адресов отсутствует.

Этап выборки операндов. В рассматриваемой системе команд этап выборки операндов имеется только в командах LDA, ADD и SUB.

Этапы команды LDA ADR. Команда обеспечивает занесение в аккумулятор содержимого ячейки ОП с адресом ADR:

- Такт T2 (этап ВО) — по сигналу РАП_РА содержимое РА (адресная часть команды) пересылается в РАП, после чего по сигналу ЧтЗУ содержимое ячейки с адресом ADR заносится в РДП.
- Такт T3 (этап ИО) — по сигналу Акк_РДП операнд из РДП пересылается в аккумулятор.

Этап исполнения операции для команды STA ADR. Команда обеспечивает сохранение содержимого аккумулятора в ячейке ОП с адресом ADR:

- Такт T2 — сигналом РАП_РА адресная часть команды (ADR) из РА пересылается в РАП, одновременно с этим содержимое аккумулятора по сигналу РДП_Акк заносится в РДП.
- Такт T3 — по сигналу ЗпЗУ происходит физическая запись содержимого РДП в ячейку ОП, на которую указывает адрес, находящийся в РАП.

Этапы команды ADD ADR. Команда обеспечивает суммирование текущего содержимого аккумулятора с содержимым ячейки ОП, имеющей адрес ADR. Результат сложения остается в аккумуляторе. Одновременно с этим в АЛУ формируются признаки результата, которые запоминаются в регистре признаков:

- Такт Т2 (этап ВО) — вырабатывается сигнал РАП_РА, и содержимое РА поступает в РАП. В том же такте по сигналу ЧтЗУ второе слагаемое из ячейки с адресом ADR (первое слагаемое берется из аккумулятора) заносится в РДП.
- Такт Т3 (этап ИО) — сигнал управления РХ_РДП вызывает пересылку второго операнда из РДП в регистр РХ арифметико-логического устройства. Одновременно с этим, сигналом РУ_Акк первый операнд из аккумулятора переписывается в РУ. Операционный блок выполняет над данными, расположенными в РХ и РУ, операцию, заданную в коде операции команды (в нашем случае — сложение), а также формирует признаки, характеризующие этот результат. Признаки автоматически заносятся в регистр признаков Рпрз.
- Такт Т4 (этап ИО) — по сигналу Акк_ОпБ информация с выхода ОпБ сохраняется в аккумуляторе.

Этапы команды SUB ADR**.** Команда обеспечивает вычитание из текущего содержимого аккумулятора содержимого ячейки ОП, имеющей адрес ADR. Результат вычитания остается в аккумуляторе. Как и при сложении, формируются и запоминаются признаки результата. Содержание подобно описанию для команды ADD ADR, за исключением действия, выполняемого в ОПБ в такте Т3 (вычитание вместо сложения).

Этап исполнения для команды INP IPRT**.** Команда обеспечивает занесение в аккумулятор информации из периферийного устройства (ПУ), подключенного к порту ввода с номером IPRT:

- Такт Т2 (этап ИО) — вырабатывается управляющий сигнал ДВВ_РА, по которому адресная часть команды — номер порта ввода — из РА поступает на вход дешифратора номера порта ввода/вывода.
- Такт Т3 (этап ИО) — по сигналу Вв информация из ПУ, подключенного к выбранному дешифратором порту ввода, заносится в аккумулятор.

Этап исполнения для команды OUT OPRT**.** Команда обеспечивает вывод содержимого аккумулятора на ПУ, подключенное к порту вывода с номером OPRT:

- Такт Т2 (этап ИО) — по сигналу ДВВ_РА адресная часть команды — номер порта вывода — из РА подается на вход дешифратора номера порта ввода/вывода.
- Такт Т3 (этап ИО) — сигналом Выв содержимое аккумулятора через выбранный с помощью ДВВ порт вывода передается на подключенное ПУ.

Этап исполнения операции для команды JMP ADR**.** Команда обеспечивает безусловный переход к команде, расположенной в ячейке ОП с адресом ADR:

- Такт Т2 — по сигналу БПУП адресная часть команды (ADR) заносится в счетчик команд, тем самым фактически реализуется этап формирования адреса следующей команды.

Этап исполнения операции для команды BRZ ADR**.** Команда анализирует хранящийся в Рпрз признак (флаг) нулевого результата, выработанный в АЛУ на предыдущем этапе вычислений, и формирует адрес следующей команды в зависимости от состояния этого признака:

- Такт Т2 — при нулевом значении признака (условие перехода не выполнено) естественный порядок выполнения программы не нарушается, и адрес следующей

команды формируется обычным образом, путем увеличения содержимого СК на единицу; при единичном значении признака (условие перехода выполнено) в СК заносится содержимое РА. Напомним, что в РА находится адресная часть извлеченной из ОП команды перехода, то есть адрес точки перехода (ADR).

Этап исполнения для команды HLT**.** Команда приводит к завершению вычислений. При этом вырабатывается сигнал OCT, нужный для того, чтобы известить операционную систему о завершении текущей программы.

Этап формирования адреса следующей команды. Для формирования адреса следующей команды (если текущая команда не меняет естественной последовательности вычислений) достаточно увеличить содержимое счетчика команд на единицу.

Машинный цикл с косвенной адресацией

Многие команды предполагают чтение операндов из памяти или запись в память. В простейшем случае в адресном поле таких команд явно указывается исполнительный адрес соответствующей ячейки ОП. Однако часто используется и другой способ указания адреса, когда адрес операнда хранится в какой-то ячейке памяти, а в команде указывается адрес такой ячейки. Как уже отмечалось ранее, подобный прием называется *косвенной адресацией*. Чтобы прочитать или записать операнд, сначала нужно извлечь из памяти его адрес и только после этого произвести нужное действие (чтение или запись операнда), иными словами, требуется выполнить два обращения к памяти. Это, естественно, отражается и на цикле команды, в котором появляется косвенная адресация. Этап косвенной адресации можно отнести к этапу вычисления адресов операндов, поскольку его сущность сводится к определению исполнительного адреса операнда.

Применительно к вычислительной машине, приведенной на рис. 3.1, при косвенной адресации должны обеспечиваться следующие микрооперации:

РАП_РА: РАП := РК(РА), ЧтЗУ: РДП := ОП[(РАП)];
РА_РДП: РК(РА) := РДП <Заметим, что в нашей ВМ такого сигнала нет>.

Контрольные вопросы

1. Какую функцию выполняет счетчик команд и какой должна быть его разрядность?
2. Какое из полей регистра команд должно быть заполнено в первую очередь?
3. Какой адрес должен быть занесен в указатель стека при его инициализации?
4. Какими средствами компенсируется различие в быстродействии процессора и основной памяти?
5. На основании какой информации микропрограммный автомат формирует сигналы управления?
6. Можно ли считать наличие регистров операндов обязательным условием работы любого операционного блока?
7. Каким образом используется информация, хранящаяся в регистре признаков?
8. С каким понятием можно ассоциировать сигнал управления?

9. В чем состоит различие между микрокомандой и микрооперацией?
10. Какие существуют способы записи микропрограмм?
11. Перечислите основные правила составления граф-схемы алгоритма.
12. Как в предложенном языке микропрограммирования описывается разрядность шины?
13. Какие варианты описания слова памяти допускает язык микропрограммирования?
14. Описание каких видов микроопераций допускает рассмотренный в учебнике язык микропрограммирования?
15. Что подразумевает понятие «совместимость микроопераций»?
16. Какие из этапов цикла команды являются обязательными для всех команд?
17. Какие узлы ВМ участвуют в реализации этапа выборки команды?

Глава 4

Устройства управления

Данная глава детализирует аспекты структурной организации и функционирования устройств управления вычислительной машины, не рассмотренные в главах 1, 3.

Функции и структура устройства управления

Устройство управления (УУ) вычислительной машины реализует функции управления ходом вычислительного процесса, обеспечивая автоматическое выполнение команд программы.

Процесс выполнения программы в ВМ представляет собой последовательность машинных циклов отдельных команд. Напомним основные целевые функции устройства управления в ходе типового машинного цикла:

- выборка и декодирование команды,
- вычисление исполнительных адресов и выборка операндов,
- исполнение операции,
- формирование адреса следующей команды.

В свою очередь, каждая функция (этап цикла) реализуется последовательностью элементарных действий в узлах. Такие элементарные действия, выполняемые в течение одного такта сигналов синхронизации, называются *микрооперациями* (МО). Совокупность сигналов управления, вызывающих одновременно выполняемые микрооперации, образует *микрокоманду* (МК). Последовательность микрокоманд, определяющую содержание и порядок реализации цикла команды, принято называть *микропрограммой*. Сигналы управления генерируются центральным узлом устройством управления — *микропрограммным автоматом* (МПА). Название отражает то, что МПА определяет микропрограмму как последовательность выполнения микроопераций.

Микропрограммы реализации перечисленных целевых функций инициируются *задающим оборудованием*, то есть собственно УУ.

Выполняются микропрограммы *исполнительным оборудованием* вычислительной машины. Основной частью исполнительного оборудования является операционное устройство процессора.

В обобщенной структуре УУ (рис. 4.1) можно выделить две части: управляющую и адресную. Управляющая часть УУ предназначена для управления работой исполнительного оборудования ВМ. Адресная часть УУ обеспечивает формирование адресов команд и исполнительных адресов операндов (в основной памяти), то есть относится к исполнительному оборудованию ВМ.

В состав управляющей части УУ входят: регистр команды (РК), дешифратор кода операции (ДКОП), микропрограммный автомат (МПА) и узел прерывания программ (УПП).

Регистр команды предназначен для приема очередной команды из запоминающего устройства и ее хранения в течение всего цикла команды. В соответствии со структурой типовой команды он содержит операционную часть для хранения кода операции (РКОП) и адресную часть (РА), представленную адресным кодом ($А_к$) и кодом способа адресации (СА).

Дешифратор кода операции обеспечивает преобразование кода операции в форму, обеспечивающую эффективный запуск микропрограммного автомата.

Микропрограммный автомат на основании результатов декодирования кода операции (и кода способа адресации) вырабатывает определенную последовательность микрокоманд, вызывающих выполнение всех целевых функций УУ.

Узел прерываний программ позволяет реагировать на различные ситуации, связанные как с выполнением рабочих программ, так и с состоянием ВМ.

Адресная часть УУ включает в себя: операционный узел устройства управления (ОПУУ), счетчик команд (СК), указатель стека (УС) и регистр адреса памяти (РАП).

Операционный узел устройства управления, называемый иначе узлом индексной арифметики или узлом адресной арифметики, обрабатывает адресные части команд, формируя исполнительные адреса операндов, а также подготавливает адрес следующей команды при выполнении команд перехода. Состав ОПУУ может быть аналогичен составу основного операционного устройства ВМ (иногда в простейших ВМ с целью экономии затрат на оборудование ОПУУ совмещается с основным операционным устройством).

Указатель стека хранит адрес вершины стека, а его содержимое используется при выполнении операций со стеком.

Регистр адреса памяти используется для хранения исполнительных адресов операндов, а *счетчик команд* — для выработки и хранения адресов команд. Содержимое РАП и СК посылается в регистр адреса основной памяти (ОП) для выборки операндов и команд соответственно.

В состав УУ могут также входить дополнительные узлы, в частности узел организации прямого доступа к памяти. Этот узел обычно реализуется в виде

самостоятельного устройства — контроллера прямого доступа к памяти (КПДП). КПДП обеспечивает совмещение во времени работы операционного устройства с процессом обмена информацией между ОП и другими устройствами ВМ, тем самым повышая общую производительность машины.

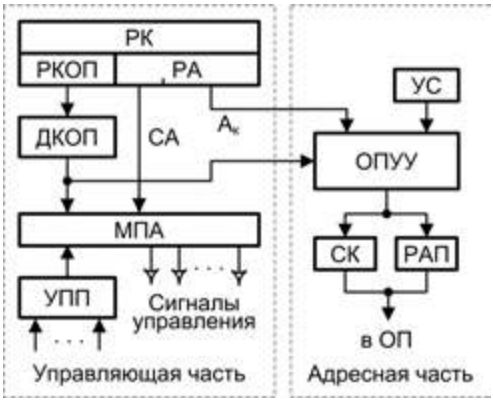


Рис. 4.1. Обобщенная структура устройства управления

Довольно часто регистры различных узлов УУ объединяют в отдельный узел управляющих (специальных) регистров устройства управления.

Микропрограммный автомат

Как уже отмечалось в главе 3, функционирование ВМ обеспечивают сигналы управления (СУ), формируемые устройством управления, конкретно – микропрограммным автоматом (МПА). На рис. 4.2 представлена информационная модель МПА [143].

Рис. 4.2. Информационная модель микропрограммного автомата

На вход микропрограммного автомата поступают:

- *код операции*, по которому МПА определяет, какие микропрограммы нужно выполнить для реализации данной команды;
- *тактовые импульсы*, задающие разрешенные моменты формирования сигналов управления;
- *признаки* результата предшествующей арифметической или логической операции (анализируются в микропрограммах команд, реализация которых зависит от выполнения или невыполнения какого-либо условия, представленного одним из признаков);
- *сигналы из системной шины*, поступающие от запоминающих устройств или устройств ввода/вывода и извещающие о событиях в этих устройствах (запросах прерывания, поступлении подтверждений выполнения каких-либо действий и т. п.).

На выходе МПА формируются:

- *внутренние сигналы управления*, циркулирующие внутри центрального процессора и предназначенные для его внутренних узлов;
- *сигналы в системную шину*, предназначенные для управления памятью и системой ввода/вывода.

Обобщенно структуру МПА можно представить в виде, показанном на рис. 4.3.

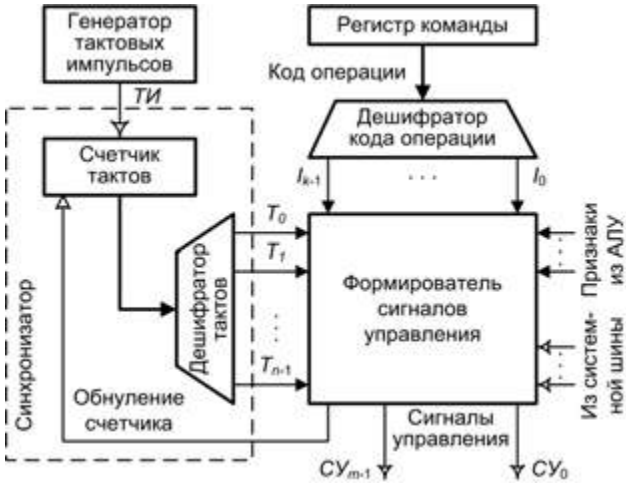


Рис. 4.3. Обобщенная структура микропрограммного автомата

Сигналы управления вырабатываются формирователем сигналов управления (ФСУ). Каждый СУ «привязан» к определенному периоду тактовых импульсов (ТИ). Отсчет периодов ведется от начала цикла команды с помощью синхронизатора, состоящего из счетчика тактов и дешифратора тактов. Очередной тактовый импульс увеличивает содержимое счетчика тактов на единицу. С выхода дешифратора тактов снимаются сигналы номеров тактовых периодов: T_0, ..., T_{n-1}. При такой

организации в МПА предусмотрена обратная связь для обнуления счетчика тактов по окончании цикла команды. Нулевое состояние счетчика соответствует тактовому периоду T_0, то есть началу цикла очередной команды.

Цикл команды начинается с микропрограммы выборки, обеспечивающей извлечение кода команды из памяти и его занесение в регистр команды (РК). Этот этап одинаков для любой команды и выполняется с помощью стандартной последовательности СУ. Продолжение цикла команды, находящейся в РК, зависит от ее кода операции. Этот код с помощью дешифратора преобразуется в форму $I_{k-1}...I_0$, требуемую для работы формирователя сигналов управления. В ФСУ происходит выбор нужной микропрограммы и выработка сигналов управления, определяемых этой микропрограммой. На выбор микропрограммы и ход ее выполнения могут влиять как состояние осведомительных сигналов (признаков), отражающих ход вычислений, так и сигналы о внешних событиях в памяти и системе ввода/вывода, поступающие из системной шины. Выработка сигналов управления выражается в переводе соответствующих линий управления в активное состояние.

Наибольшее распространение получили два варианта микропрограммных автоматов:

- с аппаратной или «жесткой» логикой;
- с программируемой логикой (хранимой в памяти логикой).

Различие между данными вариантами, по сути, сводится к способу реализации формирователя сигналов управления. В обоих случаях при проектировании ФСУ сигналы управления представляются двоичными цифрами 1 (активное состояние СУ) и 0 (отсутствие СУ).

Микропрограммный автомат с аппаратной логикой

При проектировании МПА с аппаратной логикой каждый используемый в ВМ сигнал управления описывается логическим выражением, единичное значение которого означает необходимость формирования данного СУ. После совместной минимизации выражений для всех СУ синтезируется комбинационная схема формирователя сигналов управления.

В качестве примера спроектируем возможную схему ФСУ для гипотетической ВМ, воспользовавшись для этого табл. 4.1, где перечислены все сигналы управления с указанием команд и тактов, когда эти СУ должны формироваться.

На основе приведенных данных можно составить следующую систему логических выражений:

$РАП\_СК = ВК \cdot T_0$

$ЧтЗУ = ВК \cdot T_0 \vee LDA \cdot T_2 \vee (ADD \vee SUB) \cdot T_3$

$ЗпЗУ = STA \cdot T_3$

$Вв = INP \cdot T_3$

$Выв = OUT \cdot T_3$

$+1СК = BRZ\cdot \overline{Z} \cdot T_2 \vee HLT\cdot T_3 \vee (LDA \vee STA \vee INP \vee OUT) \cdot T_4 \vee (ADD \vee SUB) \cdot T_5$

$БПУП = (JMP \vee BRZ\cdot Z) \cdot T_2$

$РАП\_РА = (LDA \vee STA \vee ADD \vee SUB) \cdot T_2$

$ДВВ\_РА = (INP \vee OUT) \cdot T_2$

$РДП\_Акк = STA\cdot T_2$

$РК\_РДП = ВК\cdot T_1$

$Акк\_РДП = LDA\cdot T_3$

$РХ\_РДП = (ADD \vee SUB) \cdot T_3$

$РУ\_Акк = (ADD \vee SUB) \cdot T_3$

$Акк\_ОпБ = (ADD \vee SUB) \cdot T_4$

$ОСТ = HLT\cdot T_2$

Таблица 4.1. Описание СУ и моментов их формирования

	ВК	**LDA**	**STA**	**ADD**	**SUB**	**INP**	**OUT**	**JMP**	**BRZ**	**HLT**
РАП_СК	T_0									
ЧтЗУ	T_0	T_2		T_3	T_3					
ЗпЗУ			T_3							
Вв						T_3				
Выв							T_3			
+1СК		T_4	T_4	T_5	T_5	T_4	T_4		$T_2\cdot \overline{Z}$	T_3
БПУП								T_2	$T_2\cdot Z$	
РАП_РА		T_2	T_2	T_2	T_2					
ДВВ_РА						T_2	T_2			
РДП_Акк			T_2							
РК_РДП	T_1									
Акк_РДП		T_3								
РХ_РДП				T_3	T_3					
РУ_Акк				T_3	T_3					
Акк_ОпБ				T_4	T_4					
ОСТ										T_2

Соответствующая этой системе схема ФСУ показана на рис. 4.4.

Будучи комбинационной схемой, ФСУ позволяет обеспечить очень высокую частоту формирования сигналов управления, что является основным достоинством МПА с аппаратной логикой. В то же время с возрастанием объема и сложности системы команд увеличивается количество сигналов управления и усложняется схема ФСУ, следствием чего становится снижение ее быстродействия. По этой причине МПА с аппаратной логикой характерны, главным образом, для вычислительных

машин с RISC-архитектурой, со сравнительно простой системой команд и ограничениями на форматы команд и способы адресации.

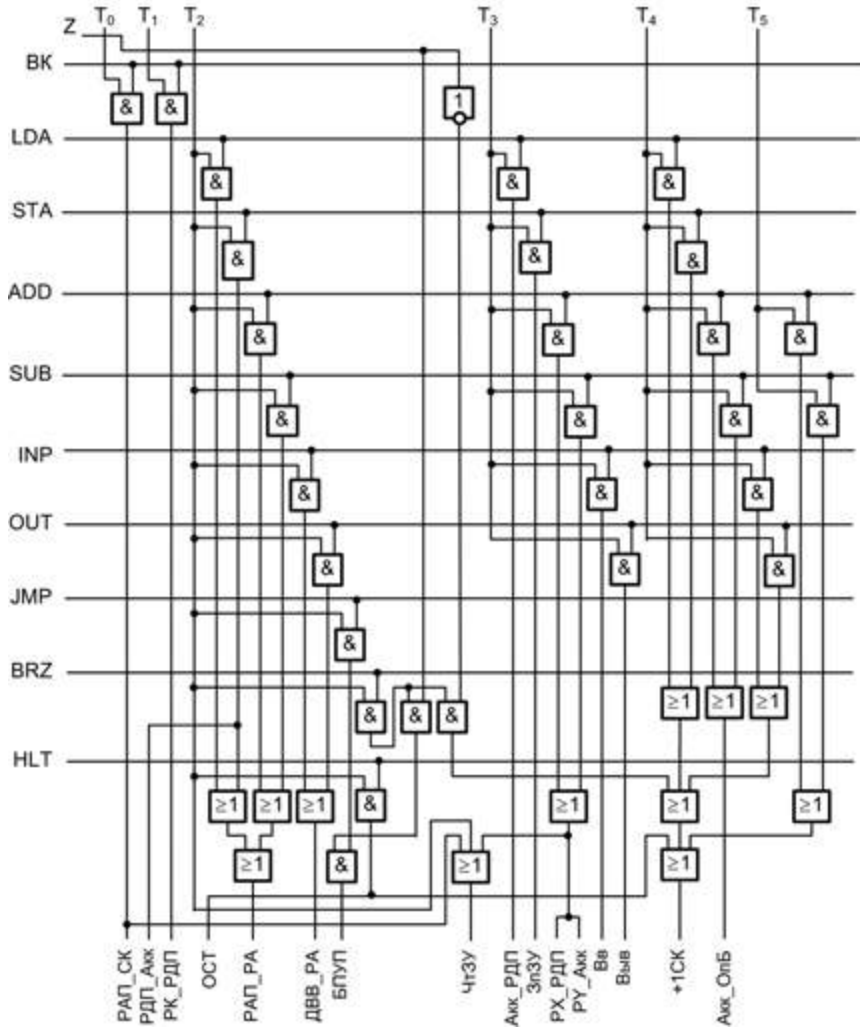


Рис. 4.4. Аппаратная реализация формирователя сигналов управления для гипотетической ВМ

Микропрограммный автомат с программируемой логикой

Проблему сложности МПА с аппаратной логикой позволяет разрешить принципиально иной подход, выражающийся в способе формирования СУ, то есть в структуре формирователя сигналов управления. Он был предложен британским ученым

М. Уилксом в начале 50-х годов [163][1]. В основе идеи лежит тот факт, что для инициирования любой микрооперации достаточно сформировать соответствующий СУ на соответствующей линии управления, то есть перевести такую линию в активное состояние. Это может быть представлено с помощью двоичных цифр 1 (активное состояние — есть СУ) и 0 (пассивное состояние — нет СУ). Сигналы управления в МПА с программируемой логикой представляются с помощью управляющих слов — *микрокоманд* (МК). Микрокоманда соответствует одному такту работы ВМ и определяет, какие СУ должны быть сформированы в данном такте. В простейшем случае каждый из возможных СУ может быть представлен в микрокоманде одним разрядом, а установка *i*-го разряда в 1 означает необходимость формирования *i*-го СУ. Последовательность МК, по тактам описывающая выполнение определенного этапа цикла команды, образует *микропрограмму* (МП).

Микропрограммы составляются для каждого этапа машинного цикла каждой команды ВМ и размещаются в специальном запоминающем устройстве, называемом управляющей памятью (УПМ) или памятью микропрограмм. Процесс формирования сигналов управления сводится к последовательному (с каждым тактовым импульсом) извлечению из управляющей памяти очередной МК микропрограммы. Содержащаяся в МК информация интерпретируется как набор СУ. В терминологии на английском языке микропрограмму часто называют firmware, подчеркивая тот факт, что это нечто среднее между аппаратурой (hardware) и программным обеспечением (software).

Возможное размещение микропрограмм в управляющей памяти показано на рис. 4.5. Основное содержимое УПМ составляют микропрограммы этапа исполнения для каждой из команд, входящих в систему команд ВМ. Некоторые из хранящихся в УПМ микропрограмм являются общими для всех команд, например микропрограмма этапа выборки команды или этапа формирования адреса следующей команды.

Обобщенная структура формирователя сигналов управления МПА с программируемой логикой показана на рис. 4.6. Работу узла задает схема формирования адреса микрокоманды (СФАМ). Она обеспечивает определение адреса очередной микрокоманды, занесение его в регистр адреса управляющей памяти, считывание микрокоманды из управляющей памяти и занесение в регистр микрокоманды.

В состав СФАМ входит синхронизатор, аналогичный по структуре рассмотренному ранее (см. рис. 4.3), и вспомогательные логические схемы.

Цикл любой команды начинается с исходного состояния синхронизатора (такт T_0 — счетчик тактов обнулен). СФАМ заносит в регистр адреса управляющей памяти адрес первой МК микропрограммы выборки команды. В том же такте по сигналу чтения Чт происходит считывание адресуемой микрокоманды из управляющей памяти и ее запись в регистр микрокоманды (РМК).

---

[1] Аналогичную идею, независимо от Уилкса, в 1957 году выдвинул российский ученый Н. Я. Матюхин. Предложенное им УУ с программируемой логикой было реализовано в 1962 году в специализированной ВМ «Тетива», предназначенной для системы противовоздушной обороны.

Рис. 4.5. Размещение микропрограмм в управляющей памяти

Рис. 4.6. Формирователь сигналов управления в микропрограммном автомате с программируемой логикой

Занесение микрокоманды в РМК эквивалентно формированию указанных в ней сигналов управления. С очередным тактовым импульсом адрес текущей микрокоманды в СФАМ увеличивается на 1, а в РМК считывается следующая микрокоманда. Процесс повторяется до завершения микропрограммы выборки команды.

В последней микрокоманде этой микропрограммы в специальном бите указывается, что дальнейшие действия зависят от типа команды, помещенной в регистр команды. Единичное состояние этого бита для СФАМ означает, что в качестве адреса следующей МК должен быть выдан начальный адрес микропрограммы, реализующей этап исполнения команды данного типа. Такой адрес формируется преобразователем кода операции, и СФАМ выдает его в регистр адреса управляющей памяти вместо увеличенного на 1 адреса текущей МК. Далее происходит последовательное выполнение микропрограммы этапа исполнения. На завершающей фазе СФАМ передает управление микропрограмме формирования адреса следующей команды, которая увеличивает содержимое счетчика команд в командах, не нарушающих естественный порядок выполнения программы. В командах перехода формирование адреса следующей команды — это часть микропрограммы исполнения. В микрокоманде, завершающей формирование адреса следующей команды, указывается сигнал управления Сброс, устанавливающий синхронизатор в нулевое состояние, что соответствует нулевому такту T_0, то есть началу цикла следующей команды.

Каждая микрокоманда, считанная из управляющей памяти, в общем случае содержит микрооперационную (МО) и адресную (А) части. Микрооперационная часть микрокоманды поступает на дешифратор микрокоманды[1], на выходе которого образуются сигналы управления. Адресная часть микрокоманды подается в СФАМ, где формируется адрес следующей микрокоманды, который может зависеть от адреса на выходе преобразователя кода операции, адресной части текущей микрокоманды и значений осведомительных сигналов (признаков), поступающих от исполнительных устройств. Сформированный адрес микрокоманды снова записывается в регистр адреса управляющей памяти, и процесс повторяется вплоть до завершения микропрограммы.

Кодирование микрокоманд

Информация о том, какие сигналы управления должны быть сформированы в процессе выполнения текущей микрокоманды, в закодированном виде содержится в микрооперационной части (МО) микрокоманды. Способ кодирования микроопераций во многом определяет сложность аппаратных средств ФСУ и его скоростные характеристики. Известные варианты кодирования сигналов управления можно свести к трем группам: горизонтальному, вертикальному и смешанному кодированию. Структуры микропрограммных автоматов при различных способах кодирования микроопераций показаны на рис. 4.7 [11, 23].

При горизонтальном кодировании (см. рис. 4.7, *а*) каждый разряд микрооперационной части микрокоманды представляет один из возможных сигналов управления,

[1] В случае горизонтального кодирования микроопераций (поясняется далее) дешифратор микрокоманды не нужен.

то есть определенную микрооперацию. При общем количестве СУ равном m, длина микрооперационной части МК составляет m битов. Единица в определенном разряде МО означает необходимость формирования соответствующего СУ, независимо от состояния остальных битов МО. Независимость разрядов МО позволяет в рамках одного тактового периода одновременно выполнять вплоть до m микроопераций, причем в любом их сочетании, что является несомненным достоинством горизонтального кодирования. Кроме того, выходы регистра микрокоманды, куда помещается считанная из УПМ микрокоманда, могут быть подключены непосредственно к соответствующим управляемым точкам ВМ без предварительного декодирования МО (показанный на рис. 4.6 дешифратор микрокоманды не нужен). Основной недостаток горизонтального кодирования — большая длина МО, из-за чего возрастают требования к емкости управляющей памяти. По этой причине данный способ кодирования используется, если $m \leqslant 150$.

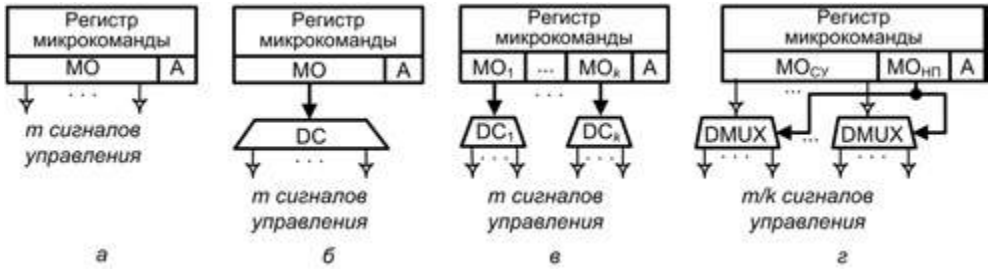


Рис. 4.7. Способы кодирования микроопераций: *а* — горизонтальный; *б* — вертикальный; *в* — горизонтально-вертикальный; *г* — вертикально-горизонтальный

При вертикальном кодировании (см. рис. 4.7, *б*) каждой микрооперации присваивается уникальный код, который и указывается в микрооперационной части МК. Разрядность этого кода (длина МО) равна наименьшему целому, большему или равному $\log_2 m$, что можно записать как $\lceil \log_2 m \rceil$. Таким образом, требования к емкости управляющей памяти по сравнению с горизонтальным кодированием существенно снижаются, но возникает потребность в дешифраторе для преобразования кода микрооперации в соответствующий сигнал управления. Кроме того, при вертикальном кодировании в МК можно указать лишь один сигнал управления. Микрооперации, которые можно было бы выполнить одновременно, приходится выполнять последовательно в разных тактовых периодах, что ведет к увеличению длины микропрограммы и времени ее реализации.

Компромиссное сочетание достоинств и недостатков горизонтального и вертикального кодирования достигается путем смешанного кодирования. Известны несколько вариантов такого кодирования, среди которых наиболее распространены *горизонтально-вертикальное* и *вертикально-горизонтальное* кодирование. В обоих вариантах множество всех m возможных сигналов управления разбивается на k подмножеств. При разбиении на подмножества каждая микрооперация может присутствовать лишь в одном из подмножеств.

В варианте горизонтально-вертикального кодирования (см. рис. 4.7, *в*) в микрооперационной части МК одновременно представлены все k подмножеств, что соответствует идее горизонтального кодирования. МО состоит из k полей ($МО_1$ ... $МО_k$), каждое из которых представляет m/k сигналов управления, входящих в соответствующее подмножество. Сигналы управления внутри каждого поля описываются вертикальным способом, то есть представлены кодами, каждый из которых уникален для данного подмножества. В таких условиях поле $МО_i$ состоит из $\lceil \log_2(m/k) \rceil$ битов, а общая длина МО в микрокоманде равна $k \times \lceil \log_2(m/k) \rceil$ битов. Распределение по подмножествам производится так, чтобы микрооперации, которые можно выполнить одновременно, оказались в разных подмножествах. Этим обеспечивается возможность одновременного выполнения вплоть до k микроопераций. Ввиду того, что микрооперации в каждом поле заданы вертикальным способом (закодированы), для формирования СУ каждое поле в регистре микрокоманд необходимо снабдить дешифратором. Так как количество микроопераций в подмножестве m/k меньше их общего числа m, то такие дешифраторы проще по сравнению с применяемыми в вертикальным кодировании, что положительно сказывается на быстродействии ФСУ.

В вертикально-горизонтальном способе (см. рис. 4.7, *г*) при разбиении СУ на подмножества ставится задача собрать в подмножестве те микрооперации, которые можно выполнять одновременно, в рамках одной микрокоманды. МО делится на два поля: $МО_{СУ}$ и $МО_{НП}$. Поле $МО_{СУ}$, длина которого равна максимальному количеству микроопераций в подмножестве (m/k), кодируется горизонтально, то есть для каждого СУ выделен один разряд, и декодирование содержимого поля $МО_{СУ}$ не требуется. В отдельной микрокоманде может быть представлено только одно из подмножеств. В поле $МО_{НП}$ указывается номер этого подмножества, иными словами, имеет место вертикальное кодирование. Длина этого поля равна $\lceil \log_2 k \rceil$, следовательно, общая длина МО составит $m/k + \lceil \log_2 k \rceil$ битов. При смене подмножества меняются и управляемые точки, куда должны быть направлены сигналы управления из каждой позиции $МО_{СУ}$. Это достигается с помощью демультиплексоров, управляемых кодом номера подмножества из поля $МО_{НП}$ регистра микрокоманд.

Из двух рассмотренных вариантов смешанного кодирования микроопераций более гибким и популярным считается горизонтально-вертикальный.

При горизонтальном, вертикальном и горизонтально-вертикальном способах кодирования микроопераций каждое поле микрокоманды несет фиксированные функции, то есть имеет место *прямое кодирование*. При *косвенном кодировании* одно из полей отводится для интерпретации других полей. Примером косвенного кодирования микроопераций может служить вертикально-горизонтальное кодирование.

Иногда используется двухуровневое кодирование микроопераций. На первом уровне с вертикальным кодированием выбирается микрокоманда, поле МО содержит адрес горизонтальной микрокоманды второго уровня — *нанокоманды*. Данный способ сочетания вертикального и горизонтального кодирования часто называют *нанокодированием*. Метод предполагает двухуровневую систему кодирования микроопераций и, соответственно, двухуровневую организацию управляющей памяти (рис 4.8).

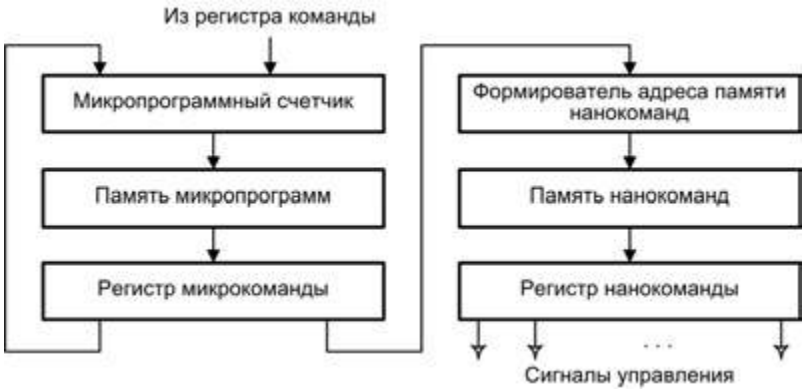


Рис. 4.8. Нанопрограммное устройство управления

Верхний уровень управления образуют микропрограммы, хранящиеся в памяти микропрограмм. Каждой микрокоманде соответствует какая-то из нанокоманд, хранящихся в управляющей памяти нижнего уровня — памяти нанокоманд. В нанокомандах применено горизонтальное кодирование микроопераций. Именно нанокоманды используются для непосредственного формирования сигналов управления. Микрокоманды вместо закодированного номера СУ содержат адресную ссылку на соответствующую нанокоманду.

Такой подход, сохраняя все достоинства горизонтального кодирования, позволяет значительно сократить суммарную емкость управляющей памяти. Дело в том, что число различных сочетаний сигналов управления, а следовательно, количество «длинных» нанокоманд, обычно невелико. В то же время практически все микрокоманды многократно повторяются в различных микропрограммах, и замена в микрокомандах «длинного» горизонтального управляющего поля на короткую адресную ссылку дает ощутимый эффект. Проиллюстрируем это примером.

Пусть в системе вырабатывается 200 управляющих сигналов, а общая длина микропрограмм составляет 2048 микрокоманд. Предположим также, что реально используется только 256 различных сочетаний сигналов управления. В случае обычного УУ с горизонтальным микропрограммированием емкость управляющей памяти составила бы $2048 \times 200 = 409\,600$ битов. При нанокодировании требуется память микропрограмм емкостью $2048 \times 8 = 16\,384$ бита (для указания одного из 256 сочетаний СУ достаточно 8 битов — $256 = 2^8$) и память нанокоманд емкостью $256 \times 200 = 51\,200$ битов. Таким образом, суммарная емкость обоих видов управляющей памяти равна 67 584 битам.

Основным недостатком нанокодирования является невысокое быстродействие, поскольку для выработки СУ требуются два обращения к памяти, что, однако, частично компенсируется исключением из схемы дешифратора, свойственного вертикальному кодированию.

Двухуровневое кодирование рассматривается как способ для уменьшения необходимой емкости управляющей памяти. Ее использование целесообразно только при многократном повторении микрокоманд в микропрограмме.

При разбиении микрокоманды на поля могут действовать два принципа: по функциональному назначению СУ и по ресурсам. *Функциональное кодирование* предполагает, что каждое поле соответствует одной функции внутри ВМ. Например, информация в аккумулятор может заноситься из различных источников, и для указания источника в микрокоманде может быть отведено одно поле, где каждая кодовая комбинация прикреплена к определенному источнику. При *ресурсном кодировании* ВМ рассматривается как набор независимых ресурсов (память, УВВ, АЛУ и т. п.), и каждому из ресурсов в микрокоманде отводится свое поле.

Обеспечение порядка следования микрокоманд

Порядок следования микрокоманд в микропрограмме в общем случае может зависеть от состояния какого-либо признака, обычно из тех, что хранятся в регистре признаков АЛУ. По этой причине в УУ необходимо предусмотреть эффективную систему реализации переходов. При выполнении микропрограммы адрес очередной микрокоманды относится к одной из трех категорий:

- определяется кодом операции команды;
- является следующим по порядку адресом;
- является адресом перехода.

Первый случай имеет место только один раз в каждом цикле команды, сразу же вслед за ее выборкой. Как уже отмечалось ранее, каждой команде ВМ в памяти микропрограмм соответствует «своя» микропрограмма, поэтому первое действие, которое нужно произвести после выборки команды, — преобразовать код операции в адрес первой микрокоманды соответствующей микропрограммы. Эта задача решается преобразователем кода операции (см. рис. 4.6). Такой преобразователь обычно реализуется в виде запоминающего устройства, хранящего начальные адреса микропрограмм в управляющей памяти. Обычно в микрокомандах выделяется специальное однобитовое поле, единица в котором указывает на то, что адрес следующей МК должен вычисляться на основе кода операции. Эта единица устанавливается в конце микропрограммы этапа выборки или в микропрограмме анализа кода операции, то есть перед выбором микропрограммы реализации текущей команды. При нулевом значении бита адрес следующей микрокоманды вычисляется исходя из адреса текущей МК или определяется адресом перехода.

Дальнейшая очередность выборки микрокоманд может быть задана путем указания в каждой МК адреса следующей микрокоманды (*принудительная адресация*) либо путем автоматического увеличения на единицу адреса текущей МК (*естественная адресация*) [15].

В обоих случаях необходимо предусмотреть ситуацию, когда адрес следующей микрокоманды зависит от состояния осведомительных сигналов. В качестве таких осведомительных сигналов выступают признаки, хранящиеся в регистре признаков АЛУ. Для указания того, какое условие должно быть проверено, в МК вводится поле условия перехода (УП), где каждому возможному признаку соответствует свой разряд. Содержимое УП определяет, состояние какого признака должно анализироваться при формировании адреса очередной МК микропрограммы. Если все

позиции в УП содержат 0, то никакие признаки не проверяются и адрес следующей МК берется из адресной части текущей микрокоманды. Если *i*-й разряд УП содержит единицу, то проверяется *i*-й разряд регистра признаков. При единичном состоянии последнего адрес следующей МК вычисляется путем увеличения на единицу адресной части текущей МК. Опишем схему формирования адреса для случая принудительной адресации, когда в МК указан лишь один адрес (рис. 4.9, *а*).

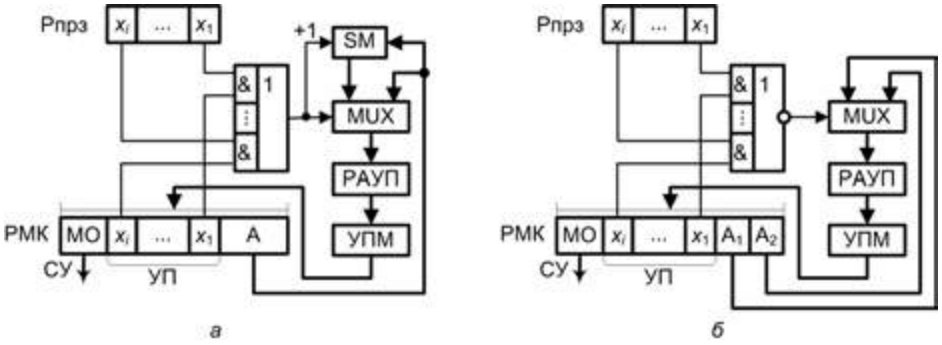


Рис. 4.9. Формирование адреса очередной микрокоманды при принудительной адресации микрокоманд: *а* — микрокоманда с одним адресным полем; *б* — микрокоманда с двумя адресными полями

На схеме состояние каждого разряда поля условия перехода (УП) в регистре микрокоманды (РМК) сравнивается с состоянием соответствующего бита в регистре признаков (Рпрз). Ноль на выходе схемы И-ИЛИ означает, что переход не нужен. Этот ноль коммутирует мультиплексор на занесение в регистр адреса управляющей памяти (РАУП) адресной части микрокоманды А из РМК. При наличии условия и его выполнении на выходе схемы И-ИЛИ появится единица, которая используется для увеличения А в сумматоре на единицу, и передачи суммы через мультиплексор в РАУП.

Рассмотренный способ позволяет учесть состояние только одного из осведомительных сигналов. Более гибкий подход реализован в ряде вычислительных машин фирмы IBM. В нем адреса микрокоманд считаются разбитыми на две части. Старшие n разрядов просто копируются из адресной части МК в старшие позиции РАУП, определяя блок из 2^n микрокоманд в памяти микропрограмм. Остальные (младшие) k разрядов РАУП устанавливаются в 1 или 0 в зависимости от того, проверка каких осведомительных сигналов была задана в поле УП и в каком состоянии эти сигналы находятся. Такой метод позволяет в одной микрокоманде сформировать 2^k вариантов перехода.

Возможен также вариант принудительной адресации, когда в МК указан не один адрес, а два (рис. 4.9, *б*). Здесь в зависимости от ситуации (есть готовность к переходу или нет) в регистр адреса управляющей памяти заносится один из двух адресов.

Основной недостаток принудительной адресации — повышенные требования к емкости управляющей памяти, так как все микрокоманды содержат адресное поле.

При естественной адресации отпадает необходимость во введении адресной части в каждую МК. Используются МК двух типов: операционные и управляющие. Тип МК определяет ее старший разряд, например: 0 — операционная МК, 1 — управляющая МК.

Операционная микрокоманда содержит только микрооперационную часть и не имеет адресной части. Подразумевается, что микрокоманды следуют в естественном порядке и процесс адресации следующей микрокоманды реализуется счетчиком микрокоманд (СМК). Значение СМК увеличивается на единицу после чтения из управляющей памяти микрокоманд (УПМ) очередной микрокоманды.

Для реализации условных и безусловных переходов используются специальные управляющие микрокоманды, состоящие только из двух полей: адресного и поля УП.

Структура МПА с естественной адресацией показана на рис. 4.10.

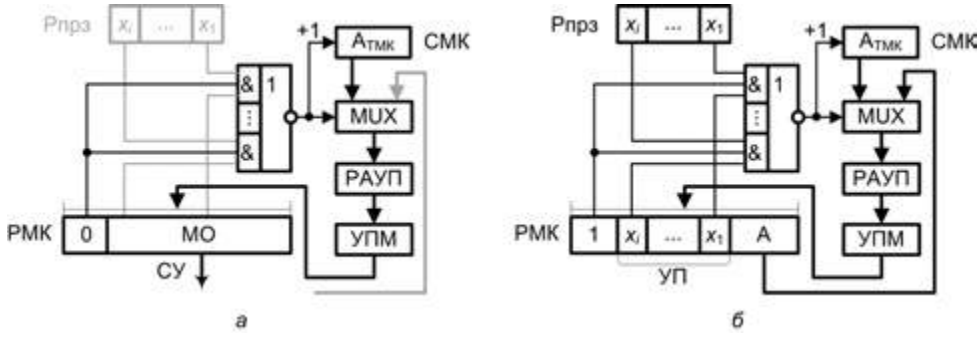


Рис. 4.10. Микропрограммный автомат с естественной адресацией при выполнении: *а* — операционной микрокоманды; *б* — управляющей микрокоманды

Достоинство естественной адресации — экономия памяти микропрограмм, а основной недостаток состоит в том, что для любого перехода требуется отдельный тактовый период, в то время как при принудительной адресации переход выполняется одновременно с формированием управляющих сигналов без дополнительных обращений к управляющей памяти.

Организация памяти микропрограмм

Функциональные возможности и структура микропрограммных автоматов в значительной степени зависят от организации управляющей памяти (УПМ) для хранения микропрограмм.

Основные способы организации памяти микропрограмм можно свести к следующим вариантам [3].

1. Каждое слово УПМ содержит одну микрокоманду. Это наиболее простая организация управляющей памяти. Основной недостаток — в каждом такте работы МПА требуется обращение к памяти микропрограмм, что приводит к снижению быстродействия МПА.

2. Одно слово УПМ содержит несколько микрокоманд. В результате осуществляется одновременное считывание из УПМ нескольких микрокоманд, что позволяет повысить быстродействие устройства управления.
3. Сегментация ПМП, при которой память разделяется на сегменты, состоящие из 2^q соседних слов, при этом адрес разделяется на два поля: S и A. Поле S определяет адрес сегмента, а поле A — адрес слова в сегменте. Адрес S устанавливается специальной микрокомандой. В последующих микрокомандах указывается только адрес слова A в сегменте. Таким образом, разрядность адресной части МК уменьшается.
4. Двухуровневая память. Данный вариант был рассмотрен при изложении идеи нанокодирования. Двухуровневая память рассматривается как способ для уменьшения необходимой емкости УПМ. Использование двухуровневой памяти целесообразно только при многократном повторении микрокоманд в микропрограмме.

Система прерывания программ

Способность реагировать на различные события, возникающие внутри и вне вычислительной машины, существенно расширяет возможности ВМ. В рамках устройства управления такой режим работы обеспечивается *системой прерывания программ* (СПП). Система прерывания программ — это совокупность аппаратных и программных средств, позволяющая ВМ (при получении соответствующего запроса) на время прервать выполнение текущей программы, передать управление программе обслуживания поступившего запроса, а по завершении последней продолжить прерванную программу с того места, где она была прервана[1]. Каждое событие, требующее прерывания, сопровождается оповещающим сигналом — *запросом прерывания* (ЗП). Программу, затребованную запросом прерывания, принято называть *прерывающей программой* или *программой обработки прерывания* (обработчиком прерывания). Важнейшим условием работы СПП является ее «прозрачность» — прерываемая программа после возобновления должна работать так, словно никаких прерываний не было.

Впервые СПП появилась в вычислительных машинах в конце 50-х годов прошлого столетия. Введение в состав ВМ системы прерывания существенно расширило сферы применения ВМ и открыло путь к широкому совмещению операций.

В зависимости от характера события, требующего реакции со стороны ВМ, сигналы прерывания разделяют на внешние, внутренние и программные. Источниками внешних прерываний могут быть устройства ввода/вывода и иные, внешние объекты. Внутренние запросы прерывания формируются внутри процессора схемами контроля ВМ и могут возникать, например, при делении на 0, арифметическом переполнении, неверном коде команды и т. п., Причинами программных прерываний обычно являются обращения программы пользователя к операционной системе.

[1] ГОСТ 15971-90 определяет прерывание как «операцию процессора, состоящую в регистрации предшествующего прерыванию состояния процессора и установлении нового состояния».

Цикл команды с учетом прерываний

Процедуру обработки любого прерывания можно разделить на несколько этапов:

1. Установка запрета на прием запросов прерывания.
2. Сохранение всей информации прерванной программы, которая необходима для возобновления выполнения этой программы (контекста программы) после завершения обработки прерывания.
3. Снятие запрета на прием запросов прерывания.
4. Идентификация источника ЗП, определение нужного обработчика прерывания и его запуск.
5. Завершение программы обработки прерывания.
6. Установка запрета на прием запросов прерывания.
7. Восстановление контекста прерванной программы (возврат к состоянию на момент прерывания).
8. Снятие запрета на прием запросов прерывания.
9. Возврат к выполнению прерванной программы.

В упрощенном виде процедуру прерывания можно описать следующим образом. Объект, требующий внеочередного обслуживания, выставляет на соответствующем входе центрального процессора (ЦП) сигнал запроса прерывания. Перед переходом к очередному циклу команды процессор проверяет состояние этого входа. Обнаружив запрос, ЦП запоминает информацию, необходимую для продолжения нормальной работы после возврата из прерывания, и переходит к выполнению программы обработки прерывания. По завершении обработки запроса ЦП восстанавливает состояние прерванного процесса, используя сохраненную информацию, и продолжает вычисления. Описанный процесс иллюстрирует рис. 4.11.

Рис. 4.11. Передача управления при прерываниях

Для обеспечения перечисленных возможностей в цикл команды добавляется этап прерывания. На этом этапе процессор проверяет, не поступил ли запрос прерывания. Если запроса нет, ЦП переходит к этапу выборки следующей команды программы. При наличии запроса процессор:

1) приостанавливает выполнение текущей программы и запоминает в стеке контекст программы (содержимое всех узлов, состояние которых может быть изменено программой обработки прерывания). В первую очередь необходимо сохранить содержимое счетчика команд, аккумулятора и регистра признаков;
2) заносит в счетчик команд начальный адрес программы обработки прерывания.

Соответственно, в гипотетической ВМ (см. главу 3) должна быть развернута следующая микропрограмма:

```
РДП_СК: РДП := СК; <Запись в стек содержимого счетчика команд>
–1УС: УС := УС – 1;
РАП_УС: РАП := УС;
ЗпЗУ: ОП[(РАП)] := РДП;
РДП_Акк: РДП := Акк; <Запись в стек содержимого аккумулятора>
–1УС: УС := УС – 1;
РАП_УС: РАП := УС;
ЗпЗУ: ОП[(РАП)] := РДП;
РДП_Рпрз: РДП := Рпрз; <Запись в стек содержимого регистра признаков>
–1УС: УС := УС – 1;
РАП_УС: РАП := УС;
ЗпЗУ: ОП[(РАП)] := РДП;
СК := Аобр. <Занесение в СК начального адреса обработчика прерывания>
```

Теперь процессор продолжает с этапа выборки первой команды обработчика прерывания. Когда программа обработки прерывания завершается, процессор может возобновить выполнение программы пользователя с точки, где она была прервана. Для этого он восстанавливает из стека контекст программы и начинает с цикла выборки очередной команды прерванной программы. Требуемая микропрограмма представлена ниже.

```
РАП_УС: РАП := УС; <Восстановление состояния регистра признаков>
ЧтЗУ: РДП := ОП[(РАП)] ;
+1УС: УС := УС + 1;
Рпрз_РДП: Рпрз := РДП;
РАП_УС: РАП := УС; <Восстановление содержимого аккумулятора>
ЧтЗУ: РДП := ОП[(РАП)];
+1УС: УС := УС + 1;
Акк_РДП: Акк := РДП;
РАП_УС: РАП := УС; <Восстановление содержимого счетчика команд>
ЧтЗУ: РДП := ОП[(РАП)];
+1УС: УС := УС + 1;
СК_РДП: СК := РДП.
```

Характеристики систем прерывания

Системы прерывания принято характеризовать следующими параметрами:

- *время реакции* $T_Р$ — время между появлением запроса прерывания и началом выполнения первой команды обработчика прерывания;
- *затраты времени на переключение программ* — суммарный расход времени на запоминание $T_З$ и восстановление $T_В$ состояния программы;

- *эффективность прерывания* η — отношение времени выполнения прерывающей программы к общему времени, необходимому для обслуживания прерывания;
- *глубина прерываний* — максимальное число программ, которые могут последовательно прерывать друг друга.

Время реакции, в общем случае, существенно зависит от приоритета запроса прерывания. При сравнении систем прерывания обычно используют величину времени реакции $T_Р$ на запрос с наивысшим приоритетом.

Затраты времени на переключение программ иногда называют временем обслуживания прерывания ($T_{ОБС} = T_З + T_В$) и определяют как разность между полным временем выполнения прерывающей программы ($T_{ПР}$) и временем выполнения «полезных» команд ($T_П$).

С учетом введенных обозначений эффективность прерывания, называемую иногда удельным весом прерывающих программ, можно определить как $\eta = \frac{T_П}{T_{ПР}}$.

С позиций глубины прерывания можно рассматривать три варианта СПП:

- СПП способна воспринимать только один запрос;
- глубина прерываний ограничена некоторым значением n;
- программы могут неограниченно прерывать друг друга.

Если после перехода к прерывающей программе и вплоть до ее завершения прием других запросов запрещается, то говорят, что система имеет глубину прерывания, равную 1. Глубина прерывания равна n, если допускается последовательное прерывание до n программ. Обычно глубина прерывания совпадает с количеством уровней приоритета, распознаваемых системой прерывания программ. Чем больше глубина прерывания, тем лучшее приоритетное обслуживание обеспечивает СПП.

Рассмотренные характеристики систем прерывания программ проиллюстрированы рис. 4.12 и 4.13

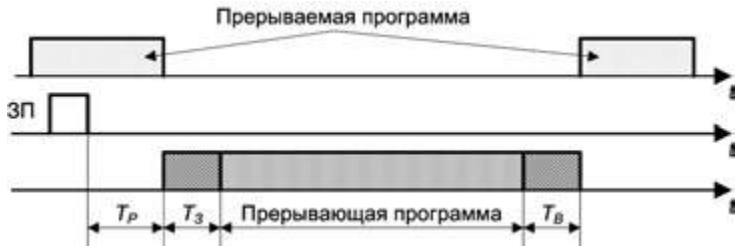


Рис. 4.12. Упрощенная временная диаграмма процесса прерывания

Запрос прерывания обязательно должен быть обслужен до момента прихода ЗП от того же источника, в противном случае возникает *насыщение системы прерывания*, и предыдущий запрос может остаться необслуженным. Такая ситуация является недопустимой, поэтому быстродействие ВМ и характеристики СПП должны быть согласованы так, чтобы насыщение было невозможным.

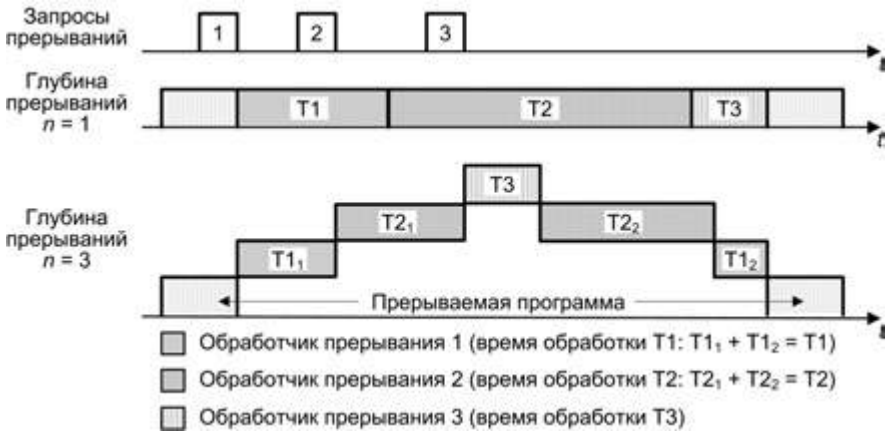


Рис. 4.13. Прерывания в СПП с различной глубиной прерываний (без учета времени реакции и обслуживания)

Допустимые моменты прерывания программ

Эффективность СПП во многом зависит от того, в какой момент допускается прерывание выполняемой программы. В известных ВМ этот момент определяется одним из трех методов.

- *Метод помеченного оператора*, известный также как *метод опорных точек*, предполагает наличие в коде команд специального бита, единичное значение которого означает разрешение прерывания по завершении данной команды, а нулевое — запрет. Это позволяет расставить в программе «опорные точки» так, чтобы содержимое регистров процессора, используемых в обработчике прерывания, после возврата к прерванной программе уже не требовалось, то есть исключить запоминание содержимого таких регистров. В результате сокращается время обслуживания, однако время реакции увеличивается.
- В *покомандном методе* прерывание допускается после завершения любой текущей команды. Метод обеспечивает уменьшение времени реакции $T_{\text{Р}}$, но при этом возрастает время обслуживания прерывания $T_{\text{ОБС}}$.
- *Метод быстрого реагирования* допускает прерывание после любого такта выполнения команды. Для метода характерно минимальное время реакции $T_{\text{Р}} \rightarrow \min$, но, с другой стороны, возрастает объем запоминаемого контекста программы (количества информации, подлежащей запоминанию и восстановлению при переключении программ) и, соответственно, время обслуживания прерывания $T_{\text{ОБС}}$.

Наибольшее распространение среди рассмотренных методов получил покомандный метод, хотя в современных ВМ все более популярным становится метод быстрого реагирования, особенно в ВМ, работающих в режиме реального времени.

Дисциплины обслуживания множественных прерываний

В любой ВМ предполагается наличие множества источников запросов прерывания. В СПП с множественными запросами оперируют понятием «код прерывания», под

которым понимают двоичное слово $P = p_{n-1} \ldots p_1 p_0$, где каждый разряд соответствует определенной причине прерывания. Установка в единицу разряда p_i означает, что поступил запрос от i-го источника.

В ряде случаев процессор должен в обязательном порядке реагировать на ЗП, однако иногда процессор не должен отзываться на запрос, например, если в данное время обслуживается запрос с более высоким уровнем приоритета. По этой причине в большинстве ВМ различают два вида запросов прерывания: немаскируемые и маскируемые.

Немаскируемые запросы прерывания не могут управляться программистом. Сигналы немаскируемых запросов поступают по отдельной шине и имеют наивысший приоритет. Обычно немаскируемыми делают запросы от таких важнейших компонентов ВМ, как системы питания, схемы контроля правильности передач данных и т. п.

Маскируемые запросы прерывания управляются командами программы и дают программисту возможность гибкого управления вычислительным процессом. Запрещение прерывания производится кодом защиты от прерывания — маской прерываний. Маска прерываний — это двоичное слово $M = m_{n-1} \ldots m_1 m_0$ с числом разрядов, равным количеству маскируемых причин прерывания. Если разряд маски $m_i = 0$, то запрос $ЗП_i$ игнорируется, а при $m_i = 1$ — прерывание допустимо. В случае маскируемых запросов взамен кода прерывания P принимается во внимание результат поразрядного логического умножения этого кода и маски прерываний M ($M \wedge P$).

С учетом двух видов запросов прерывания СПП обычно имеет два физических входа для сигналов ЗП — отдельно для немаскируемых и маскируемых запросов. Ввиду ограничения на число входов все линии, по которым приходят ЗП от различных источников, предварительно каким-либо образом объединяются, образуя общий сигнал прерывания, и именно этот сигнал подается в процессор. Объединение ЗП порождает задачу идентификации источника прерывания и выбора запроса с наиболее высоким приоритетом.

Идентификация источника запроса прерывания

Проблему идентификации источника запроса прерывания сначала рассмотрим при условии, что общий сигнал прерывания вызван запросом только от одного из источников и что приоритет этого запроса выше, чем у прерываемой программы.

По способу выявления этого источника различают *обзорные* СПП (с опросом источников прерывания) и ССП с *векторными прерываниями*.

В обзорных СПП поступление в процессор сигнала ЗП инициирует общую программу обработки прерываний. В ее задачи входит опрос потенциальных источников ЗП и выяснение того из них, которые породил общий сигнал прерывания в ЦП. Простейшим вариантом такого подхода может служить СПП с регистром запросов прерывания (рис. 4.14).

В такой СПП каждому потенциальному источнику ЗП выделен отдельный разряд в параллельном регистре запросов прерывания (РЗП). При поступлении запроса

от i-го источника $ЗП_i$ соответствующий (i-й) разряд РЗП устанавливается в 1. Выходы всех разрядов РЗП подключены к логическому элементу «ИЛИ», где формируется общий сигнал прерывания ($ЗП_{ОБЩ}$), если хотя бы в одном из разрядов РЗП содержится единица. Поступив в процессор, этот сигнал вызывает общую прерывающую программу, которая считывает состояние РЗП и определяет, какой разряд содержит единицу. После определения такого разряда вызывается соответствующая запросу программа обработки прерывания. По завершении обработки прерывания процессор путем посылки сигнала подтверждения $ПДТ_i$ обнуляет тот разряд РЗП, для которого выполнялась обработка. СПП получается простой, но одновременно и самой медленнодействующей, поскольку анализ запросов выполняется программно.

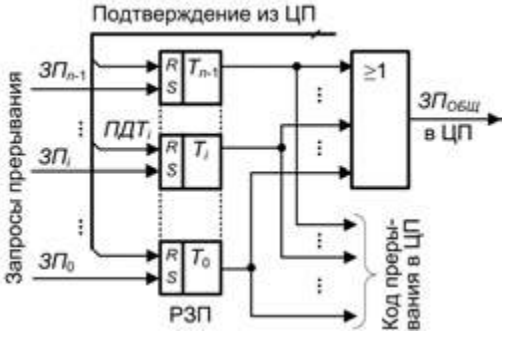


Рис. 4.14. Система прерывания с регистром запросов прерывания

Обзорные системы прерывания предполагают процедуру опроса источников прерывания, которая, даже если и выполняется аппаратными средствами, требует сравнительно больших затрат времени. Более гибкими и динамичными являются СПП с векторными прерываниями, где исключается опрос источников прерывания. По этой причине в современных ВМ доминируют векторные СПП, и именно они будут предметом дальнейшего рассмотрения.

Прерывание называется векторным, если источник прерывания, выставляя запрос, посылает в процессор информацию о месте расположения в памяти ВМ своего вектора прерывания. Вектор прерывания содержит адрес соответствующей ему прерывающей программы, что и позволяет сразу же передать ей управление.

Идею векторного прерывания иллюстрирует рис. 4.15. Как и в схеме с регистром запросов прерывания, общий сигнал запроса $ЗП_{ОБЩ}$ формируется путем объединения сигналов ЗП от всех потенциальных источников. Так как векторы прерываний поступают в процессор по общей шине данных, то возникает проблема подключения к этой шине именно того источника, который выставил запрос. На рис. 4.15 показан простейший вариант ее решения — цепочечный. Если процессор готов к обслуживанию прерывания, он выдает сигнал подтверждения ПДТ, который последовательно проходит через источники запросов, образующие цепочку, пока не достигнет источника, выставившего запрос. Именно этот источник получает право выдать на шину данных свой вектор прерывания.

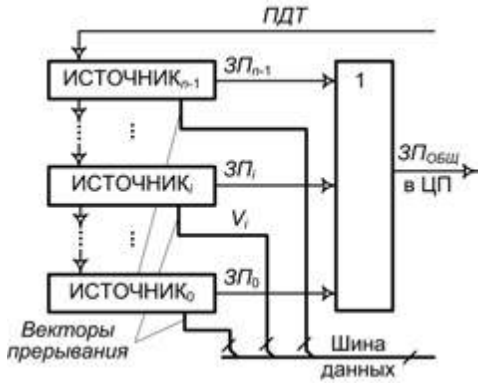


Рис. 4.15. Цепочечная схема векторных прерываний

Векторы прерывания хранятся в фиксированных ячейках основной памяти, поэтому обычно формируется не собственно вектор прерывания, а код номера запроса (КНЗ), который либо совпадает, либо легко преобразуется в адрес той фиксированной ячейки, где хранится соответствующий вектор прерывания.

Фиксированные ячейки, где хранятся векторы, рассматриваются как элементы таблицы векторов прерывания. Таблица эта обычно размещается в основной памяти, начиная с адреса 0. Элемент таблицы занимает несколько смежных ячеек, с тем чтобы в нем можно было разместить полный адрес обработчика прерываний (вектор прерывания). Пример таблицы векторов показан на рис. 4.16. Предполагается, что она располагается, начиная с адреса 0000H[1], а длина вектора прерывания равна четырем байтам (32 бита). Так как при 8-битовой длине ячейки памяти вектор занимает четыре смежных ячейки, то адрес входа в таблицу вычисляется умножением КНЗ на четыре (это достигается сдвигом кода номера запроса на два разряда влево).

Рис. 4.16. Структура таблицы векторов прерывания

Векторная организация позволяет отказаться от общей программы обработки прерывания, благодаря чему обеспечивается малое время реакции.

[1] Символ H означает, что адрес указан в 16-й системе счисления.

Выбор и обслуживание запроса с наиболее высоким приоритетом

При наличии нескольких источников прерывания должен быть установлен определенный порядок (дисциплина) обслуживания поступивших запросов. Иными словами, между запросами должны быть установлены приоритетные соотношения, определяющие, какой из нескольких поступивших запросов подлежит обработке в первую очередь, а также имеет ли право данный запрос прервать выполняемую в данный момент программу.

Каждому потенциальному источнику ЗП присваивается свой уровень приоритета. Определенный уровень приоритета имеет и выполняемая в данный момент программа. При поступлении нескольких запросов СПП, прежде всего, должна выделить из них тот, который имеет наиболее высокий уровень приоритета. Далее этот уровень сравнивается с уровнем приоритета прерываемой программы. Общий сигнал запроса $ЗП_{ОБЩ}$ должен подаваться на вход процессора только в случае, когда уровень приоритета прерываемой программы ниже.

В случае векторной СПП все эти задачи возлагаются на контроллер векторных прерываний (КВП), обобщенная структура которого показана на рис. 4.17.

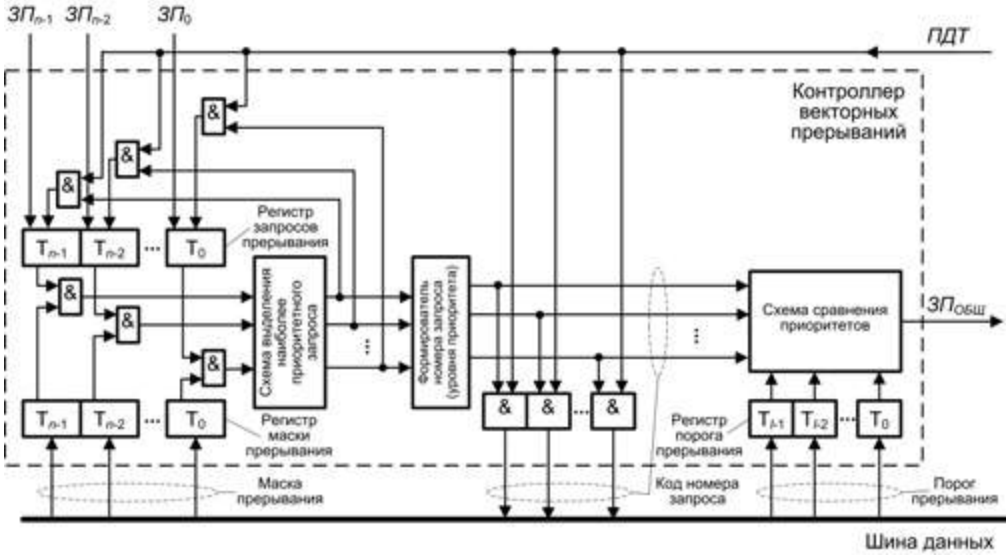


Рис. 4.17. Структура контроллера векторных прерываний

КВП имеет отдельные входы для каждого из n источников ЗП. Обычно это 8 или 16 входов. Поступившие запросы фиксируются в регистре запросов прерывания.

Дальнейшее прохождение запросов зависит от состояния разрядов регистра маски прерывания. Код, поступающий на схему выделения наиболее приоритетного запроса, формируется поразрядным логическим умножением кода прерывания P и маски прерываний M ($M \wedge P$). Маска прерываний задается программистом и заносится в регистр маски КВП из процессора в начале вычислений, но может быть заменена в ходе вычислений.

На вход схемы выделения наиболее приоритетного запроса поступает код, содержащий единицы в позициях, соответствующих незамаскированным ЗП. Если таких единиц несколько (поступило несколько запросов), схема оставляет из них одну, а именно ту, что представляет запрос, имеющий более высокий уровень приоритета.

Далее код с единственной единицей поступает на вход формирователя номера запроса. Обычно код номера запроса (КНЗ) совпадает с номером бита в коде прерывания. Таким образом, формирователь преобразует унитарный код на своем входе в двоичный код номера запроса. По сложившейся практике уровень приоритета принимается равным КНЗ, причем считается, что наименьшим уровнем приоритета обладает запрос с КНЗ = 0.

Как уже отмечалось выше, прервать выполняемую программу может лишь тот ЗП, у которого уровень приоритета больше, чем у прерываемой программы. Уровень приоритета выполняемой в данный момент программы хранится в регистре порога прерываний. Содержимое регистра обновляется при каждом изменении уровня приоритета программы, текущее значение регистра называют порогом прерывания. Схема сравнения приоритетов сопоставляет приоритет поступившего запроса с порогом прерывания, и только если порог превышен (приоритет запроса выше), вырабатывает общий сигнал запроса прерывания $ЗП_{ОБЩ}$.

Получив сигнал $ЗП_{ОБЩ}$, процессор выдает в контроллер векторных прерываний сигнал подтверждения ПДТ. По этому сигналу КВП выставляет на шине данных код номера запроса, который поступает в процессор и используется как номер входа в таблицу векторов прерываний. Одновременно сигнал ПДТ обнуляет в регистре запросов прерывания разряд, представляющий ЗП, к обслуживанию которого приступил процессор.

В свою очередь, процессор, используя принятый КНЗ в качестве номера входа в таблицу векторов, извлекает из последней вектор прерывания (адрес обработчика) и приступает к обслуживанию прерывания. В момент запуска обработчика прерывания в регистр порога прерывания КВП заносится уровень приоритета обработчика прерывания, численно равный КНЗ. Обновление содержимого регистра порога может быть произведено процессором, но возможно и средствами контроллера (на схеме такой вариант не отражен). В любом случае, при возврате к прерванной программе процессор должен вернуть в регистр порога прерывания значение уровня приоритета этой программы.

Система приоритетов

Система приоритетов позволяет определить: имеет ли право поступивший запрос прерывания прервать выполняемую в данный момент программу. В случае одновременного поступления нескольких ЗП эта система дает возможность выбрать тот из них, который должен быть обслужен в первую очередь.

Различают абсолютный и относительный приоритеты. Запрос, имеющий абсолютный приоритет, прерывает выполняемую программу и запускает выполнение соответствующей прерывающей программы. Запрос с относительным приоритетом является первым кандидатом на обслуживание после завершения выполнения текущей программы.

Простейший способ установления приоритетных соотношений между источниками ЗП заключается в том, что уровень приоритета определяется порядком присоединения линий сигналов запроса к линии общего сигнала ЗП. При появлении нескольких ЗП первым воспринимается запрос, поступивший на вход с большим номером. В этом случае приоритет является жестко фиксированным. Изменить приоритетные соотношения можно лишь изменением порядка подключения источников запросов на входах системы прерывания.

На практике степень важности прерывающих программ в процессе вычислений может меняться, поэтому соотношение приоритетов между ЗП должно быть динамичным, то есть программно-управляемым. В ВМ широко применяются два способа реализации программно-управляемого приоритета: способ порога прерывания и способ маски прерывания.

Сущность способа порога прерывания и его реализация были рассмотрены в рамках описания контроллера векторных прерываний.

В методе маски, изменяя содержимое регистра маски, можно устанавливать произвольные приоритетные соотношения между программами без коммутации линий, по которым поступают запросы. Каждая прерывающая программа может установить свою маску. С замаскированным запросом можно поступить двояко: или он игнорируется, или запоминается, с тем чтобы осуществить затребованные действия, когда запрет будет снят.

Метод порога более часто используется в микро- и мини-ЭВМ, а метод маски — в ВМ общего назначения.

Запоминание состояния процессора при прерываниях

Поскольку обслуживание запросов прерывания связано с необходимостью временного переключения на новую программу, то для обеспечения последующего корректного возврата к прерванной программе требуется запомнить состояние основных узлов ВМ на момент прерывания — контекст программы.

Неотъемлемым элементом контекста программы является состояние счетчика команд — там хранится адрес команды, с которой начинается возобновление выполнения прерванной программы. Как правило, работа обработчика прерывания сопровождается изменением содержимого аккумулятора (Акк) и регистра признаков (Рпрз), и последние также следует считать обязательными элементами контекста. Запоминание содержимого СК, Акк и Рпрз и их восстановление по завершении прерывания производится автоматически.

Помимо этого обработчик прерывания может менять и содержимое некоторых регистров общего назначения (РОН). Хотя в некоторых ВМ их состояние также сохраняется и восстанавливается автоматически, в общем случае забота о РОН, используемых в прерывающей программе, возлагается на эту прерывающую программу. Команды, отвечающие за такое сохранение и восстановление, размещаются в самом начале и самом конце обработчика прерывания соответственно. На время выполнения этих команд система прерывания отключается, чтобы исключить искажение сохраняемой и восстанавливаемой информации в случае, если в этот момент поступит новый ЗП и начнется его обслуживание.

Подлежащая сохранению информация, как правило, заносится в аппаратный стек. В общем случае сохранение содержимого регистров процессора можно осуществить следующими способами:

- прерывающая программа решает, содержимое каких регистров необходимо сохранить. Этот способ требует наименьших затрат, если система команд построена так, что в большинстве операций принимает участие малое число регистров;
- при прерывании с помощью аппаратных средств происходит автоматическая передача содержимого всех РОН в стек, а после окончания обработки прерывания — обратная передача. Эта система сохраняет содержимое всех регистров, вне зависимости от того, есть ли в этом необходимость.

В роли регистров в прерывающих программах используются ячейки основной памяти. Эта часть памяти разделена на секции, и каждой программе обработки прерываний выделена своя секция. Таким образом, каждая прерывающая программа как бы имеет свой независимый набор регистров, поэтому в их запоминании и восстановлении нет необходимости.

Вычислительные машины с опросом внешних запросов

В ряде случаев процессоры могут не иметь системы прерываний. Реагирование на внешние события в них осуществляется за счет периодического опроса источников таких событий («сканирования входов»). Если внешнее событие предполагает обслуживание, то запускается обработчик события, аналогичный обработчику прерывания. Опрос осуществляется периодически, причем частота опроса может выбираться в зависимости от важности события и возможной интенсивности возникновения данного события. Чтобы учесть случай возникновения события с высоким приоритетом в момент обслуживания события низкого приоритета, обработчики низкоприоритетных событий сами должны содержать цикл опроса высокоприоритетных событий.

Системы с опросом можно встретить в микроконтроллерах, например в тех, что управляют клавиатурой персонального компьютера. С другой стороны, системы с опросом встречаются в суперкомпьютерах, где для уменьшения нагрузки на центральный процессор, связанной с опросом устройств, используются специализированные процессоры, основной задачей которых является опрос источников событий.

Контрольные вопросы

1. Охарактеризуйте основные функции устройства управления.
2. Дайте характеристику входной и выходной информации модели УУ.
3. Поясните задачи декодирования команд и основные этапы такого декодирования.
4. Перечислите достоинства и недостатки МПА с аппаратной логикой.
5. Обоснуйте название МПА с программируемой логикой. Сформулируйте достоинства и недостатки таких МПА.

6. Объясните принцип управления на основе МПА с программируемой логикой.
7. Какие способы кодирования микрокоманд вы знаете? Перечислите их достоинства и недостатки.
8. Поясните подходы к адресации микрокоманд, охарактеризуйте их сильные и слабые стороны.
9. Какими параметрами характеризуются системы прерывания программ?
10. На какой стадии выполнения команды анализируются запросы прерывания?
11. Опишите последовательность действий, выполняемых при поступлении запроса прерывания.
12. Решение каких проблем позволяет решить маскирование прерываний?
13. Какие методы используются для идентификации источника запроса прерывания?
14. Как обеспечивается возобновление вычислений после обработки прерывания?
15. Для чего в обработчике прерываний предусмотрен временный запрет на прием запросов прерывания? Когда он вводится и снимается?

Глава 5

Операционные устройства

В классической фон-неймановской ВМ арифметическая и логическая обработка данных возлагается на арифметико-логическое устройство (АЛУ). В реальных вычислительных машинах АЛУ обычно реализуется не как единое устройство, а в виде комплекса операционных устройств (ОПУ), каждое из которых ориентировано на определенные операции и определенные типы данных. В общем случае можно выделить:

- ОПУ для арифметической обработки чисел в форме с фиксированной запятой;
- ОПУ для арифметической обработки чисел в форме с плавающей запятой;
- ОПУ для логической обработки данных;
- ОПУ десятичной арифметики.

Специализированные ОПУ для логической обработки данных и десятичной арифметики в современных ВМ встречаются достаточно редко, поскольку подобную обработку можно достаточно эффективно организовать на базе ОПУ для чисел с фиксированной запятой. Таким образом, будем считать, что АЛУ образуют два вида операционных устройств: для обработки чисел в форме с фиксированной запятой (ФЗ) и обработки чисел в форме с плавающей запятой (ПЗ). В свою очередь, эти ОПУ также могут представлять собой совокупность операционных устройств, специализированных под определенную арифметическую операцию, например ОПУ сложения и вычитания, ОПУ умножения, ОПУ деления и т. п.

В минимальном варианте АЛУ должно содержать аппаратуру для реализации лишь основных логических операций, сдвигов, а также сложения и вычитания чисел в форме с фиксированной запятой. Опираясь на этот набор, можно программным способом обеспечить выполнение остальных арифметических и логических операций как для чисел с фиксированной запятой, так и для других форм представления информации. Подобный вариант, однако, не позволяет добиться высокой скорости вычислений, поэтому по мере расширения технологических возможностей доля аппаратных средств в АЛУ постоянно возрастает.

На рис. 5.1 показана динамика изменения соотношения между аппаратной и программной реализациями функций АЛУ по мере развития элементной базы вычисли-

тельной техники. Здесь подразумевается, что по вертикальной оси откладывается календарное время.

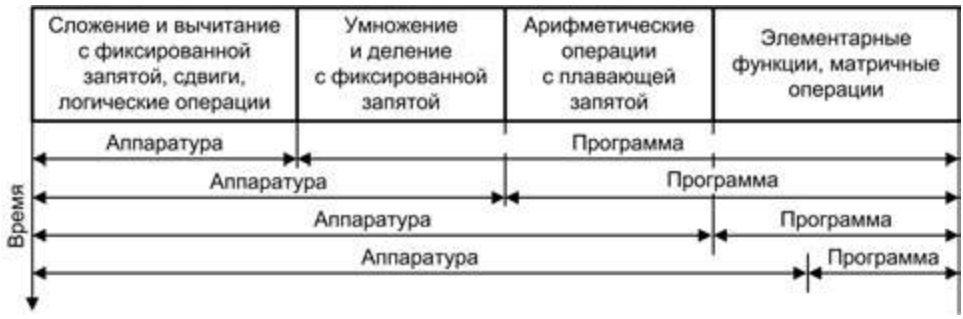


Рис. 5.1. Динамика изменения соотношения между аппаратной и программной реализациями функций АЛУ

Структуры операционных устройств

Набор элементов, на основе которых строятся различные ОПУ, называется *структурным базисом*. Структурный базис ОПУ включает в себя:

- регистры, обеспечивающие кратковременное хранение слов данных;
- управляемые шины, предназначенные для передачи слов данных;
- комбинационные схемы, реализующие вычисление логических условий и выполнение микроопераций по сигналам от устройства управления.

Используя методику, изложенную в [15], можно синтезировать ОПУ с так называемой канонической структурой, являющуюся основополагающей для синтеза других структур. Каноническая структура предполагает максимальную производительность по сравнению с другими вариантами структур, однако по затратам оборудования является избыточной. С практических позиций больший интерес представляют другие виды структур ОПУ: жесткая и магистральная.

Операционные устройства с жесткой структурой

В ОПУ с жесткой структурой комбинационные схемы жестко распределены между всеми регистрами. Каждому регистру придается свой набор комбинационных схем, позволяющих реализовать определенные микрооперации. Пример ОПУ с жесткой структурой, обеспечивающего выполнение операций типа «сложение», приведен на рис. 5.2.

В состав ОПУ входят три регистра со своими логическими схемами:

- регистр первого слагаемого РСл1 и схема ЛРСл1;
- регистр второго слагаемого РСл2 и схема ЛРСл2;
- регистр суммы РСм и схема комбинационного сумматора См.

Логическая схема ЛРСл1 обеспечивает передачу результата из регистра РСм в регистр РСл1:

- прямым кодом РСл1 := РСм (по сигналу управления $П_2$РСл1);
- со сдвигом на один разряд влево РСл1 := L1(РСм · 0) (по сигналу управления L_1РСм);
- со сдвигом на два разряда вправо РСл1 := R2(s · s · РСм) (по сигналу управления R_1РСм).

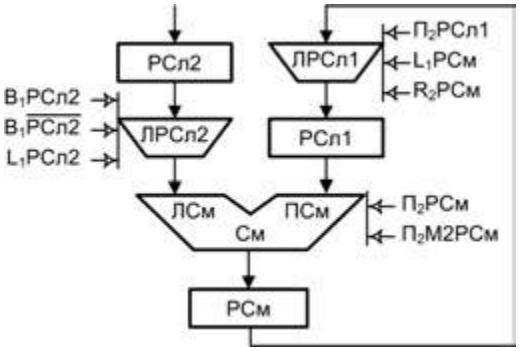


Рис. 5.2. Операционное устройство с жесткой структурой

Логическая схема ЛРСл2 реализует микрооперации передачи второго слагаемого из РСл2 на левый вход сумматора:

- прямым кодом ЛСм := РСл2 (по сигналу управления B_1РСл2);
- инверсным кодом ЛСм := $\overline{РСл2}$ (по сигналу управления $B_1\overline{РСл2}$);
- со сдвигом на один разряд влево ЛСм := L1(РСл2 · 0) (по сигналу управления L_1РСл2).

Комбинационный сумматор См предназначен для суммирования (обычного или по модулю 2) операндов, поступивших на его левый (ЛСм) и правый (ПСм) входы. Результат суммирования заносится в регистр РСм: РСм := ЛСм + ПСм (по сигналу управления $П_2$РСм) или РСм := ЛСм ⊕ ПСм (по сигналу управления $П_2$М2РСм).

Моделью ОПУ с жесткой структурой является так называемый *I*-автомат, с особенностями синтеза которого можно ознакомиться в [15, 20].

Аппаратные затраты на ОПУ с жесткой структурой $C_Ж$ можно оценить по выражению:

$$C_Ж = nC_T N + 3\sum_{i=1}^{N} n_i \sum_{j=1}^{K} k_{ij} + \sum_{i=1}^{N} n_i \sum_{j=1}^{K} C_j k_{ij},$$

где N — количество внутренних слов ОПУ; $n_1, ..., n_N$ — длины слов; $n = (n_1 + ... + n_N)/N$ — средняя длина слова; k_{ij} — количество микроопераций типа $j = 1, 2, ..., K$ (сложение, сдвиг, передача и т. п.), используемых для вычисления слов с номерами $i = 1, 2, ..., N$; C_T — цена триггера; C_j — цена одноразрядной схемы для реализации микрооперации j-го типа.

В приведенном выражении первое слагаемое определяет затраты на хранение n-разрядных слов, второе — на связи регистров с комбинационными схемами,

а третье — суммарную стоимость комбинационных схем, реализующих микрооперации K типов над N словами.

Затраты времени на выполнение операций типа «сложение» в ОПУ с жесткой структурой равны:

$$T_{Ж} = t_{В} + t_{С} + t_{П},$$

где $t_{В}$ — длительность микрооперации выдачи операндов из регистров; $t_{С}$ — продолжительность микрооперации «сложение»; $t_{П}$ — длительность микрооперации приема результата в регистр.

Достоинством ОПУ с жесткой структурой является высокое быстродействие, недостатком — малая регулярность структуры, что затрудняет реализацию таких ОПУ в виде больших интегральных схем.

Операционные устройства с магистральной структурой

В ОПУ с магистральной структурой все внутренние регистры объединены в отдельный узел регистров общего назначения (РОН)[1], а все комбинационные схемы — в операционный блок (ОПБ). Операционный блок и узел регистров сообщаются между собой с помощью магистралей — отсюда и название «магистральное ОПУ».

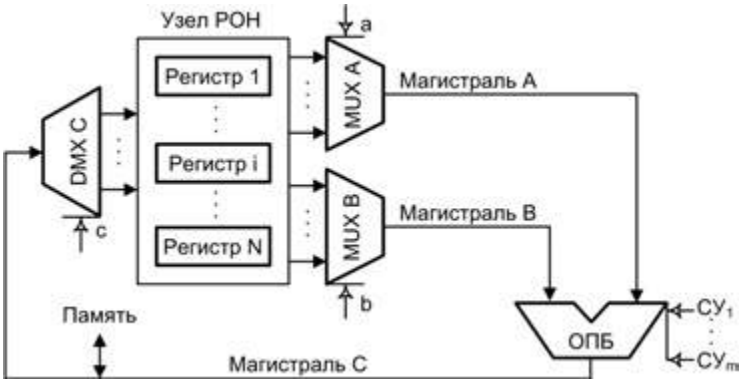


Рис. 5.3. Магистральное операционное устройство

Пример магистрального ОПУ представлен на рис. 5.3. В состав узла РОН здесь входят N регистров общего назначения, подключаемых к магистралям A и B через мультиплексоры MUX A и MUX B. Каждый из мультиплексоров является управляемым коммутатором, соединяющим выход одного из РОН с соответствующей магистралью. Номер подключаемого регистра определяется адресом a или b, подаваемым на адресные входы мультиплексора из устройства управления.

По магистралям A и B операнды поступают на входы операционного блока, к которым подключается комбинационная схема, реализующая требуемую микрооперацию (по сигналу управления из УУ). Таким образом, любая микрооперация ОПБ может быть

[1] В операционных устройствах для обработки чисел с плавающей запятой вместо РОН часто используется отдельный узел регистров с плавающей запятой.

выполнена над содержимым любых регистров ОПУ. Результат микрооперации по магистрали C через демультиплексор DMX C заносится в конкретный регистр узла РОН. Демультиплексор представляет собой управляемый коммутатор, имеющий один информационный вход и N информационных выходов. Вход подключается к выходу с заданным адресом c. Адрес c поступает на адресные входы DMX C из УУ.

Моделью ОПУ с магистральной структурой является M-автомат. M-автоматом называется модель ОПУ, построенная на основе принципа объединения комбинационных схем и реализующая в каждом такте только одну микрооперацию. Синтез M-автоматов рассматривается в [17].

Используя обозначения, введенные в предыдущем разделе, выражение для оценки аппаратных затрат на магистральное ОПУ можно записать в следующем виде:

$$C_{\text{М}} = nC_{\text{Т}}N + 3n(N + K) + n\sum_{j=1}^{K} C_j,$$

где первое слагаемое определяет затраты на N регистров, второе — затраты на связи узла РОН и ОПБ, а третье — суммарную цену ОПБ.

Из сопоставления выражений для затрат следует, что магистральная структура экономичнее жесткой структуры, если

$$3(N + K - M) < \sum_{i=1}^{N}\sum_{j=1}^{K} C_j K_{ij} - \sum_{j=1}^{K} C_j,$$

где $M = \sum_{i=1}^{N}\sum_{j=1}^{K} K_{ij}$ — количество микроопераций, реализуемых ОПУ с жесткой структурой.

С учетом последнего неравенства можно сформулировать следующее сильное условие экономичности магистральных структур:

$$M > N + K.$$

Затраты времени на сложение в магистральных ОПУ больше, чем в ОПУ с жесткой структурой:

$$T_{\text{М}} = t_{\text{В}} + t_{\text{С}} + t_{\text{П}} + t_{\text{MUX}} + t_{\text{DMX}} = t_{\text{Ж}} + t_{\text{MUX}} + t_{\text{DMX}},$$

где t_{MUX} — задержка на подключение операндов из РОН к ОПБ; t_{DMX} — задержка на подключение результата к РОН.

Основным достоинством магистральных ОПУ является высокая универсальность и регулярность структуры, что облегчает их реализацию на кристаллах ИС. Вообще говоря, магистральная структура ОПУ в современных ВМ является превалирующей.

Классификация операционных устройств с магистральной структурой

Магистральные ОПУ классифицируют по виду и количеству магистралей, организации узла регистров общего назначения, типу операционного блока.

Магистрали могут быть однонаправленными и двунаправленными, соответственно обеспечивающими передачу данных в одном или двух различных направлениях. Типичным режимом работы магистрали является разделение времени, при котором в различные моменты времени магистраль используется для передачи функционально разнотипных данных.

По функциональному назначению выделяют:

- *магистрали внешних связей*, соединяющих ОПУ с памятью и каналами ввода/вывода ВМ;
- *внутренние магистрали ОПУ*, отвечающие за связь между узлом РОН и операционным блоком.

Количество магистралей внешних связей зависит от архитектуры конкретной ВМ и обычно не превышает двух для внешних связей и трех — для внутренних.

Структура трехмагистрального ОПУ (3М) представлена на рис. 5.4, *а*, а соответствующая ему микропрограмма выполнения операции типа «сложение» — на рис. 5.4, *б*.

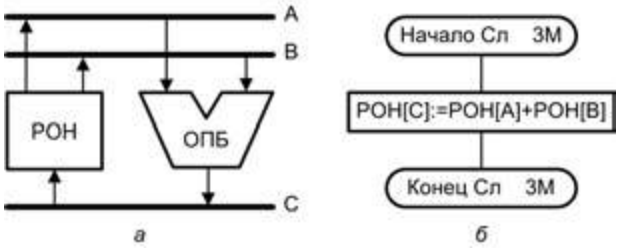


Рис. 5.4. Трехмагистральное ОПУ: *а* — структура; *б* — микропрограмма сложения

Данный вариант характеризуется наибольшим быстродействием: выборка операндов из РОН, выполнение микрооперации суммирования и запись результата в РОН производятся за один такт. Основной недостаток трехмагистральной организации — относительно большая площадь, занимаемая магистралями на кристалле интегральной микросхемы (ИМС).

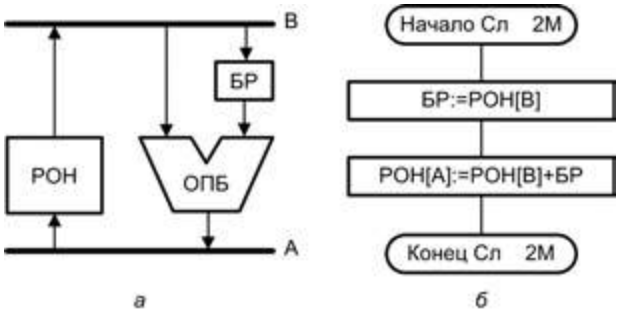


Рис. 5.5. Двухмагистральное ОПУ: *а* — структура; *б* — микропрограмма сложения

Двухмагистральная организация (2М) при меньшей площади, покрываемой магистралями, требует введения как минимум одного буферного регистра (БР), предназначенного для временного хранения одного из операндов (рис. 5.5, *а*), при этом операция сложения будет выполняться уже за два такта (рис. 5.5, *б*):

- Такт 1: загрузка БР одним из операндов.
- Такт 2: выполнение микрооперации в ОПБ над содержимым БР и одного из РОН; запись результата в РОН.

Наконец, организация ОПУ на основе только одной магистрали минимизирует расходы площади ИМС (рис. 5.6, *а*).

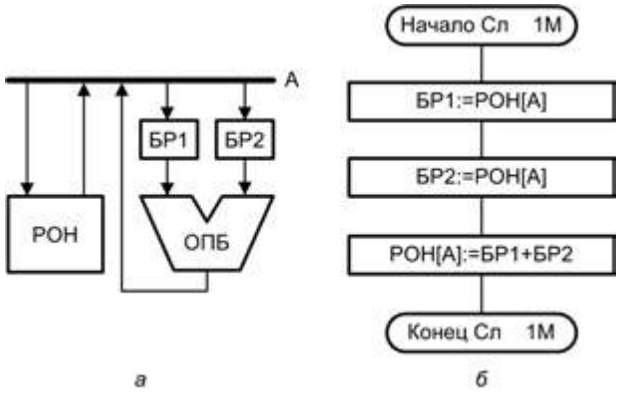


Рис. 5.6. Одномагистральное ОПУ: *а* — структура; *б* — микропрограмма сложения

В одномагистральном ОПУ (1М), вместе с тем, возникает необходимость введения не менее двух буферных регистров БР1, БР2, и длительность операции возрастает до трех тактов (рис. 5.6, *б*):

- Такт 1: загрузка БР1 одним из операндов.
- Такт 2: загрузка БР2 вторым операндом.
- Такт 3: выполнение микрооперации в ОПБ над содержимым БР1 и БР2; запись результата в один из РОН.

Организация узла РОН магистрального операционного устройства

Количество регистров в узле РОН магистрального ОПУ обычно превышает тот минимум, который необходим для реализации универсальной системы операций. Избыток регистров используется:

- для хранения составных частей адреса (индекса, базы);
- в качестве буферной, сверхоперативной памяти для повышения производительности ВМ за счет уменьшения требуемых пересылок между основной памятью и ОПУ.

Количество регистров колеблется в среднем от 8 до 16, иногда может достигать 32–64. В процессорах с сокращенным набором команд количество РОН доходит до нескольких сотен.

Организация узла РОН может обеспечивать одноканальный или двухканальный доступ как по входу (записи), так и по выходу (считыванию). В первом случае ко входу узла подключается один демультиплексор, а к выходу — один мультиплексор. Во втором случае доступ осуществляется с помощью двух демультиплексоров и (или) двух мультиплексоров. Двухканальный доступ повышает быстродействие ОПУ, так как позволяет обратиться параллельно к двум регистрам.

Организация операционного блока магистрального операционного устройства

Тип операционного блока (ОПБ) определяется способом обработки данных. Различают ОПБ последовательного и параллельного типа.

В *последовательном операционном блоке* (рис. 5.7) операции выполняются разряд за разрядом.

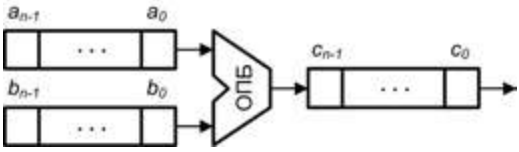


Рис. 5.7. Последовательный операционный блок

Бит переноса, возникающий при обработке i-го разряда операндов, подается на вход ОПБ и учитывается при обработке $(i + 1)$-го разряда операндов. Результат поразрядно заносится в выходной регистр, предыдущее содержимое которого перед этим сдвигается на одну позицию. Таким образом, после n циклов в выходном регистре формируется слово результата, где каждый разряд занимает предназначенную для него позицию.

При *параллельной организации операционного блока* (рис. 5.8) все разряды операндов обрабатываются одновременно. Внутренние переносы обеспечиваются схемой ОПБ.

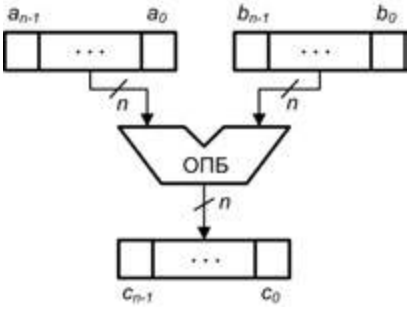


Рис. 5.8. Параллельный операционный блок

Реализация эффективной системы переносов в рамках «длинного» слова сопряжена с определенными аппаратурными издержками, поэтому на практике часто

используют параллельно-последовательную схему ОПБ. В ней слово разбивается на группы по 2, 4 или 8 разрядов, обработка всех разрядов внутри группы осуществляется параллельно, а сами группы обрабатываются последовательно.

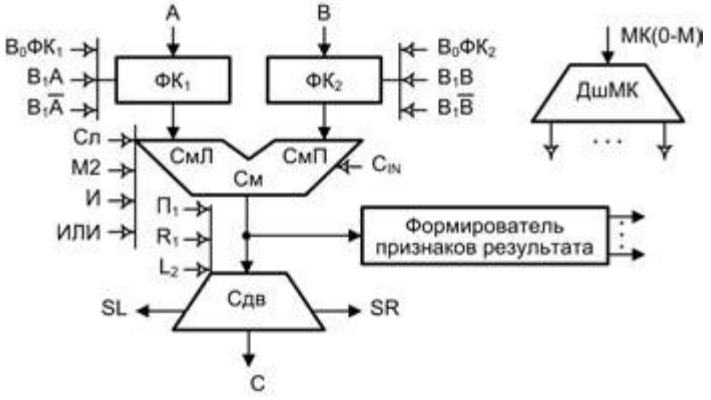


Рис. 5.9. Обобщенная схема операционного блока

Обобщенная схема ОПБ приведена на рис. 5.9. В нее входят: дешифратор микрокоманды ДшМК, формирователи кодов $ФК_1$ и $ФК_2$, многофункциональный сумматор См, сдвигатель Сдв и формирователь признаков результата (ФПР). Набор микроопераций, реализуемых ОПБ, выбран таким образом, чтобы обеспечить выполнение основных арифметических и логических операций, предусмотренных системой команд ВМ.

Дешифратор микрокоманды вырабатывает внутренние сигналы управления для элементов ОПБ. Он введен в схему с целью минимизации количества связей, требуемых для передачи сигналов управления из УУ.

Формирователи кодов $ФК_1$ и $ФК_2$ служат для формирования прямых и инверсных кодов операндов, поступающих по магистралям A и B. Они реализуют следующий набор микроопераций:

$B_0ФК_1$: СмЛ := 0; $B_0ФК_2$: СмП := 0;
B_1A: СмЛ := A; B_1B: СмП := B;
$B_1\overline{A}$: СмЛ := $\overline{A}$; $B_1\overline{B}$: СмП := $\overline{B}$.

Многофункциональный сумматор выполняет микрооперации арифметического сложения (с учетом переноса C_{IN}), сложения по модулю два, логического сложения и логического умножения кодов на левом и правом входах:

Сл: См := СмЛ + СмП + C_{IN};
М2: См := СмЛ $\oplus$ СмП;
И: См := СмЛ $\wedge$ СмП;
ИЛИ: См := СмЛ $\vee$ СмП;

Формирователь признаков результата на основе анализа кода на выходе См вырабатывает значения осведомительных сигналов (признаков результата), передаваемых

в УУ машины. Осведомительными сигналами могут быть: признак знака S, признак переполнения V, признак нулевого значения результата Z и т. п.

Сдвигатель служит для выполнения микроопераций сдвига кода на выходе См:

$П_1$: C := См;
R_1: C := R1(SL • См), SR := См(n);
L_1: C := L1(См • SR), SL := См(0).

Микрооперация $П_1$ обеспечивает передачу результата на магистраль C без сдвига. По ходу микрооперации R_1 результат сдвигается на один разряд вправо, при этом в освобождающийся старший разряд заносится значение с внешнего контакта SL, а выдвигаемый (младший) разряд сумматора посылается на внешний контакт SR.

В микрооперации L_1 результат сдвигается на один разряд влево. Здесь в освобождающийся младший разряд заносится значение с внешнего контакта SR, а выдвигаемый (старший) разряд См передается на внешний контакт SL.

Вспомогательные системы счисления, используемые в операционных устройствах

Хотя стандартная двоичная система в фон-неймановских ВМ, безусловно, является основной, при построении операционных устройств определенную пользу может принести временный переход к иным системам счисления. Прежде всего, это касается ОПУ для умножения и деления, где использование промежуточных систем счисления позволяет существенно сократить время выполнения данных операций. Естественно, что по завершении операции ее результат представляется в стандартной двоичной системе. В роли вспомогательных обычно выступают *избыточные системы счисления* и *системы счисления с основанием, кратным целой степени числа 2*, либо их сочетание.

Избыточные системы счисления

В отличие от обычной позиционной системы, где цифра числа может принимать значения в диапазоне от 0 до $q - 1$ (q — основание системы счисления), в избыточных системах цифра может иметь более чем q значений, например иметь знак. Если в стандартной двоичной системе счисления значение цифры ограничено множеством {0, 1}, то в знакоцифровой системе с тем же основанием $q = 2$ возможное значение цифры определяется множеством {–1, 0, 1}. Использование в ВМ знакоцифровых систем было предложено в 1961 году А. Авизиенисом [49].

Число в избыточной системе может быть записано несколькими способами (по этой причине такие системы и называют избыточными). Например, пятиразрядное двоичное представление числа одиннадцать 01011 (8 + 2 + 1) в избыточной системе с основанием 2 может иметь три представления: 01011 (8 + 2 + 1), $10\bar{1}0\bar{1}$ (16 – 4 – 1) и $0110\bar{1}$ (8 + 4 – 1)[1].

[1] Отрицательное значение цифры принято обозначать горизонтальной чертой над цифрой.

Промежуточный переход к избыточной системе счисления с последующим возвратом к стандартной двоичной системе позволяет создавать эффективные ОПУ, особенно для тех арифметических операций, реализация которых носит итеративный характер. Так, стандартное умножение на двоичное число 11111111 предполагает до 8 сложений (по числу разрядов множителя), в то время как при записи этого числа в избыточной системе с $q = 2$ $(10000000\ \bar{1})$ можно обойтись лишь одним вычитанием и одним сложением.

Системы счисления с основанием, кратным целой степени 2

В основе операционных устройств умножения и деления лежит итеративное выполнение операций сложения и сдвига. Количество итераций в общем случае равно разрядности операндов. Сокращение числа итераций — наиболее очевидный путь ускорения работы соответствующих ОПУ. Такое сокращение возможно за счет перехода к системам счисления с большим основанием, причем наиболее удобны в этом смысле системы с основанием q, кратным целой степени числа 2, в частности: $q = 4$ (Radix-4), $q = 8$ (Radix-8), $q = 16$ (Radix-16). Одна цифра в таких системах эквивалентна 2, 3 и 4 двоичным цифрам соответственно. Это позволяет, например, при выполнении операции умножения за один раз анализировать не один разряд множителя, а сразу 2, 3 или 4 и, соответственно, сократить общее число итераций, требуемых для анализа всех разрядов множителя.

Избыточные системы счисления с основанием, кратным целой степени 2

В ОПУ умножения и деления широкое распространение получили знакоцифровые системы счисления у которых основание кратно степени числа 2, а именно 2, 4, 8 и 16. В соответствии со значением q их обычно принято обозначать как Radix-2 ($q = 2$), Radix-4 ($q = 4$), Radix-8 ($q = 8$) и Radix-16 ($q = 16$). В таких системах счисления используются цифры в диапазоне от $-(q-1)$ до $+(q-1)$. Так, Radix-8 предполагает использования цифр из множества $\{-7, -6, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7\}$. Сочетание избыточности и большого основания системы счисления дает дополнительный эффект в сокращении числа итераций при выполнении операций умножения и деления, а значит, и в сокращении времени выполнения этих операций.

Естественно, что переход к вспомогательным системам счисления порождает определенное усложнение аппаратных средств ОПУ, но чаще всего это окупается выигрышем в быстродействии.

Операционные устройства для чисел в форме с фиксированной запятой

В форме с фиксированной запятой (ФЗ) могут быть представлены как числа без знака, когда все n позиций числа отводятся под значащие цифры, так и со знаком. В последнем случае старший $(n-1)$-й разряд числа занимает знак числа (0 — плюс, 1 — минус), а под значащие цифры отведены разряды с $(n-2)$-го по 0-й.

В большинстве современных ВМ форма с ФЗ используется для представления целых чисел, когда запятая фиксируется справа от младшего 0-го разряда кода числа (рис. 5.10). Соответствующие операционные устройства называют целочисленными ОПУ.

При выполнении арифметических операций над числами в форме с плавающей запятой (ПЗ) мантиссы операндов также рассматриваются как числа в форме с ФЗ. В этом случае мантиссы — это не целые числа, а правильные дроби, и в них запятая располагается перед $(n-1)$-м разрядом для чисел без знака, либо между $(n-1)$-м и $(n-2)$-м разрядами в случае чисел со знаком (рис. 5.11).

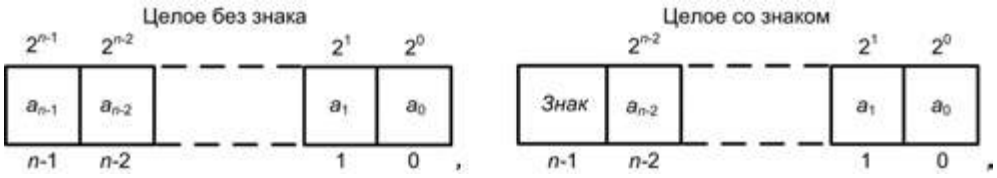


Рис. 5.10. Представление целых чисел в форме с фиксированной запятой

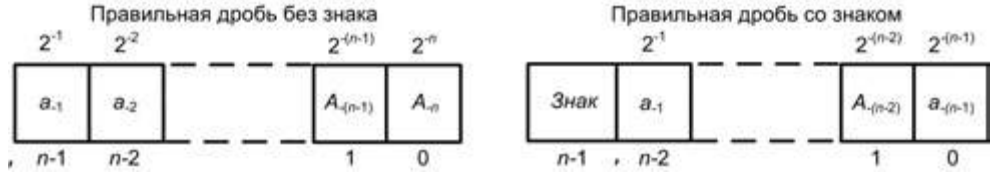


Рис. 5.11. Представление правильных дробей в форме с фиксированной запятой

При выполнении операций сложения и вычитания различие между целыми и дробными числами практически никак не сказывается[1].

Числа в форме с фиксированной запятой обычно представляются в дополнительном коде. В этом же коде должен быть представлен и результат операции.

Если исключить логические операции, которые рассматриваются отдельно, ОПУ для обработки чисел в форме с ФЗ должно обеспечивать сложение, вычитание, умножение и деление чисел без знака и со знаком.

Сложение и вычитание

Обе операции рассматриваются применительно к числам со знаком. Поскольку $(n-1)$-й разряд, хранящий знак, обрабатывается наравне со значащими цифрами, все нижесказанное справедливо и для чисел без знака, где $(n-1)$-й разряд занимает старшая цифра числа.

На рис. 5.12 приводятся примеры сложения целых чисел.

[1] Определенные нюансы могут возникать лишь при расположении результата умножения, имеющего двойную длину, в разрядной сетке двойного слова.

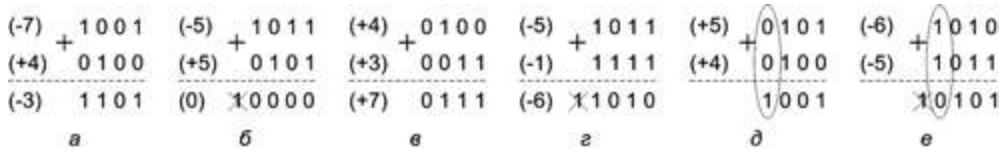


Рис. 5.12. Примеры выполнения операции сложения в дополнительном коде: *а*, *б*, *в*, *г* — сложение без возникновения переполнения; *д*, *е* — сложение с переполнением

Вычитание сводится к сложению и выполняется в соответствии с правилом: *для вычитания одного числа* (*вычитаемого*) *из другого* (*уменьшаемого*) *необходимо взять дополнение вычитаемого и прибавить его к уменьшаемому*. Под дополнением здесь понимается вычитаемое с противоположным знаком. Вычитание иллюстрируется примерами (рис. 5.13).

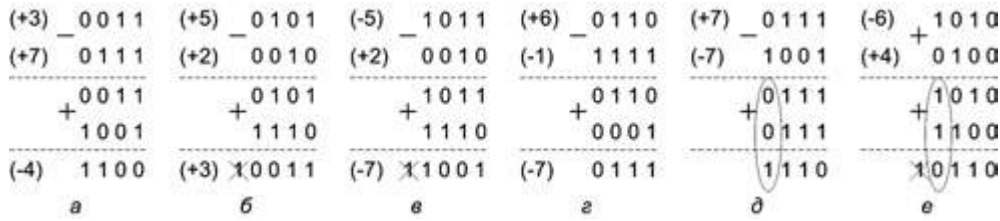


Рис. 5.13. Примеры выполнения операции вычитания в дополнительном коде: *а*, *б*, *в*, *г* — вычитание без возникновения переполнения; *д*, *е* — вычитание с переполнением

Как при сложении, так и при вычитании двоичных чисел со знаком возможен результат, старшая значащая цифра которого занимает позицию знака (см. рис. 5.12, *д*, *е* и 5.13, *в*, *г*). Ситуация известна как *переполнение*, а условие его возникновения можно сформулировать следующим образом: *если суммируются два числа и они оба положительные или оба отрицательные, переполнение имеет место тогда и только тогда, когда знак результата противоположен знаку слагаемых*. ОПУ должно выявлять факт переполнения и сигнализировать о нем. Обратим внимание, что переполнение не всегда сопровождается переносом из знакового разряда.

Чтобы упростить обнаружение ситуации переполнения, часто применяется так называемый *модифицированный дополнительный код*, когда для хранения знака отводятся два разряда, причем оба участвуют в арифметической операции наравне с цифровыми разрядами. В нормальной ситуации оба знаковых разряда содержат одинаковые значения. Различие в содержимом знаковых разрядов служит признаком возникшего переполнения (рис. 5.14).

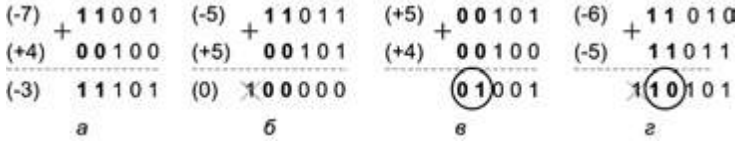


Рис. 5.14. Примеры выполнения операции сложения в модифицированном дополнительном коде: *а*, *б* — переполнения нет; *в*, *г* — возникло переполнение

На рис 5.15 показана возможная структура операционного блока для сложения и вычитания чисел со знаком в форме с фиксированной запятой. Центральным звеном устройства является n-разрядный двоичный сумматор. Операнд A поступает на вход сумматора без изменений. Операнд B предварительно пропускается через схемы сложения по модулю 2, поэтому вид кода B, поступающего на другой вход сумматора, зависит от выполняемой операции. Если задана операция сложения (управляющий код 0), то результат на выходе ОПБ определяется выражением $S = A + B$. При операции вычитания (управляющий код 1) на вход сумматора подаются инверсные значения всех разрядов B, и, кроме того, на вход переноса в младший разряд сумматора C_{IN} поступает 1. В итоге на выходе ОПБ будет $S = A + \overline{B} + 1 = A + (-B) = A - B$, что соответствует прибавлению к A числа B с противоположным знаком, то есть вычитанию.

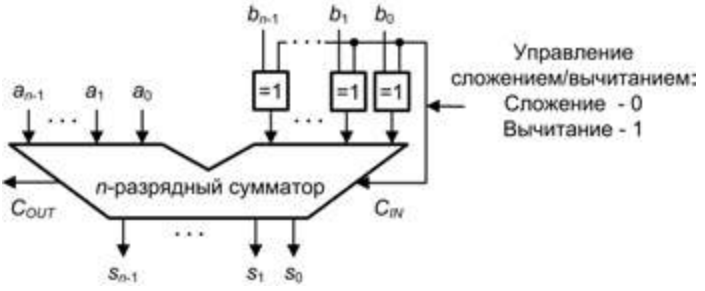


Рис. 5.15. Структура операционного блока для сложения и вычитания чисел в форме с фиксированной запятой

На рис. 5.15 не показана схема формирования признака переполнения V, который согласно описанным ранее правилам определяется логическим выражением:
$V = \overline{a_{n-1}} \wedge \overline{b_{n-1}} \wedge s_{n-1} \vee a_{n-1} \wedge b_{n-1} \wedge \overline{s_{n-1}}$.

Умножение

По сравнению со сложением и вычитанием, умножение — более сложная операция, как при программной, так и при аппаратной реализации. В ВМ применяются различные алгоритмы умножения и, соответственно, различные схемы построения операционных блоков умножения.

Традиционная схема умножения похожа на известную из школьного курса процедуру записи «в столбик». Вычисление произведения P ($p_{2n-1}\, p_{2n-2}$ ... $p_1 p_0$) двух n-разрядных двоичных чисел без знака A ($a_{n-1}a_{n-2}$ ... a_1a_0) и B ($b_{n-1}b_{n-2}$ ... b_1b_0) сводится к формированию частичных произведений (ЧП) P'_i, по одному на каждую цифру множителя, с последующим суммированием полученных ЧП. Перед суммированием каждое частичное произведение должно быть сдвинуто на один разряд относительно предыдущего согласно весу цифры множителя, которой это ЧП соответствует. Поскольку в ВМ операндами являются двоичные числа, вычисление ЧП упрощается — если цифра множителя b_i равна 0, то P'_i тоже равно 0, а при $b_i = 1$ частичное произведение равно множимому ($P'_i = A$). Перемножение двух

n-разрядных двоичных чисел $P = A \times B$ приводит к получению результата, содержащего $2n$ битов. Алгоритм умножения предполагает последовательное выполнение двух операций — сложения и сдвига (рис. 5.16). Суммирование частичных произведений обычно производится не на завершающем этапе, а по мере их получения. Это позволяет избежать необходимости хранения всех ЧП, то есть сокращает аппаратурные издержки. Согласно данной схеме, устройство умножения предполагает наличие регистров множимого, множителя и суммы частичных произведений, а также сумматора ЧП и, возможно, схем сдвига, если операция сдвига не реализована иным способом, например за счет «косой» передачи данных между узлами умножителя.

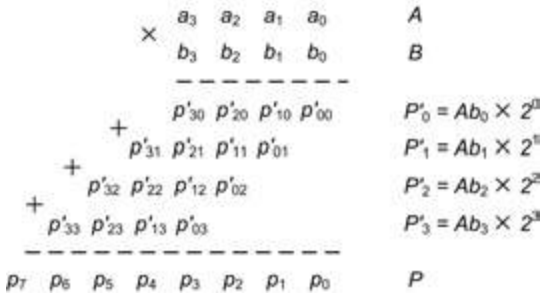


Рис. 5.16. Общая схема умножения

В зависимости от того, с какого разряда начинается анализ цифр множителя (старшего или младшего) и что сдвигается в процессе умножения (множимое или сумма частичных произведений), возможны четыре варианта реализации «традиционной» схемы умножения [10]. Из них наибольшее распространение получил метод умножения, начиная с младших разрядов множителя и со сдвигом суммы частичных произведений вправо при неподвижном множимом. Данный вариант позволяет построить ОПУ, в котором все элементы (регистры множимого, множителя и суммы частичных произведений, а также сумматор) имеют разрядность n и не требуются элементы с разрядностью $2n$.

Умножение чисел без знака

Общую процедуру традиционного умножения сначала рассмотрим применительно к числам без знака, то есть таким числам, в которых все n разрядов представляют значащие цифры.

Алгоритм сводится к следующим шагам.

1. Исходное значение суммы частичных произведений (СЧП) принимается равным нулю.
2. Анализируется очередная цифра множителя (анализ начинается с младшей цифры). Если она равна единице, то к СЧП прибавляется множимое, в противном случае (цифра множителя равна нулю) прибавление не производится.
3. Выполняется сдвиг суммы частичных произведений вправо на один разряд.
4. Пункты 2 и 3 последовательно повторяются для всех разрядов множителя.

Процедуру умножения иллюстрирует пример вычисления произведения 10×11 (рис. 5.17).

	$A = a_3a_2a_1a_0$	× 1 0 1 0	Множимое 10_{10}
	$B = b_3b_2b_1b_0$	1 0 1 1	Множитель 11_{10}
Исходное значение СЧП равно 0		0 0 0 0	
Вычисление ЧП для цифры b_0		+ 1 0 1 0	
Прибавление ЧП к СЧП		0 1 0 1 0	
Сдвиг СЧП вправо на 1 разряд		⟹ 0 1 0 1 0	
Вычисление ЧП для цифры b_1		+ 1 0 1 0	
Прибавление ЧП к СЧП		0 1 1 1 1 0	
Сдвиг СЧП вправо на 1 разряд		⟹ 0 1 1 1 1 0	
Вычисление ЧП для цифры b_2		+ 0 0 0 0	
Прибавление ЧП к СЧП		0 0 1 1 1 1 0	
Сдвиг СЧП вправо на 1 разряд		⟹ 0 0 1 1 1 1 0	
Вычисление ЧП для цифры b_3		+ 1 0 1 0	
Прибавление ЧП к СЧП		0 1 1 0 1 1 1 0	
Сдвиг СЧП вправо на 1 разряд	$P = p_7p_6p_5p_4p_3p_2p_1p_0$	⟹ 0 1 1 0 1 1 1 0	Произведение 110_{10}

Рис. 5.17. Пример умножения со сдвигом суммы частичных произведений вправо при неподвижном множимом

Алгоритм может быть реализован с помощью схемы, показанной на рис. 5.18.

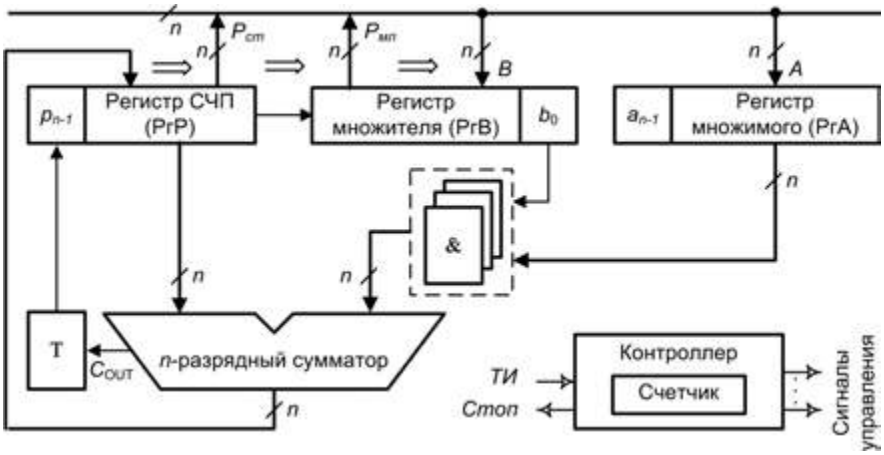


Рис. 5.18. Схема устройства умножения чисел без знака

Первоначально множимое (A) и множитель (B) заносятся в n-разрядные регистры множимого (РгА) и множителя (РгВ) соответственно, а все разряды n-разрядного регистра суммы частичных произведений (РгР) устанавливаются в 0. В счетчик циклов записывается число, равное разрядности множителя (n). Умножение происходит за n шагов. На каждом шаге n схемами «И» (по одной схеме на каждый разряд множимого) на правый вход n-разрядного сумматора подается либо множимое, либо 0. На левый вход поступает содержимое РгР. При суммировании перенос из

старшего разряда сумматора запоминается в триггере Т. Новая сумма частичных произведений из сумматора пересылается в РгР. Регистры РгР и РгВ связаны между собой так, что в операциях сдвига их можно рассматривать как один $2n$-разрядный сдвиговый регистр. В продолжение процедуры умножения содержимое РгР и РгВ совместно сдвигается на один разряд вправо. При этом в освободившийся при сдвиге старший разряд РгР из триггера Т заносится значение переноса C_{OUT} из старшего разряда сумматора. В свою очередь, освободившийся при сдвиге старший разряд РгВ заполняется битом, «вытолкнутым» из РгР. В процессе сдвига теряется уже проанализированный младший разряд множителя, а на его место в РгВ поступает очередной разряд. Далее содержимое счетчика уменьшается на 1. Описанная последовательность повторяется n раз. Дополнительно отметим, что если очередная цифра множителя равна 1, то для вычисления СЧП требуются операции сложения и сдвига, а при нулевой цифре множителя, в принципе, можно обойтись без сложения, ограничившись только сдвигом.

Процедура умножения синхронизируется тактовыми импульсами (ТИ) и завершается при достижении нулевого значения в счетчике циклов. В этот момент контроллер устройства формирует сигнал *Стоп*. Результат умножения P будет располагаться в регистрах РгР и РгВ (старшие разряды *Рст* — в РгР, а младшие разряды *Рмл* — в РгВ).

Умножение чисел со знаком

Несколько сложнее обстоит дело с умножением чисел со знаком, когда n-разрядные сомножители содержат знак (в старшем разряде слова) и n–1 значащую цифру. В принципе, задачу можно решить путем перемножения абсолютных значений сомножителей как чисел без знака с последующим учетом знаков сомножителей. С другой стороны, числа с фиксированной запятой в вычислительных машинах обычно представлены в дополнительном коде[1] и логичнее выполнять умножение непосредственно в этом коде. Такую возможность предоставляет *алгоритм Робертсона* [114]. В сущности, он аналогичен рассмотренному ранее алгоритму для чисел без знака, но учитывает особенности, обусловленные дополнительным кодом.

Первая особенность связана с тем, что в процессе умножения могут получаться как положительные, так и отрицательные частичные произведения (ЧП). Для корректного сдвига ЧП такой сдвиг должен быть арифметическим. Это означает, что при сдвиге ЧП вправо знаковый разряд не сдвигается, а освобождающаяся позиция слева должна заполняться значением знакового разряда, то есть при положительном ЧП — нулем, а при отрицательном — единицей.

Вторая проблема возникает при умножении на отрицательный множитель. Так как множитель отрицателен, он записывается в преобразованном виде $[B]_{\text{д}}$, и в результате анализа разрядов такого множителя вместо истинного произведения P получаем

[1] Положительные числа в этом представлении не отличаются от записи в прямом коде, а отрицательные записываются в виде $2^n - x$, где x — фактическое значение числа. В двоичной системе запись отрицательного числа в дополнительном коде сводится к инвертированию всех цифровых разрядов числа, представленного в прямом коде, и прибавлению единицы к младшему разряду обратного кода, получившегося после инвертирования.

псевдопроизведение P' отличное от P. В случае целых чисел $[B]_д = 1b_{n-2}... b_0$ с битом знака $b_{n-1} = 1$. Тогда

$$|B| = 2^n - [B]_д = 1(00...0) - (1b_{n-2}...b_0) = 1(00...0) - (10...0) - (b_{n-1}...b_0) =$$

$$= (10...0) - (b_{n-1}...b_0) = 2^{n-1} - \sum_0^{n-2} b_i,$$

откуда $[B]_д = -|B| = -2^{n-1} + \sum_0^{n-2} b_i$.

В алгоритме Робертсона это учитывается следующим образом. Прежде всего, знаковый разряд участвует в операции наряду с цифровыми разрядами. Анализ цифр множителя от младшей значащей b_0 до старшей значащей b_{n-2}, а также обусловленные этим анализом сложения и сдвиги ведутся стандартным образом. Эти операции представлены вторым членом выражения для $[B]_д$. Позиция b_{n-1} содержит знак и для отрицательного множителя равна 1. В стандартной процедуре это предполагает прибавление к СЧП множимого с весом 2^{n-1}. Вместо этого на шаге анализа знакового разряда множимое, которое к этому моменту имеет вес 2^{n-1}, не прибавляется к сумме частичных произведений, а вычитается из нее. Обратим внимание, что такая коррекция нужна лишь в случае отрицательного множителя.

Подводя итог, алгоритм умножения чисел со знаком можно описать следующим образом:

1. Отрицательные сомножители представляются дополнительным кодом[1].
2. Исходное значение суммы частичных произведений принимается равным нулю.
3. Анализируется очередная значащая цифра множителя (анализ начинается с младшей цифры). Если она равна единице, то к СЧП прибавляется множимое в том коде, в котором оно представлено. Если анализируемая цифра множителя равна нулю, прибавление не производится.
4. Выполняется арифметический сдвиг суммы частичных произведений вправо на один разряд (освободившаяся при сдвиге позиция заполняется значением знакового разряда СЧП).
5. Пункты 3 и 4 последовательно повторяются для всех цифровых разрядов множителя.
6. Анализируется знаковый разряд множителя. Если он равен 1 (отрицательный множитель), то из СЧП вычитается множимое (прибавляется множимое с обратным знаком, представленное в дополнительном коде).
7. Если сомножители — целые числа, производится арифметический сдвиг СЧП вправо на один разряд (освободившаяся при сдвиге позиция заполняется значением знакового разряда СЧП). В случае перемножения правильных дробей дополнительный сдвиг не требуется.

При разных знаках сомножителей произведение получается отрицательным и будет представлено в дополнительном коде.

[1] Представления положительных чисел одинаковы во всех кодах — прямом, обратном и дополнительном.

Примеры умножения чисел со знаком на основе алгоритма Робертсона приведены на рис. 5.19.

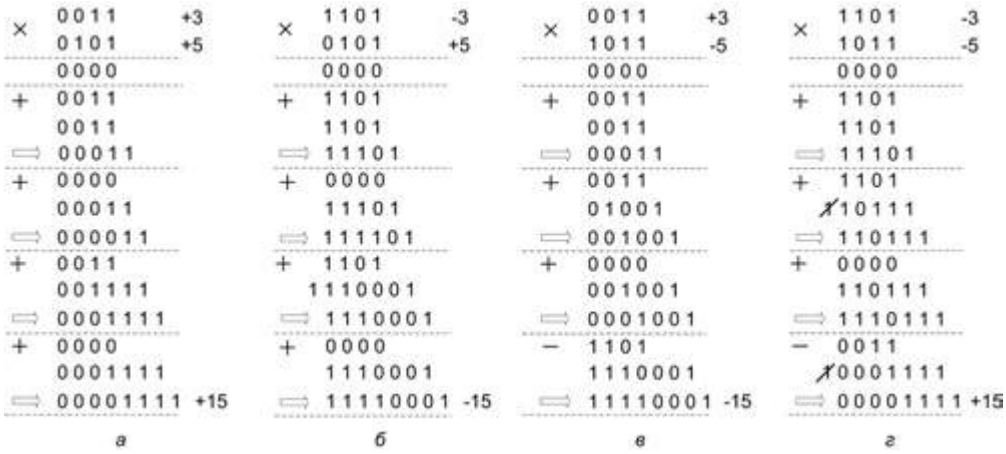


Рис. 5.19. Примеры умножения целых чисел по алгоритму Робертсона: *а*, *б* — при положительном множителе; *в*, *г* — при отрицательном множителе

На рис. 5.19, *в* и *г* вычитание множимого (коррекция псевдопроизведения) обозначено знаком «–», но в цифровой части показано как прибавление множимого с обратным знаком, которое в случае *в* отрицательно и представлено в дополнительном коде.

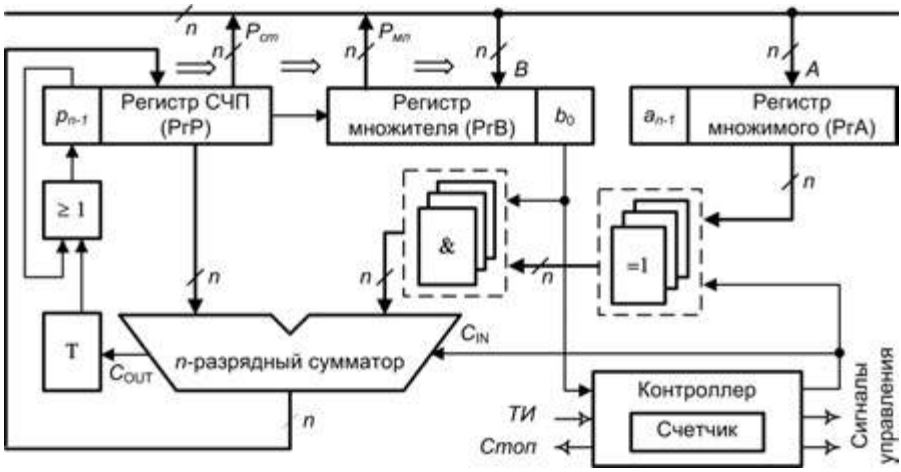


Рис. 5.20. Схема устройства умножения чисел со знаком

Умножение чисел со знаком, в принципе, может быть реализовано с помощью ранее рассмотренного устройства (см. рис. 5.18) при внесении в него некоторых изменений, показанных на рис. 5.20. Суть этих изменений сводится к введению в схему

n элементов «Исключающее ИЛИ», благодаря чему становится возможной операция вычитания множимого, требуемая на этапе коррекции результата. Кроме того, на входе знакового разряда регистра суммы частичных произведений добавлена схема «ИЛИ», с помощью которой обеспечивается правильное выполнение арифметического сдвига СЧП, если последняя отрицательна (в этом случае освобождающиеся при сдвиге разряды должны заполняться не нулями, а единицами).

Ускорение операции умножения

Методы ускорения умножения можно условно разделить на логические и аппаратные. Логические методы позволяют сократить время вычисления за счет более эффективных алгоритмов умножения, в частности за счет применения избыточных систем счисления и систем с основанием, которое больше двух. В аппаратных методах упор делается на схемное сокращение времени вычисления и суммирования частичных произведений. На практике оба этих подхода совмещают.

Умножение с использованием избыточных систем счисления

Наиболее известным и распространенным представителем этой группы логических методов ускорения операции умножения является алгоритм Бута.

Алгоритм Бута

В основе алгоритма Бута [56] лежит следующее соотношение, характерное для последовательности двоичных цифр:

$$2^m + 2^{m-1} + ... + 2^k = 2^{m+1} - 2^k,$$

где m и k — номера крайних разрядов в группе из последовательности единиц. Например, $011110 = 2^5 - 2^1$. Это означает, что при наличии в множителе групп из нескольких единиц (комбинаций вида 011110), последовательное добавление к СЧП множимого с нарастающим весом (от 2^k до 2^m) можно заменить вычитанием из СЧП множимого с весом 2^k и прибавлением к СЧП множимого с весом 2^{m+1}.

Алгоритм предполагает три операции: сдвиг, сложение и вычитание. Помимо сокращения числа сложений (вычитаний), у него есть еще одно достоинство — он в равной степени применим к числам без знака и со знаком.

Алгоритм Бута сводится к перекодированию множителя из обычной двоичной системы $\{0, 1\}$ в избыточную систему $\{\bar{1}, 0, 1\}$, из-за чего такое перекодирование часто называют перекодированием Бута (Booth recoding). В записи множителя в новой системе 1 означает добавление множимого к сумме частичных произведений, $\bar{1}$ — вычитание множимого, и 0 не предполагает никаких действий. Во всех случаях после очередной итерации производится сдвиг множимого влево или суммы частичных произведений вправо. Реализация алгоритма предполагает последовательный в направлении справа налево анализ пар разрядов множителя — текущего b_i и предшествующего b_{i-1} ($b_i b_{i-1}$). Для младшего разряда множителя ($i = 0$) считается, что предшествующий разряд равен 0, то есть имеет место пара $b_0 0$. На каждом шаге i ($i = 0, 1, ..., n-1$) анализируется текущая комбинация $b_i\, b_{i-1}$.

Комбинация 10 означает начало цепочки последовательных единиц, и в этом случае производится вычитание множимого из СЧП.

Комбинация 01 соответствует завершению цепочки единиц, и здесь множимое прибавляется к СЧП.

Комбинация 00 свидетельствует об отсутствии цепочки единиц, а 11 — о нахождении внутри такой цепочки. В обоих случаях никакие арифметические операции не производятся.

По завершении анализа и учета очередной пары разрядов множителя осуществляется сдвиг множимого влево либо суммы частичных произведений вправо, и цикл повторяется для следующей пары разрядов множителя.

Описанную процедуру рассмотрим на примерах. В них операция вычитания, как это принято в реальных умножителях, выполняется путем сложения с множителем, взятым с противоположным знаком и представленным в дополнительном коде. Напомним, что при сдвиге СЧП вправо в освободившиеся позиции заносится значение знакового разряда.

Пример 1. $0110 \times 0011 = 00010010$ (в десятичном виде $6 \times 3 = 18$). После перекодирования Бута множитель $\{0, 0, 1, 1\}$ приобретает вид $\{0, 1, 0, \bar{1}\}$.

Вначале сумма частичных произведений принимается равной нулю. Полагается, что младшему разряду множителя предшествовал 0. Дальнейший процесс поясняет рис. 5.21.

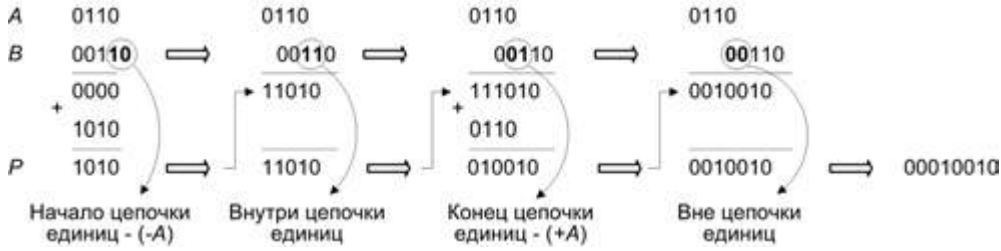


Рис. 5.21. Пример 1 умножения (6×3) в соответствии с алгоритмом Бута

Пример 2. $1100 \times 0011 = 11110100$ (в десятичной записи $-4 \times 3 = -12$).

Процесс вычисления иллюстрирует рис. 5.22.

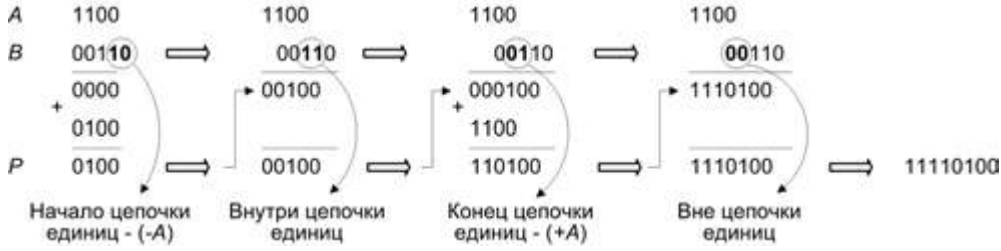


Рис. 5.22. Пример 2 умножения (-4×3) в соответствии с алгоритмом Бута

При наиболее благоприятном сочетании цифр множителя количество суммирований равно $n/2$, где n — число разрядов множителя.

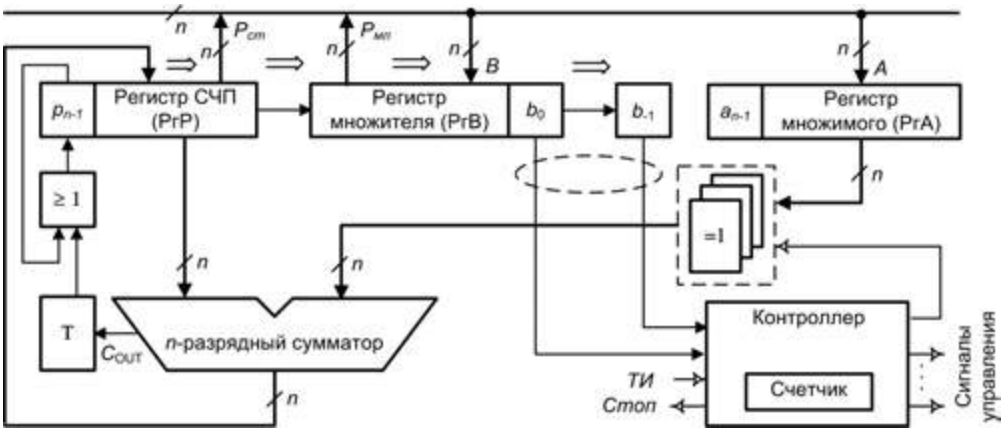


Рис. 5.23. Схема устройства умножения чисел со знаком по алгоритму Бута

Структура устройства умножения чисел со знаком по алгоритму Бута (рис. 5.23) напоминает ту, что используется для стандартного алгоритма. Поскольку на каждом шаге анализируются два разряда множителя, регистр РгВ удлинен на один разряд, в котором при сдвиге вправо сохраняется предыдущий бит множителя. Вначале процедуры умножения в этот дополнительный разряд заносится 0.

Модифицированный алгоритм Бута

На практике большее распространение получила модификация алгоритма Бута, где количество операций сложения при любом сочетании единиц и нулей в множителе всегда равно $n/2$. В модифицированном алгоритме производится перекодировка цифр множителя из стандартной двоичной системы {0, 1} в избыточную систему $\{\bar{2}, \bar{1}, 0, 1, 2\}$, где каждое число представляет собой коэффициент, на который умножается множимое перед добавлением к СЧП. Одновременно анализируются три разряда множителя $b_{i+1}b_ib_{i-1}$ (два текущих и старший разряд из предыдущей тройки) и, в зависимости от комбинации 0 и 1 в этих разрядах, выполняется прибавление или вычитание множимого, прибавление или вычитание удвоенного множимого либо никакие действия не производятся (табл. 5.1).

Таблица 5.1. Логика модифицированного алгоритма Бута

$\mathbf{x}_{i+1}$	$\mathbf{x}_i$	$\mathbf{x}_{i-1}$	**Код** $\{\bar{2}, \bar{1}, 0, 1, 2\}$,	**Выполняемые действия**
0	0	0	0	Не выполнять никаких действий
0	0	1	1	Прибавить к СЧП множимое
0	1	0	1	Прибавить к СЧП множимое
0	1	1	2	Прибавить к СЧП удвоенное множимое

продолжение ⇩

Таблица 5.1 (*продолжение*)

x_{i+1}	x_i	x_{i-1}	**Код** $\{\bar{2}, \bar{1}, 0, 1, 2\}$,	**Выполняемые действия**
1	0	0	$\bar{2}$	Вычесть из СЧП удвоенное множимое
1	0	1	$\bar{1}$	Вычесть из СЧП множимое
1	1	0	$\bar{1}$	Вычесть из СЧП множимое
1	1	1	0	Не выполнять никаких действий

Пример вычисления произведения $011001 \times 101110 = 011000111110$ (в десятичном виде $25 \times (-18) = -450$) показан на рис. 5.24. Использован алгоритм с неподвижной СЧП и сдвигом множимого влево.

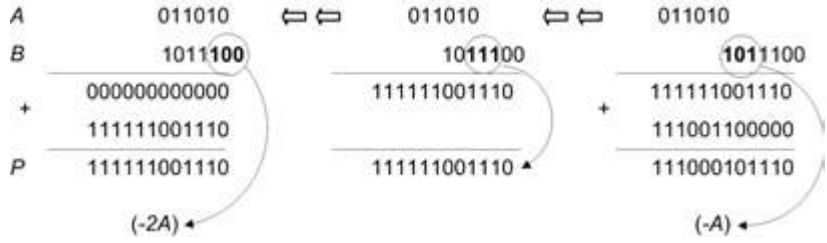


Рис. 5.24. Пример умножения $(18 \times (-25))$ в соответствии с модифицированным алгоритмом Бута

Алгоритм Лемана

Еще большее сокращение количества сложений может дать модификация, предложенная Леманом [108]. Здесь, даже при наименее благоприятном сочетании цифр множителя, количество операций сложения не превышает величины $n/2$, а в среднем же оно составляет $n/3$. Суть модификации заключается в следующем:

- если две группы нулей разделены единицей, стоящей в k-й позиции, то вместо вычитания в k-й позиции и сложения в $(k + 1)$-й позиции достаточно выполнить только сложение в k-й позиции;
- если две группы единиц разделены нулем, стоящим в k-й позиции, то вместо сложения в k-й позиции и вычитания в $(k + 1)$-й позиции достаточно выполнить только вычитание в k-й позиции.

Действия, выполняемые на i-м шаге умножения, можно описать с помощью следующих логических выражений:

$$d_i = (b_i \oplus b_{i-1})\ \overline{d_{i-1}};\quad s_i = d_i b_{i+1} \vee \overline{d_i} s_{i-1},$$

где b_i — цифра разряда множителя; d_i — двоичная переменная, единичное значение которой для соответствующего разряда множителя указывает на необходимость выполнения арифметического действия; s_i — знак арифметического действия.

При $d_i = 1$, $s_i = b_{i+1}$. Если $s_i = 0$, множимое прибавляется к СЧП, а при $s_i = 1$ — вычитается из СЧП.

Умножение в системах счисления с основанием, кратным целой степени 2

Из второй группы логических методов остановимся на умножении с обработкой за шаг нескольких разрядов множителя, то есть с представлением множителя в системе Radix-X. Рассмотрим это на примере двух разрядов, что соответствует представлению множителя в системе счисления с основанием 4.

Анализ множителя начинается с младших разрядов. В зависимости от входящей двухразрядной комбинации предусматриваются следующие действия:

- 00 — простой сдвиг на два разряда вправо суммы частичных произведений (СЧП);
- 01 — к СЧП прибавляется одинарное множимое, после чего СЧП сдвигается на 2 разряда вправо;
- 10 — к СЧП прибавляется удвоенное множимое, и СЧП сдвигается на два разряда вправо;
- 11 — из СЧП вычитается одинарное множимое, и СЧП сдвигается на два разряда вправо. Полученный результат должен быть скорректирован на следующем шаге, что фиксируется в специальном триггере признака коррекции.

Так как в случае пары 11 из СЧП вычитается одинарное множимое вместо прибавления утроенного, для корректировки результата к СЧП перед выполнением сдвига надо было бы прибавить учетверенное множимое. Но после сдвига на два разряда вправо СЧП уменьшается в четыре раза, так что на следующем шаге достаточно добавить одинарное множимое. Это учитывается при обработке следующей пары разрядов множителя, путем обработки пары 00 как 01, пары 01 как 10, 10 — как 11, а 11 — как 00. В последних двух случаях фиксируется признак коррекции.

Таблица 5.2. Формирование признака коррекции

Пара разрядов множителя	Признак коррекции из предыдущей пары	Признак коррекции в следующую пару	Знак действия	Кратность множимому
00	0	0		0
01	0	0	+	1
10	0	0	+	2
11	0	1	–	1
00	1	0	+	1
01	1	0	+	2
10	1	1	–	1
11	1	1		0

Правила обработки пар разрядов множителя с учетом признака коррекции приведены в табл. 5.2. После обработки каждой комбинации содержимое регистра множителя и регистра суммы частичных произведений сдвигается на два разряда вправо. Данный метод умножения требует корректировки результата, если старшая пара разрядов множителя равна 11 или 10 и состояние признака коррекции единичное. В этом случае к полученному произведению должно быть добавлено множимое.

Как следует из описания алгоритма, в случае $q = 4$ необходимо иметь частичные произведения $0 \times A$, $1 \times A$, $2 \times A$ и $3 \times A$. При $q = 8$ дополнительно необходимы частичные произведения $4 \times A$, $5 \times A$, $6 \times A$ и $7 \times A$. ЧП, представляющие собой произведение множимого на число, кратное степени двойки, например, $2 \times A$, $4 \times A$, легко могут быть получены путем сдвига множителя. Для учета остальных вариантов ЧП в состав устройства умножения включается специальная память, хранящая предвычисленные значения таких ЧП.

Аппаратные методы ускорения умножения

Помимо эффекта, достигаемого за счет логических методов ускорения умножения, дополнительные возможности открываются при аппаратной реализации ОПУ умножения. В общем случае аппаратные методы ускорения умножения сводятся к:

- параллельному вычислению частичных произведений;
- сокращению количества операций сложения;
- уменьшению времени распространения переносов при суммировании частичных произведений.

Остановимся на первой из перечисленных возможностей. Если рассмотреть общую схему умножения (рис. 5.25), то нетрудно заметить, что отдельные разряды ЧП представляют собой произведения вида $a_i b_j$, то есть произведение определенного бита множимого на определенный бит множителя. Это позволяет вычислить все биты частичных произведений одновременно, с помощью n^2 схем «И». При перемножении чисел в дополнительном коде отдельные разряды ЧП могут иметь вид $a_i \overline{b_j}$, $\overline{a_i} b_j$ или $\overline{a_i}\,\overline{b_j}$. Тогда элементы «И» заменяются элементами, реализующими соответствующую логическую функцию.

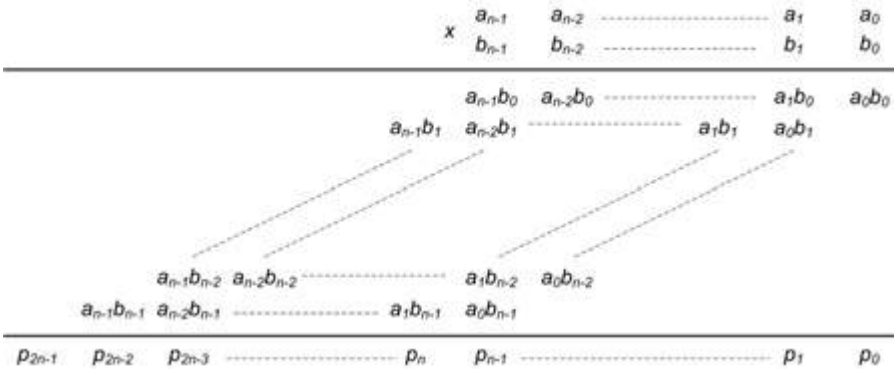


Рис. 5.25. Схема перемножения *n*-разрядных чисел без знака

Параллельное вычисление ЧП имеет место практически во всех рассматриваемых ниже схемах умножения. Различия проявляются в основном в способе суммирования полученных частичных произведений. С этих позиций используемые схемы умножения можно подразделить на *матричные* и *с древовидной структурой* [154]. В обоих вариантах суммирование осуществляется с помощью массива

взаимосвязанных одноразрядных сумматоров. В матричных умножителях сумматоры организованы в виде матрицы, а в древовидных они реализуются в виде дерева того или иного типа. Различия в рамках каждой из этих групп выражаются в количестве используемых сумматоров, их виде и способе распространения переносов, возникающих в процессе суммирования.

Матричные схемы умножения

В *матричных умножителях* суммирование осуществляется матрицей сумматоров, состоящей из последовательных линеек (строк) одноразрядных сумматоров с сохранением переноса (ССП). По мере движения данных вниз по массиву сумматоров каждая строка ССП добавляет к СЧП очередное частичное произведение. Поскольку промежуточные СЧП представлены в избыточной форме с сохранением переноса, во всех схемах, вплоть до последней строки, где формируется окончательный результат, распространения переноса не происходит. Это означает, что задержка в умножителях зависит только от «глубины» массива (числа строк сумматоров) и не зависит от разрядности операндов, если только в последней строке матрицы, где формируется окончательная СЧП, не используется схема с последовательным переносом.

Наряду с высоким быстродействием важным достоинством матричных умножителей является их регулярность, что особенно существенно при реализации таких умножителей в виде интегральной микросхемы. С другой стороны, подобные схемы занимают большую площадь на кристалле микросхемы, причем с увеличением разрядности сомножителей эта площадь увеличивается пропорционально квадрату числа разрядов. Вторая проблема с матричными умножителями — низкий уровень утилизации аппаратуры. По мере движения СЧП вниз каждая строка задействуется лишь однократно, когда ее пересекает активный фронт вычислений. Это обстоятельство, однако, может быть затребовано для повышения эффективности вычислений путем конвейеризации процесса умножения, при которой по мере освобождения строки сумматоров последняя может быть использована для умножения очередной пары чисел.

Ниже рассматриваются различные алгоритмы умножения и соответствующие им схемы матричных умножителей. Каждый из алгоритмов имеет свои плюсы и минусы, важность которых для пользователя определяет выбор той или иной схемы.

Матричное умножение чисел без знака

Результат P перемножения двух n-разрядных двоичных целых чисел A и B без знака можно описать выражением

$$P = A \times B = \left(\sum_{i=0}^{n-1} a_i \times 2^i\right) \times \left(\sum_{j=0}^{n-1} b_j \times 2^j\right) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} a_i b_j \times 2^{i+j}.$$

Умножение сводится к параллельному формированию битов из n n-разрядных частичных произведений с последующим их суммированием с помощью матрицы сумматоров. Матричный умножитель для чисел без знака был предложен Эдвардом Брауном (Edward Louis Braun) в 1963 году [58].

Умножитель Брауна

В *умножителе Брауна* реализуется стандартная матрица умножения, приведенная на рис. 5.25. Схема умножителя Брауна для четырехразрядных двоичных чисел показана на рис. 5.26. В схеме каждому столбцу в матрице умножения соответствует диагональ умножителя. Биты частичных произведений (ЧП) вида $a_i b_j$ формируются с помощью элементов «И». Для суммирования ЧП применяются два вида одноразрядных сумматоров с сохранением переноса: полусумматоры (ПС)[1] и полные сумматоры (СМ)[2].

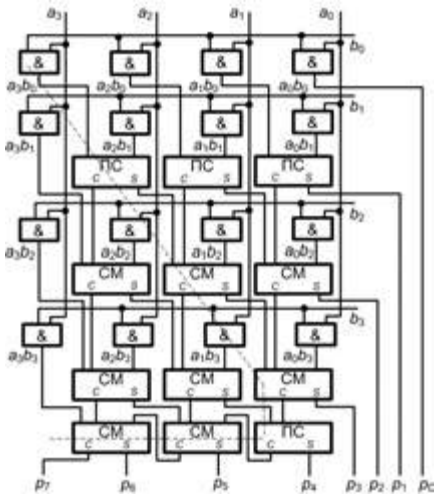


Рис. 5.26. Матричный умножитель Брауна для четырехразрядных чисел без знака

Матричный умножитель $n \times n$ содержит n^2 схем «И», n полусумматоров и $(n^2 - 2n)$ сумматоров. Если принять, что для реализации полусумматора требуются два логических элемента, а для полного сумматора — пять, то общее количество логических элементов в умножителе составляет $n^2 + 2n + 5(n^2 - 2n) = 6n^2 - 8n$.

Быстродействие умножителя определяется наиболее длинным маршрутом распространения сигнала (пунктирная линия на рис. 5.26), который в худшем случае включает в себя прохождение одной схемы «И», двух ПС и $(2n - 4)$ СМ. Полагая задержки в схеме «И» и полусумматоре равными Δ, а в полном сумматоре — 2Δ, общую задержку в умножителе можно оценить выражением $(4n - 5)\Delta$. Чтобы сократить ее длительность, n-разрядный сумматор с последовательным переносом в нижней строке умножителя можно заменить более быстрым вариантом сумматора. Последнее, однако, не всегда желательно, поскольку это увеличивает число используемых в умножителе логических элементов и ухудшает регулярность схемы.

---

[1] Полусумматором называется одноразрядное суммирующее устройство, имеющее два входа для слагаемых и два выхода — выход бита суммы и выход бита переноса.

[2] В отличие от полусумматора складывает три числа, то есть имеет три входа для слагаемых и два выхода — выход бита суммы и выход бита переноса.

В общем случае задержка в матричных умножителях пропорциональна их разрядности: $O(n)$.

Матричное умножение чисел в дополнительном коде

К сожалению, умножитель Брауна годится только для перемножения чисел без знака. В случае операндов со знаком отрицательные числа представляются дополнительным кодом, а матричные умножители строятся по схемам, отличным от схемы Брауна.

Сначала рассмотрим математические принципы, на которых основано умножение непосредственно в дополнительном коде. Число $A = (a_{n-1}a_{n-2} \ldots a_0)$, где a_{n-1} — знак числа, в дополнительном коде можно описать как

$$A = -a_{n-1} \times 2^{n-1} + \sum_{i=0}^{n-2} a_i \times 2^i .$$

Положительное $+A$ и отрицательное $-A$ значения числа в дополнительном коде соотносятся следующим образом [114]:

$$\begin{array}{cccc} +A & -a_{n-1} \times 2^{n-1} & +\sum_{i=0}^{n-2} a_i \times 2^i & +0 \\ \updownarrow & \updownarrow & \updownarrow & \updownarrow \\ -A & -(1-a_{n-1}) \times 2^{n-1} & +\sum_{i=0}^{n-2} (1-a_i) \times 2^i & +1 . \end{array}$$

Из приведенных выражений видно, что изменение знака числа может быть реализовано инвертированием битов, то есть заменой a_i на $(1-a_i)$ и заменой 0 на 1, а также обратной заменой. Определенные комбинации таких замен позволяют построить матричные умножители, обеспечивающие выполнение операции умножения чисел со знаком непосредственно в дополнительных кодах.

Умножитель Бо—Вули

Одна из возможных схем построения такого матричного умножителя была представлена в 1973 году Чарльзом Бо (Charles Baugh) и Брюсом Вули (Bruce Wooley) [53]. В алгоритме Бо—Вули произведение чисел в дополнительном коде представляется следующим соотношением:

$$P = A \times B = -2^{2n-1} + (\overline{a_{n-1}} + \overline{b_{n-1}} + a_{n-1}b_{n-1}) \times 2^{2n-2} + \sum_{i=0}^{n-2} \sum_{j=0}^{n-2} a_i b_j \times 2^{i+j} +$$

$$+ a_{n-1} \sum_{j=0}^{n-2} \overline{b_j} \times 2^{n+j-1} + b_{n-1} \sum_{i=0}^{n-2} \overline{a_i} \times 2^{n+i-1} + (a_{n-1} + b_{n-1}) \times 2^{n-1}.$$

Матрица умножения, реализующая алгоритм, приведена на рис. 5.27, а соответствующая ей схема умножителя — на рис. 5.28.

В методе Бо—Вули частичные произведения приводятся к такому виду, который обеспечивает максимальную регулярность массива, что очень удобно при

микросхемной реализации. По ходу умножения частичные произведения, имеющие знак «минус», смещаются к последней ступени суммирования. Операция вычитания ЧП заменяется прибавлением их инвертированных значений. Недостатком схемы можно считать то, что в последней строке матрицы требуется дополнительный сумматор, из-за чего регулярность схемы нарушается. В целом, в схеме используются $n(n-2)+4$ полных сумматора и $(n-1)$ полусумматор.

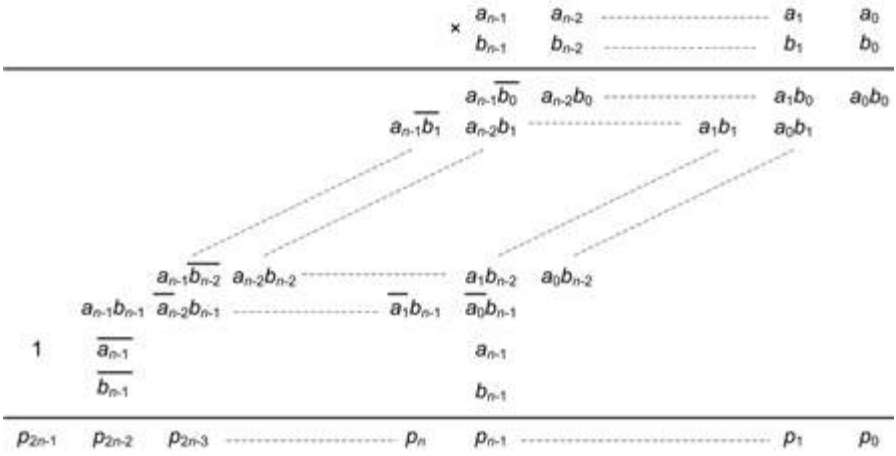


Рис. 5.27. Матрица перемножения n-разрядных чисел согласно алгоритму Бо—Вули

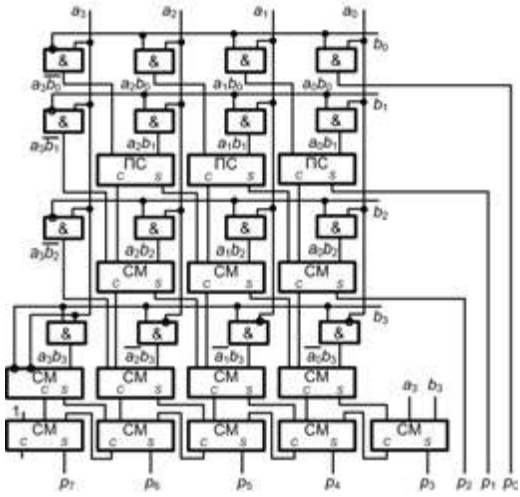


Рис. 5.28. Матричный умножитель для четырехразрядных чисел в дополнительном коде по схеме Бо—Вули

Если в выражении, описывающем алгоритм Бо—Вули, изменить форму представления разрядов множителя: $b_i = (1 - |b_i|) + 1 - 2 = \overline{b_i} + 1 - 2$, то получим модифицированный вариант алгоритма:

$$P = A \times B = a_{n-1}b_{n-1} + \sum_{i=0}^{n-2}\sum_{j=0}^{n-2} a_i b_j \times 2^{i+j} + \sum_{i=0}^{n-2} \overline{a_i b_{n-1}} \times 2^{n+i-1} +$$

$$+ \sum_{j=0}^{n-2} \overline{a_{n-1} b_j} \times 2^{n+j-1} + 2^n + 2^{2n-1}.$$

Матрица умножения чисел со знаком (рис. 5.29), представленных в дополнительном коде, похожа на матрицу перемножения чисел без знаков. Отличие состоит в том, что $(2n - 2)$ частичных произведений инвертированы, а в столбцы n и $(2n - 1)$ добавлены единицы.

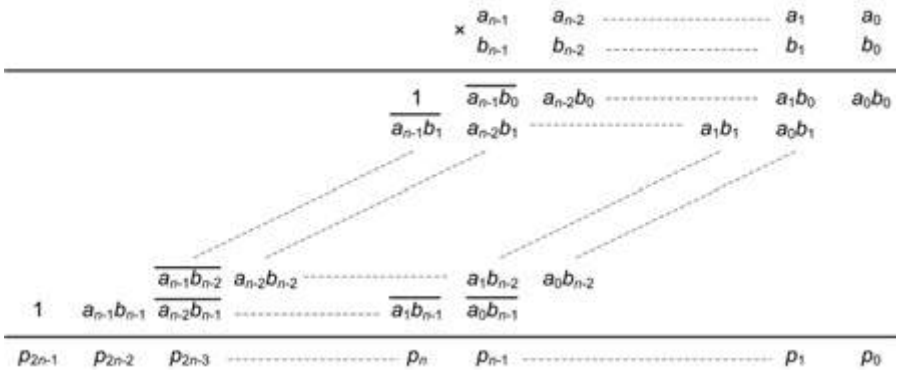


Рис. 5.29. Матрица перемножения n-разрядных чисел в дополнительном коде согласно модифицированному алгоритму Бо—Вули

Соответствующая схема матричного умножителя для четырехразрядных чисел показана на рис. 5.30.

Здесь $(2n - 2)$ частичных произведений инвертированы за счет замены элементов «И» на элементы «И-НЕ». Сумматор в младшем разряде нижнего ряда складывает 1 в столбце n с вектором сумм и переносов из предшествующей строки, реализуя при этом следующие выражения:

$$s_i = \overline{a_i \oplus b_i}; \quad c_{i+1} = a_i \times b_i.$$

Инвертор в нижней строке слева обеспечивает добавление единицы в столбец $(2n - 1)$.

Модификация алгоритма привела к сокращению количества сумматоров в последних строках матричного массива.

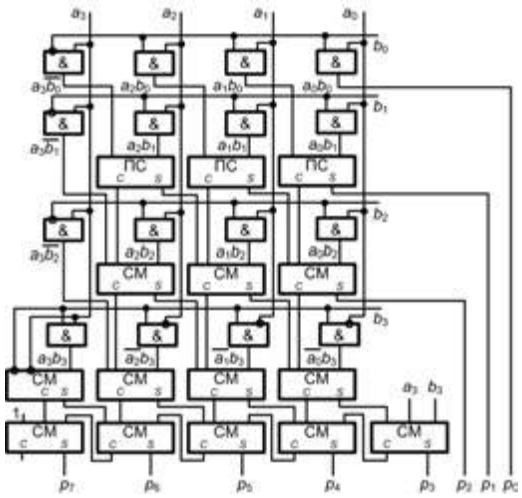


Рис. 5.30. Матричный умножитель для четырехразрядных чисел в дополнительном коде по модифицированной схеме Бо—Вули

Умножитель Пезариса

Еще один алгоритм для вычисления произведения чисел в дополнительном коде был предложен Стилианосом Пезарисом (Stylianos Pezaris) в 1971 году [130]. При представлении числа в дополнительном коде старший разряд числа имеет отрицательный вес. Это учитывается в матрице умножения (рис. 5.31). Перед логическими произведениями, имеющими отрицательный вес, стоит знак «–», трактуемый следующим образом: $-1 = -2 \times 1 + 1$; $-0 = -2 \times 0 + 0$.

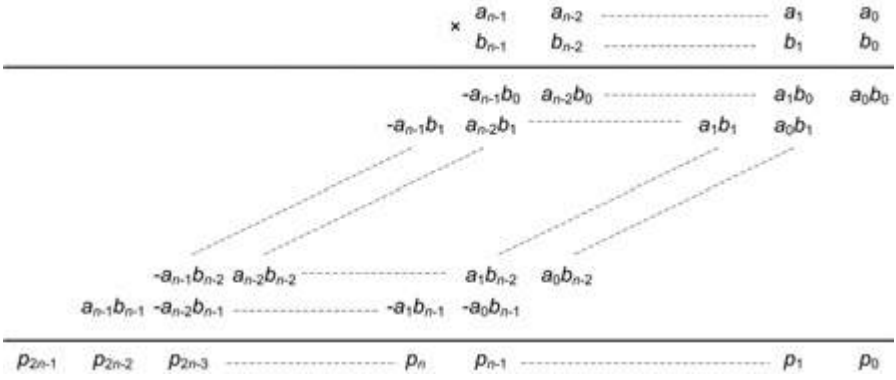


Рис. 5.31. Матрица перемножения n-разрядных чисел согласно алгоритму Пезариса

Для реализации приведенной схемы матричного умножения Пезарис предлагает использовать в умножителе четыре вида полных сумматоров (рис. 5.32).

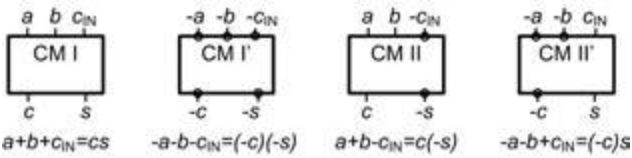


Рис. 5.32. Виды сумматоров, применяемых в матричном умножителе Пезариса

В сумматоре типа SM I, который фактически является обычным полным сумматором, все входные данные (a, b, c_{IN}) имеют положительный вес, а результат лежит в диапазоне 0–3. Этот результат представлен двухразрядным двоичным числом cs, где c и s также присвоены положительные веса. В остальных трех типах сумматоров некоторые из сигналов имеют отрицательный вес.

Схема умножителя, реализующего алгоритм Пезариса, приведена на рис. 5.33. Для построения умножителя Пезариса $n \times n$ требуется $(n-2)^2$ сумматоров типа CM I, $(n-2)$ — типа CM II, $(2n-3)$ — типа CM II' и один сумматор типа CM I'. Всего — $n(n-1)$ сумматор.

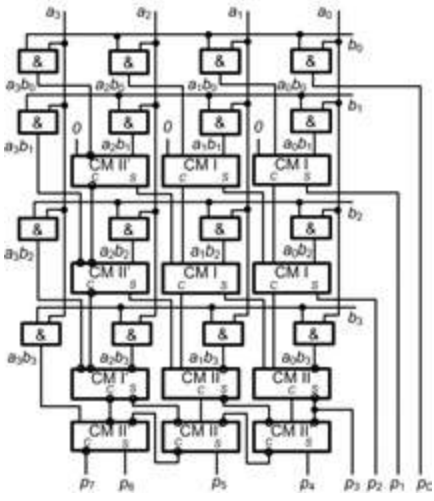


Рис. 5.33. Матричный умножитель для четырехразрядных чисел в дополнительном коде по схеме Пезариса

По сравнению с умножителем Бо—Вули, схема Пезариса имеет более регулярный вид, но, с другой стороны, она предполагает присутствие нескольких типов сумматоров.

Древовидные схемы умножения

Сократить задержку, свойственную матричным умножителям, удается в схемах с древовидной структурой. В матричных умножителях каждое очередное частичное произведение прибавляется к СЧП с помощью отдельной строки сумматоров, и для перемножения n-разрядных чисел требуется n таких строк (рис. 5.34, *а*).

В древовидных умножителях процесс получения СЧП также реализуется за счет строк сумматоров, но организованных по схеме дерева, благодаря чему количество строк сумматоров сокращается до $\log_2 n$ (рис. 5.34, *б*). Так как в умножителях обоих типов каждая строка вносит задержку, свойственную одному полному сумматору, количество строк во многом определяет общее быстродействие умножителя.

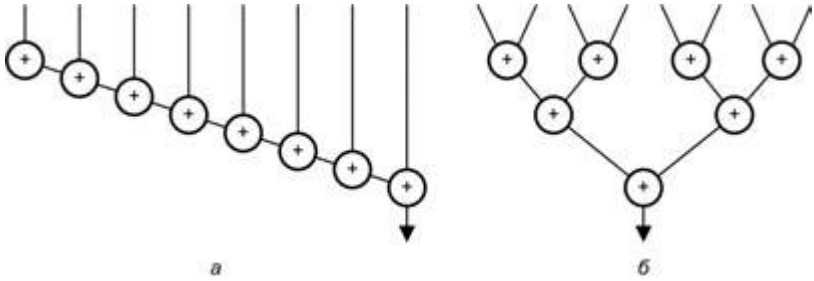


Рис. 5.34. Суммирование частичных произведений в умножителях: *а* — с матричной структурой; *б* — со структурой двоичного дерева

Древовидные умножители включают в себя три ступени (рис. 5.35):

- ступень формирования всех возможных комбинаций логических произведений вида $a_i b_j$;
- ступень сжатия частичных произведений;
- ступень заключительного суммирования.

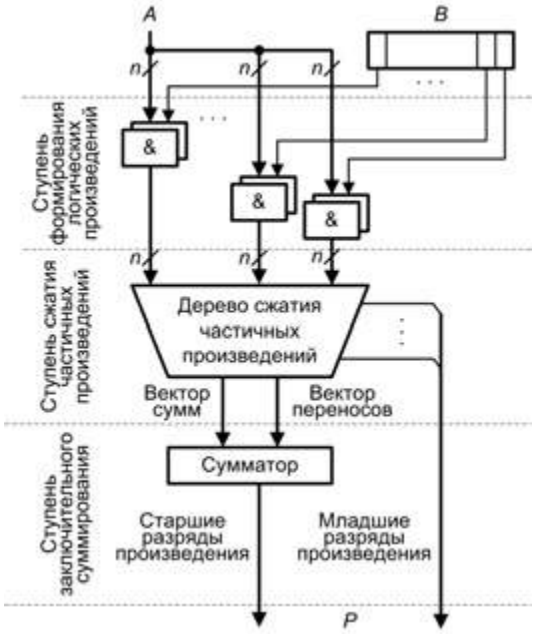


Рис. 5.35. Структура древовидного умножителя

Назначение ступени формирования логических произведений очевидно из общей схемы перемножения двоичных чисел (см. рис. 5.25). Из нее явствует, что элементами частичных произведений служат логические произведения определенного разряда множимого на определенный разряд множителя. Для представления всех получающихся при умножении частичных произведений нужно вычислить все возможные логические произведения вида $a_i b_j$, ($i = 0 \ldots n-1$; $j = 0 \ldots n-1$). В рассматриваемой ступени умножителя это обеспечивается n^2 элементами «И».

Ступень сжатия частичных произведений — это ядро древовидных умножителей. Процесс суммирования ЧП реализуется строками сумматоров, но, в отличие от матричной схемы, связанных между собой по схеме дерева. Как известно, идея древовидного суммирования заключается в поэтапном уменьшении («сжатии») числа слагаемых. Известные древовидные умножители различаются по способу и порядку суммирования ЧП. Роль устройств сжатия в большинстве умножителей играют полные сумматоры (СМ) и полусумматоры (ПС), которые обычно называют счетчиками (3, 2) и (2, 2) соответственно. Связано это с тем, что код на выходах СМ и ПС *cs*[1], как и в двоичном счетчике, равен количеству единиц, поданных на входы. На выходе дерева формируются два вектора: вектор сумм и вектор переносов.

На ступени заключительного суммирования вектор сумм и вектор переносов, сформированные в процессе сжатия ЧП, обрабатываются многоразрядным сумматором, в результате чего образуется окончательный результат. Обычно здесь применяется быстрый сумматор с временем задержки, пропорциональным $O(\log_2(n))$.

Различие между схемами сжатия касается, главным образом, способа формирования вектора суммы и вектора переносов. В известных на сегодня умножителях наибольшее распространение получили три древовидных схемы суммирования ЧП: дерево Уоллеса, дерево Дадда и дерево со структурой «перевернутой лестницы».

Умножитель Уоллеса

В наиболее общей формулировке дерево Уоллеса — это оператор с n входами и $\log_2 n$ выходами, в котором код на выходе равен числу единиц во входном коде. Вес битов на входе совпадает с весом младшего разряда выходного кода. Простейшим деревом Уоллеса является одноразрядный сумматор. Используя такие сумматоры, а также полусумматоры, можно построить дерево Уоллеса для перемножения чисел любой разрядности, при этом количество сумматоров возрастает пропорционально величине $\log_2 n$. В такой же пропорции растет время выполнения операции умножения.

Согласно алгоритму Уоллеса, строки матрицы частичных произведений группируются по три. Полные сумматоры используются для сжатия столбцов с тремя битами, а полусумматоры — столбцов с двумя битами. Строки, не попавшие в набор из трех строк, учитываются в следующем каскаде сжатия. Алгоритм Уоллеса ориентирован на сжатие кодов как можно раньше, на самых ранних этапах.

Логика построения дерева Уоллеса для суммирования частичных произведений в умножителе 4×4 показана на рис. 5.36, *а*. Для пояснения структуры дерева сумматоров часто применяют так называемую точечную диаграмму (рис. 5.36, *б*). В ней

[1] *c* — выход переноса, *s* — выход суммы.

точки обозначают биты частичных произведений, прямые диагональные линии представляют выходы полных сумматоров, а перечеркнутые диагонали — выходы полусумматоров. Хотя на рис. 5.36, *а* в третьем каскаде показаны три строки, фактически после сжатия остаются две первых, а третья лишь отражает переносы, которые учитываются при окончательном суммировании. Этим объясняется кажущееся отличие от точечной диаграммы.

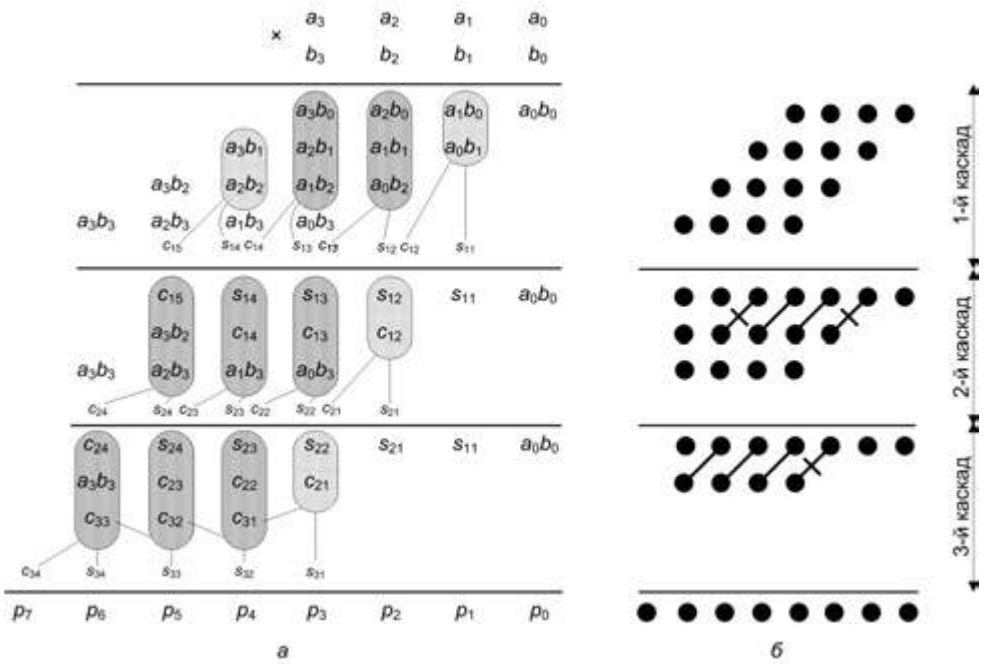


Рис. 5.36. Суммирование ЧП с помощью дерева Уоллеса (вариант 1): *а* — логика суммирования; *б* — точечная диаграмма

Умножитель (рис. 5.37) состоит из трех ступеней и содержит 16 схем «И», 3 полусумматора, 5 полных сумматоров. Сложение векторов сумм и переносов в последнем каскаде реализуется четырехразрядным сумматором с последовательным распространением переноса, однако чаще для ускорения привлекаются более эффективные схемы распространения переноса, например параллельная.

Отметим, что избыточность кодирования, заложенная в алгоритм Уоллеса, приводит к тому, что возможно построение различных вариантов схемы дерева. На рис. 5.38 показана иная реализация дерева Уоллеса для четырехразрядных операндов.

Как видно, новый вариант схемы никакого выигрыша в аппаратурном плане не дает.

Схема Уоллеса считается наиболее быстрой, но в то же время ее структура наименее регулярна, из-за чего предпочтение отдается иным древовидным структурам. Основная сфера использования умножителей со схемой Уоллеса — перемножение чисел большой разрядности. В этом случае быстродействие имеет превалирующее значение.

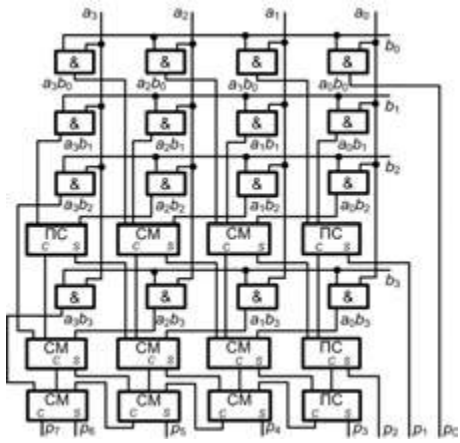


Рис. 5.37. Умножитель 4 × 4 со структурой дерева Уоллеса

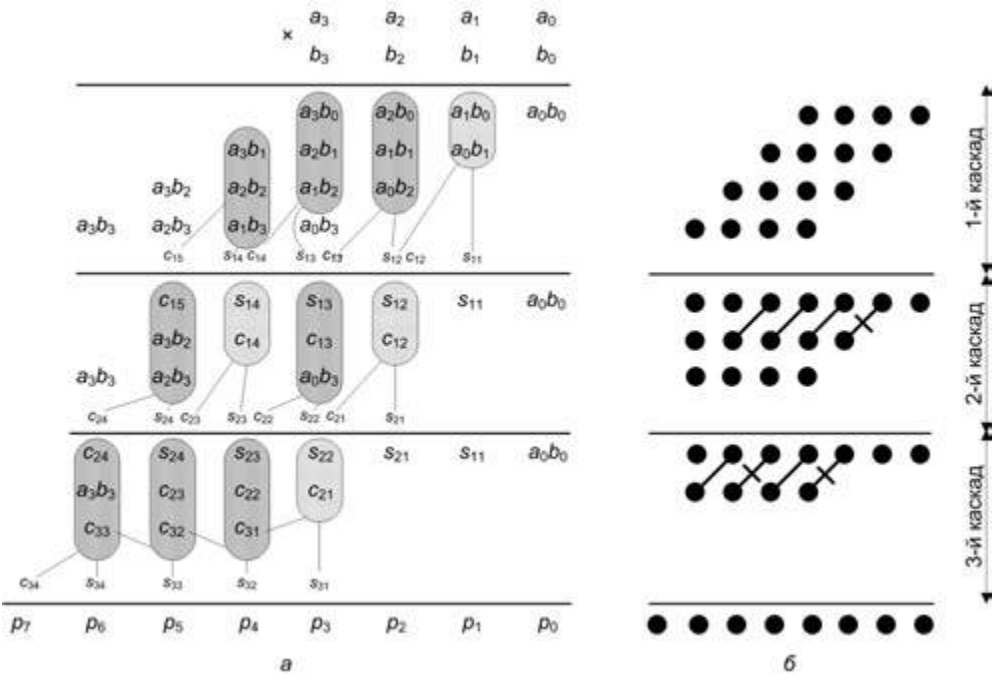


Рис. 5.38. Суммирование ЧП с помощью дерева Уоллеса (вариант 2): *а* — логика суммирования; *б* — точечная диаграмма

Умножитель Дадда

При умножении чисел небольшой разрядности более распространена другая схема сжатия ЧП — схема дерева Л. Дадда. В ее основе также лежит дерево Уоллеса, но

реализуемое минимальным числом сумматоров. Различия методов Уоллеса и Дадда являются следствием разных подходов к решению задачи сжатия. Алгоритм Уоллеса ориентирован на сжатие кодов как можно раньше, на самых ранних этапах, а алгоритм Дадда стремится это сделать по возможности позже, то есть наибольший уровень сжатия относит к завершающим стадиям.

На рис. 5.39 описан умножитель 4 × 4, реализующий алгоритм дерева Дадда. Для его реализации требуется 16 схем «И», два полусумматора, четыре полных сумматора и шестиразрядный сумматор. Схема содержит три каскада. На этапе суммирования вектора сумм и вектора переносов необходим $(2n - 2)$-разрядный сумматор (в нашем примере — 6-разрядный).

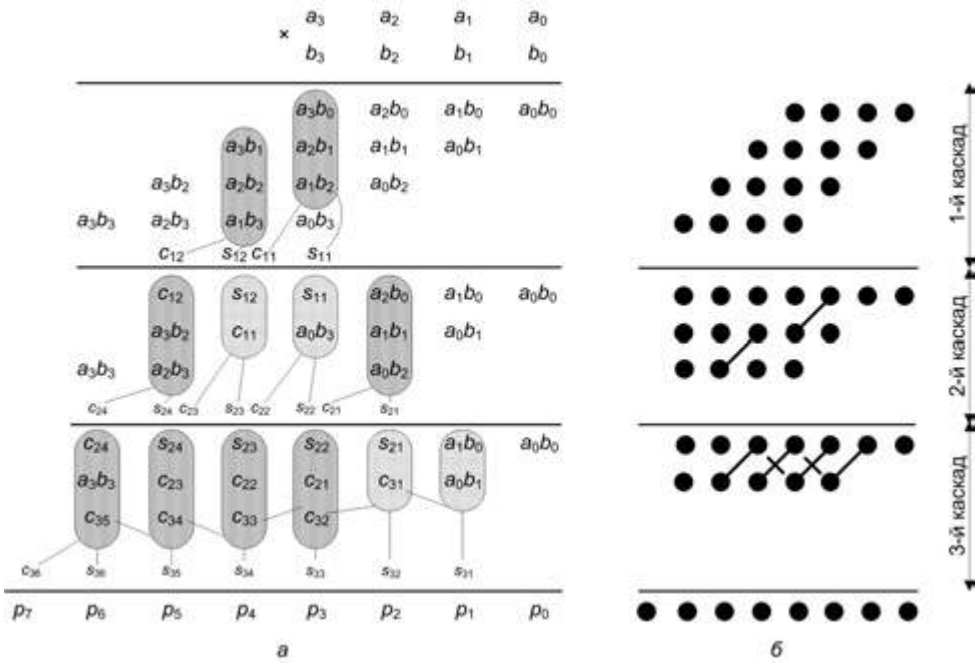


Рис. 5.39. Суммирование ЧП с помощью дерева Дадда для случая чисел без знака: *а* — логика суммирования; *б* — точечная диаграмма

Схема умножения чисел в дополнительном коде, рассмотренная применительно к матричному умножителю, может быть адаптирована и для умножителя со схемой Дадда. В таком случае схема сжатия приобретает вид, показанный на рис. 5.40.

Сравнивая схемы Уоллеса и Дадда, можно отметить, что число каскадов сжатия в них одинаково, однако количество используемых полусумматоров и полных сумматоров в схеме Дадда меньше (при подсчете числа элементов обычно не учитывают многоразрядные сумматоры, предназначенные для окончательного сложения векторов сумм и переносов). С другой стороны, на этапе сложения векторов сумм и переносов в варианте Уоллеса требуется сумматор с меньшим числом разрядов (в нашем примере — 4 против 6).

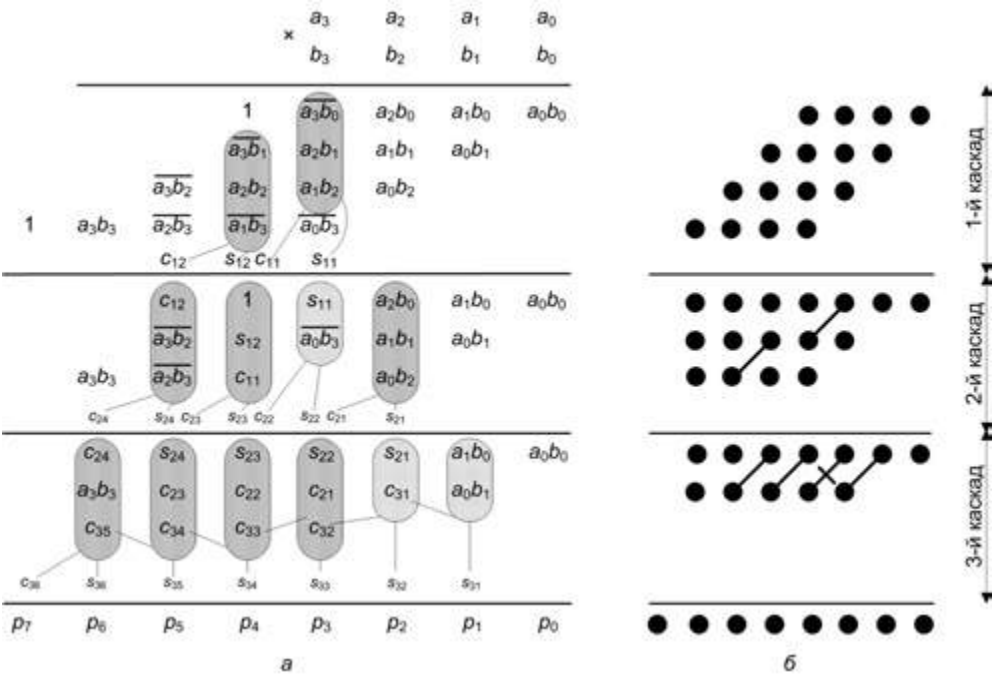


Рис. 5.40. Суммирование ЧП в дополнительном коде с помощью дерева Дадда:
а — логика суммирования; *б* — точечная диаграмма

У обеих схем имеется общий недостаток — нерегулярность структуры, особенно у дерева Уоллеса.

Умножитель со схемой перевернутой лестницы

Схема перевернутой лестницы (overturned stairs), предложенная в [121], являет собой одну из попыток сделать древообразную структуру более регулярной. Как и в предыдущих древовидных умножителях, речь идет о схеме объединения сумматоров для получения суммы частичных произведений. Отличительными особенностями рассматриваемой схемы является ее рекурсивная структура (схема на n входов строится на основе схемы на $n - 1$ вход) и малая высота дерева (под высотой дерева здесь будет пониматься количество ступеней сумматоров от корня до самой удаленной от него точки, включая и сам корень).

«Лестница» строится из базовых блоков четырех видов, которые авторы назвали *Телом* (body), *Корнем* (root), *Соединителем* (connector) и *Ветвлением* (branch). На рис. 5.41 показаны структуры деревьев типа перевернутой лестницы на разное число входов с указанием использованных в них базовых блоков. Общую процедуру построения «лестницы» иллюстрирует рис. 5.41, *д*. Предположим, что требуется создать дерево высотой $j + 1$. Как видно из рисунка, *Тело* с высотой j может быть построено из *Тела* высотой $j - 1$, *Соединителя* и *Ветвления*. Соединитель состоит из двух сумматоров (рис. 5.41, *в–д*). Три выхода *Соединителя* служат входами *Корня*,

представленного сумматором. Ветвление высотой $j-2$ реализуется цепочкой из определенным образом соединенных $j-2$ сумматоров. Следуя этой логике, можно итеративно создавать деревья типа «перевернутая лестница» на произвольное число входов.

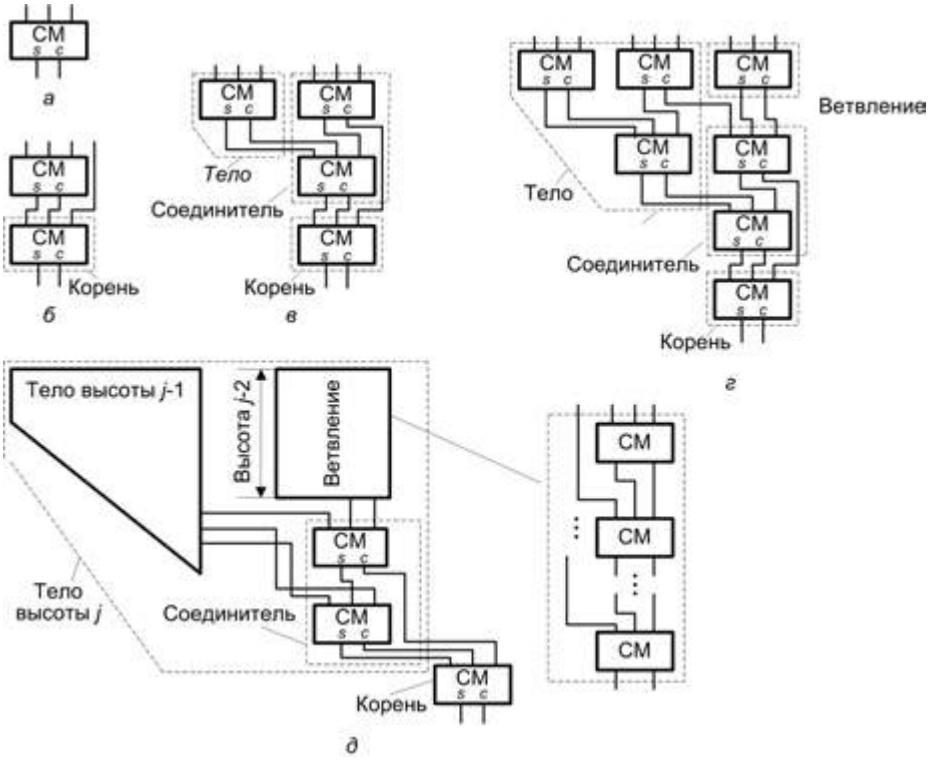


Рис. 5.41. Перевернутое дерево: *а, б, в* — базовые блоки; *г, д* — структуры деревьев

Базовые элементы объединяются, образуя дерево, имеющее n входов. Как видно, дерево имеет достаточно регулярную структуру, однако нужно учитывать, что веса отдельных входов различаются. Кроме того, конструкция умножителя получается сложной.

Сравнительная оценка схем умножения с матричной и древообразной структурой

Анализ схем умножителей в совокупности с данными, приводимыми в литературе, позволяет сделать следующие выводы. Наиболее быстро работают матричные умножители, построенные по схеме Бута (этот вид умножителей реализует алгоритм Бута, но в матричном варианте, в учебнике не рассматривался), а также умножители с древовидной структурой, в частности умножитель Дадда. Для операндов длиной в 16 разрядов и более привлекательной представляется модифицированная схема Бута, как по скорости, так и по затратам оборудования. Максимально

быстрое выполнение операции умножения обеспечивает сочетание алгоритма Бута и дерева Уоллеса. С другой стороны, достаточно хорошие показатели скорости при умножении чисел небольшой разрядности демонстрирует схема Бо—Вули. В плане потребляемой мощности наиболее экономичными являются умножители, построенные по схемам Брауна и Пезариса. Несмотря на сравнительно небольшое число используемых транзисторов, схемы на базе алгоритма Бута, а также древовидные реализации более энергоемки из-за избыточных внутренних связей, связанных с нерегулярной структурой этих схем.

Конвейеризация параллельных умножителей

В матричной и древовидной структурах параллельных умножителей заложен еще один потенциал повышения производительности — возможность конвейеризации. При конвейеризации весь процесс вычислений разбивается на последовательность законченных шагов. Каждый из этапов процедуры умножения выполняется на своей ступени конвейера, причем все ступени работают параллельно. Результаты, полученные на i-й ступени, передаются на дальнейшую обработку в $(i + 1)$-ю ступень конвейера. Перенос информации со ступени на ступень происходит через буферную память, размещаемую между ними (рис. 5.42).

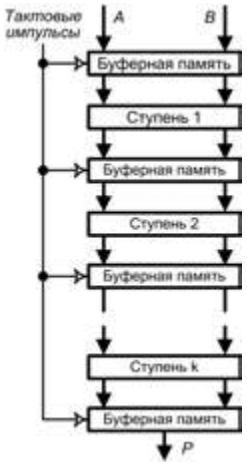


Рис. 5.42. Структура конвейерного умножителя

Выполнившая свою операцию ступень помещает результат в буферную память и может приступать к обработке следующей порции данных операций, в то время как очередная ступень конвейера в качестве исходных использует данные, хранящиеся в буферной памяти, расположенной на ее входе. Синхронность работы конвейера обеспечивается тактовыми импульсами, период которых τ определяется самой медленной ступенью конвейера τ_i и задержкой в элементе буферной памяти τ_1: $\tau = \max(\tau_1, \tau_2, ..., \tau_k) + \tau_i$. Несмотря на то что время выполнения операции умножения для каждой конкретной пары сомножителей в конвейерном умножителе не только не уменьшается, но даже несколько увеличивается за счет задержек в буферной памяти, при перемножении последовательностей пар сомножителей,

достигаемый выигрыш весьма ощутим. Действительно, в конвейерном умножителе из k ступеней перемножаемые данные могут подаваться на вход с интервалом в k раз меньшим, чем в случае обычного умножителя. В том же темпе появляются и результаты на выходе.

Схема конвейера легко может быть применена к матричным и древовидным умножителям. В матричных умножителях в качестве ступени конвейера выступает каждая строка матрицы сумматоров. В качестве примера конвейеризации матричных умножителей на рис. 5.43 приведена схема четырехразрядного умножителя Брауна. Черными прямоугольниками обозначены триггеры, образующие буферную память.

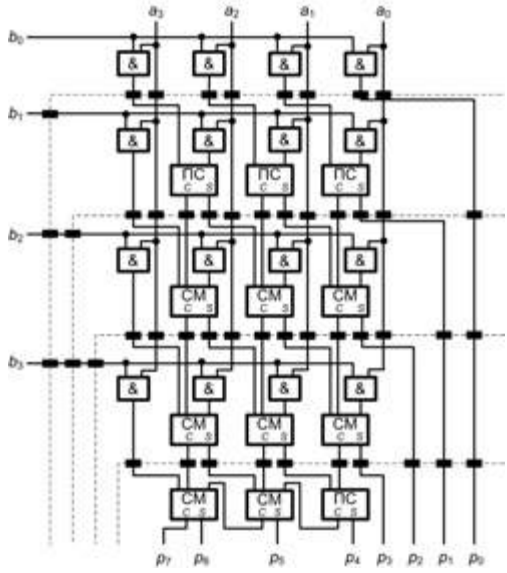


Рис. 5.43. Конвейеризированный матричный умножитель Брауна

Конвейеризация матричных умножителей на уровне строк сумматоров может быть затруднительной из-за большого числа ступеней и необходимости введения в состав умножителя значительного количества триггеров. Сократить их число можно за счет следующих приемов:

- отказа от использования идеи конвейеризации между входными схемами «И» и первой строкой полных сумматоров;
- увеличением времени обработки на каждой ступени, например, можно принять его равным удвоенному времени срабатывания полного сумматора;
- отказом от формирования всех n^2 битов частичных произведений в самом начале, перед первой ступенью конвейера, и вычислением их по мере необходимости на разных ступенях конвейера.

В древовидных умножителях в качестве ступеней конвейера выступают каскады сжатия, то есть более крупные образования, чем в матричных умножителях.

Поскольку количество каскадов значительно меньше числа строк в матричных умножителях, конвейеризация древовидных умножителей более привлекательна. Конвейеризированный вариант древовидных умножителей получается из обычного путем размещения буферной памяти между соседними каскадами схемы.

Рекурсивная декомпозиция операции умножения

Как правило, аппаратные умножители, построенные на рассмотренных принципах, имеют ограничение на число разрядов вводимых чисел. Умножитель повышенной разрядности можно получить из модулей меньшей разрядности, выстраивая так называемую *рекурсивную декомпозицию операции умножения*. Так, для построения умножителя 8×8 можно использовать четыре модуля типа 4×4. Множимое A разбивается на четыре старших (A_h) и четыре младших (A_l) разряда. Множитель B таким же образом разбивается на части B_h и B_l. Четыре модуля типа 4×4 вычисляют соответственно произведения $A_h \times B_h$, $A_h \times B_l$, $A_l \times B_h$, $A_l \times B_l$. На выходах модулей получаются восьмиразрядные результаты, которые соответствуют частичным произведениям в разрядах: 15–8, 11–4, снова 11–4 и 7–0. Окончательный результат формируется путем суммирования этих четырех частичных произведений с учетом их положения в разрядной сетке (рис. 5.44).

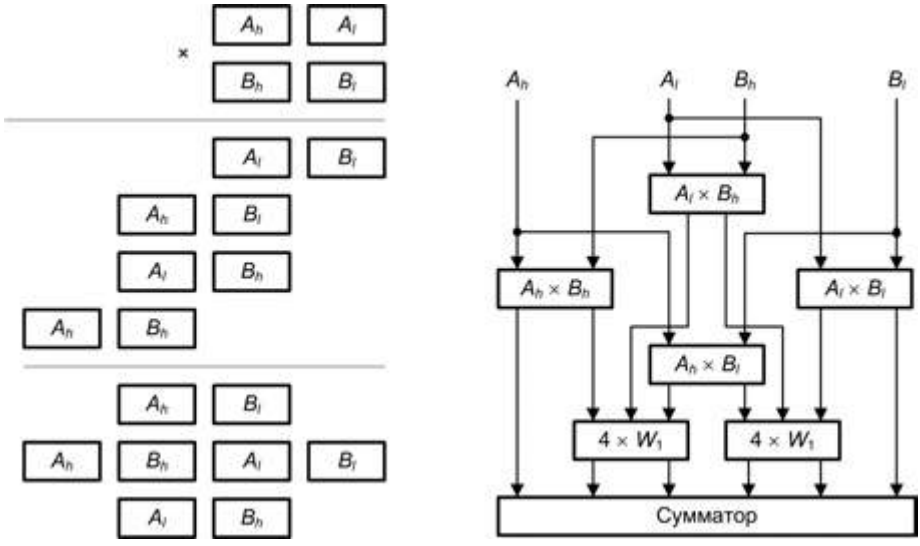


Рис. 5.44. Декомпозиция операции умножения

Деление

Деление несколько более сложная операция, чем умножение, но базируется на тех же принципах. Если умножение выполняется путем многократных сдвигов и сложений, то деление, будучи операцией обратной умножению,— путем многократных сдвигов и вычитаний. Основу составляет общепринятый способ деления с помощью операций вычитания (сложения) и сдвига (рис. 5.45).

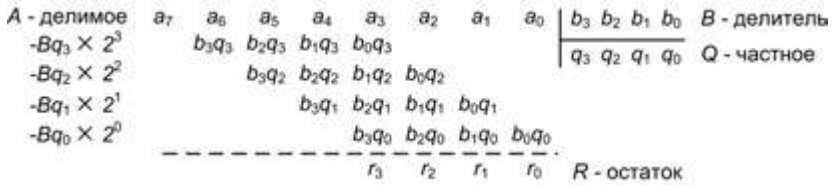


Рис. 5.45. Общая схема операции деления

Задача деления числа A на число B сводится к вычислению частного Q и остатка R:

$$\frac{A}{B} = Q + \frac{R}{B} \qquad (A = QB + R; \quad R < B).$$

Делимое A ($a_{2n-1}a_{2n-2} \ldots a_1a_0$) обычно представляется двойным словом ($2n$ разрядов), делитель B ($b_{n-1}b_{n-2} \ldots b_1b_0$), частное Q ($q_{n-1}q_{n-2} \ldots q_1q_0$) и остаток R ($r_{n-1}r_{n-2} \ldots r_1r_0$) имеют разрядность n. При делении дробных чисел делимое может иметь формат одинарного слова (n разрядов). В остальном процедуры деления целых чисел и дробных чисел практически идентичны, поэтому иллюстрация алгоритмов деления приводится на примере целых чисел.

Частное от деления $2n$-разрядного числа на n-разрядное может содержать более чем n разрядов. В этом случае возникает переполнение, поэтому перед выполнением деления необходима проверка. Переполнения не будет, если число, содержащееся в старших n разрядах делимого, меньше делителя (для целых чисел) или делимое меньше делителя (для дробных чисел). Помимо этого требования, перед началом операции необходимо исключить возможность ситуации деления на 0.

Реализовать деление можно двумя основными способами:

- с неподвижным делимым и сдвигаемым вправо делителем;
- с неподвижным делителем и сдвигаемым влево делимым.

Недостатком первого способа является потребность иметь в устройстве деления сумматор и регистр двойной длины. Второй способ позволяет обойтись узлами одинарной длины: неподвижный делитель D хранится в регистре одинарной длины, а делимое A, сдвигаемое относительно D, находится в двух таких же регистрах. Образующиеся цифры частного Q заносятся в освобождающиеся при сдвиге A разряды одного из регистров делимого.

Ниже на примере чисел без знака рассматриваются два основных алгоритма целочисленного деления.

Деление с восстановлением остатка

Наиболее очевидный алгоритм носит название алгоритма *деления с неподвижным делителем и восстановлением остатка*. Он приводится в силу того, что очень похож на общепринятый способ деления столбиком. Данный алгоритм может быть описан следующим образом.

1. Исходное значение частичного остатка (ЧО) полагается равным старшим разрядам делимого.

2. Из ЧО вычитается делитель и анализируется знак остатка.
3. Если остаток положительный, то деление невозможно, формируется признак переполнения и процесс завершается, в противном случае ЧО восстанавливается путем прибавления делителя и деление продолжается.
4. Частичный остаток сдвигается на один разряд влево, а в освобождающийся при сдвиге младший разряд ЧО заносится очередная цифра делимого.
5. Из сдвинутого ЧО вычитается делитель и анализируется знак результата вычитания.
6. Очередная цифра модуля частного равна единице, когда результат вычитания положителен, и нулю, если отрицателен. В последнем случае ЧО восстанавливается до того значения, которое было до вычитания.
7. Пункты 4–6 последовательно выполняются для получения всех цифр модуля частного.

На рис. 5.46 показан процесс деления с восстановлением остатка, здесь число 41 делится на 7.

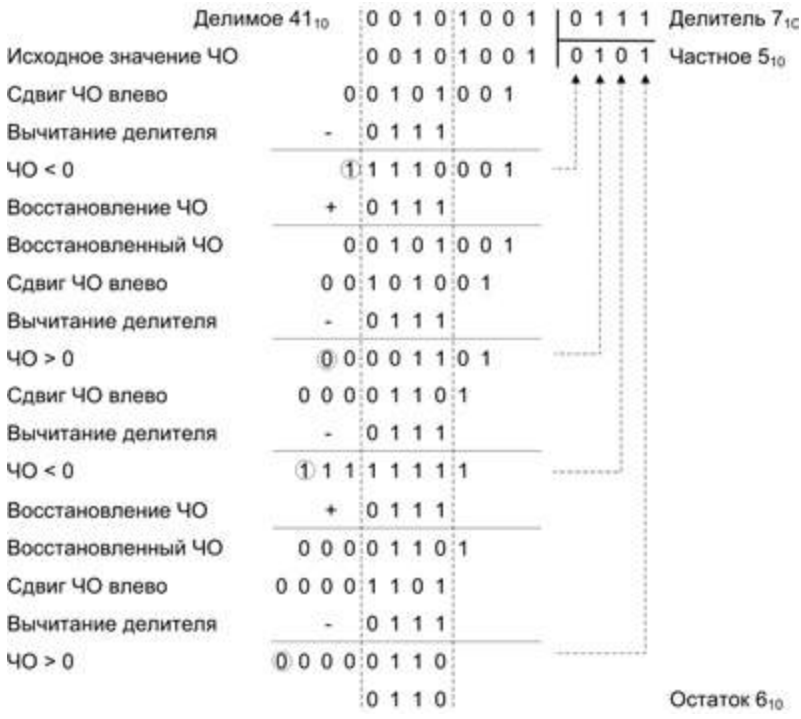


Рис. 5.46. Пример деления с восстановлением остатка

Деление без восстановления остатка

Недостаток описанного алгоритма — необходимость выполнения на отдельных шагах дополнительных операций сложения для восстановления частичного остатка.

Это увеличивает время выполнения деления, которое, к тому же, может меняться в зависимости от конкретного сочетания кодов операндов. В силу указанных причин реальные делители строятся на основе алгоритма *деления с неподвижным делителем без восстановления остатка*. В нем пункты 1–4 и 7 полностью совпадают с соответствующими пунктами предыдущего алгоритма деления, а пункты 5 и 6 имеют следующую формулировку:

1. Из сдвинутого ЧО вычитается делитель, если остаток положителен, и к сдвинутому частичному остатку прибавляется делитель, если остаток отрицательный.
2. Очередная цифра модуля частного равна единице, если результат операции (сложения или вычитания) положителен, и нулю, если он отрицателен.

Процесс деления без восстановления остатка для ранее рассмотренного примера демонстрируется на рис. 5.47.

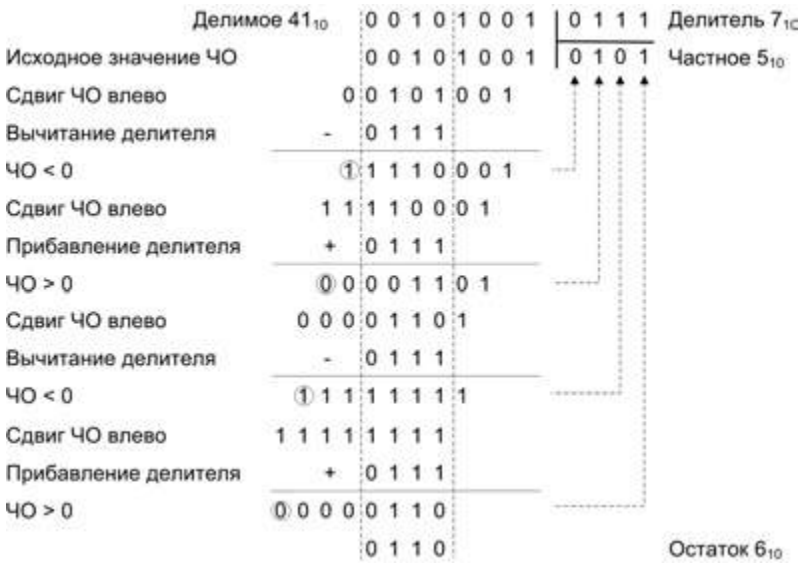


Рис. 5.47. Пример деления без восстановления остатка

Деление чисел со знаком

Как и в случае умножения, деление чисел со знаком может быть выполнено путем перехода к абсолютным значениям делимого и делителя, с последующим присвоением частному знака «плюс» при совпадающих знаках делимого и делителя либо «минус» — в противном случае.

Деление чисел, представленных в дополнительном коде, можно осуществлять не переходя к модулям. Рассмотрим необходимые для этого изменения в алгоритме без восстановления остатка. Так как делимое и делитель не обязательно имеют одинаковые знаки, то действия с частичным остатком (прибавление или вычитание B) зависят от знаков ЧО и делителя. Если эти знаки совпадают, то в очередной итерации деления производится вычитание делителя, а очередной цифрой частного

будет 1. При разных знаках ЧО и делителя последний прибавляется к ЧО, и очередная цифра частного — 0.

По завершении деления может понадобиться коррекция результата, заключающаяся в увеличении частного на единицу. Такая коррекция производится в следующих случаях:

- $A > 0$ и $B < 0$;
- $A < 0$, $B > 0$ и $R \neq 0$;
- $A < 0$, $B < 0$ и $R = 0$.

При стандартном определении операции деления частное Q и остаток R отвечают отношению $A = Q \times B + R$, где остаток от деления $0 \le |R| < |B|$. При таком определении возможны два вида остатка. Например, результат деления −42 на −5 может быть представлен как $-42 = 9 \times (-5) + 3$ или $-42 = 8 \times (-5) + (-2)$, то есть остаток может быть 3 или −2. Для определенности в ВМ принято, что остаток всегда приводится к положительному числу, то есть если по завершении деления он отрицателен, к нему следует прибавить модуль делителя.

Устройство деления

Рассмотренный алгоритм деления без восстановления остатка может быть реализован с помощью устройства, схема которого приведена на рис. 5.48.

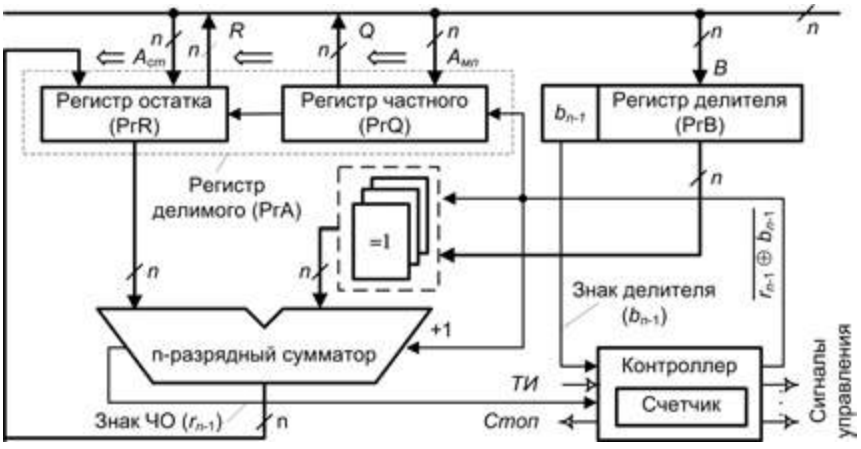


Рис. 5.48. Схема деления по алгоритму без восстановления остатка

Процедура начинается с занесения делимого A в $2n$-разрядный регистр делимого (PrA), роль которого исполняют n-разрядный регистр остатка (PrR) и n-разрядный регистр частного (PrQ). В PrR заносятся старшие разряды делимого ($A_{ст}$), а в PrQ — младшие ($A_{мл}$). Делитель B записывается в n-разрядный регистр делителя (PrB). В счетчик циклов, служащий для подсчета количества полученных цифр частного, помещается исходное значение, равное числу итераций в процедуре деления.

На каждом шаге содержимое PrR и PrQ совместно сдвигается на один разряд влево. В зависимости от сочетания знаков частичного остатка (r_{n-1}) и делителя (b_{n-1})

определяется значение очередной цифры частного и требуемое действие: вычитание или прибавление делителя. Знаки анализируются контроллером в соответствии с выражением $\overline{r_{n-1} \oplus b_{n-1}}$. Операция вычитания заменяется прибавлением делителя, взятого с обратным знаком и представленного в дополнительном коде. Изменение знака и преобразование в дополнительный код реализуется путем передачи делителя на вход сумматора инверсным кодом (инвертирование обеспечивают схемы исключающего ИЛИ на одном из входов сумматора) с последующим добавлением единицы к младшему разряду сумматора. Очередная цифра частного записывается в освободившийся при сдвиге младший разряд PrQ. Содержимое счетчика циклов уменьшается на единицу.

Процедура повторяется до исчерпания всех цифр делимого, о чем свидетельствует нулевое содержимое счетчика циклов. По окончании операции деления частное располагается в регистре частного, а в регистре остатка будет находиться остаток от деления.

На заключительном этапе, если это необходимо, производится корректировка полученного результата, как это предусматривает алгоритм деления чисел со знаком (цепи коррекции результата на схеме не показаны).

Ускорение целочисленного деления

Следует отметить, что операция деления предоставляет не слишком много путей для своей оптимизации по времени. Тем не менее определенные возможности для убыстрения деления существуют. В ряде случаев эффект достигается применением алгоритмов, сокращающих число итераций в процедуре деления. Ускорение деления может быть достигнуто и за счет более совершенной аппаратной реализации операции. Наилучшие результаты обычно дает сочетание обоих этих подходов. Ниже рассматриваются наиболее распространенные в настоящее время методы ускорения операции деления.

Алгоритм SRT

Свое название алгоритм SRT [68, 132, 157] получил по фамилиям авторов (Sweeney, Robertson, Tocher), разработавших его независимо друг от друга приблизительно в одно и то же время. Алгоритм представляет собой модификацию деления без восстановления остатка, где на каждом шаге помимо сдвига частичного остатка производится прибавление либо вычитание делителя. В SRT-алгоритме сдвиг ЧО также имеется в каждой итерации, однако сложение или вычитание, в зависимости от получающегося ЧО, на отдельных шагах может не выполняться, что, естественно, позитивно влияет на быстродействие устройства деления.

Алгоритм был ориентирован на операции над мантиссами чисел с плавающей запятой и опирается на то обстоятельство, что мантиссы в таких числах нормализованы. Впервые SRT-алгоритм был реализован в модели 91 вычислительной машины IBM 360. В настоящее время он широко применяется в блоках обработки чисел с плавающей запятой, в частности в микропроцессорах фирмы Intel.

Сначала рассмотрим алгоритм применительно к положительным целым числам. Делимое представляется $(2n + 1)$-разрядным числом, а делитель — n-разрядным.

Процедура деления начинается с удаления в делителе всех нулей, предшествующих старшей единице, то есть с операции, аналогичной нормализации мантиссы в числах с плавающей запятой. По той же аналогии будем в дальнейшем условно называть эту операцию нормализацией. Исключение k предшествующих нулей реализуется за счет сдвига делителя влево на k разрядов. На аналогичное число разрядов влево сдвигается и делимое. Далее выполняются n итераций, в которых вычисляются цифры частного и частичные остатки. Действия, выполняемые на i-й итерации, можно описать следующим образом:

$$q_i = \begin{cases} 1, \text{ если } 2R^{(i-1)} \geq B \\ 0, \text{если } -B \leq 2R^{(i-1)} < B \\ -1, \text{ если } 2R^{(i-1)} < -B \end{cases}$$

$$R^{(i)} = 2R^{i-1} - q_i B.$$

Обратим внимание на то, что частное представляется в избыточной системе счисления, где цифры частного могут иметь три значения: –1, 0, 1.

По завершении всех n итераций, если последний остаток отрицателен, выполняется коррекция этого остатка и полученного частного, для чего к остатку прибавляется делитель, а из частного вычитается единица с весом младшего разряда.

Последний момент в алгоритме — преобразование частного из системы $\{\bar{1}, 0, 1\}$ в систему $\{0, 1\}$, то есть в обычную двоичную систему.

На практике это выливается в следующую процедуру (при объяснении будем ссылаться на схему деления без восстановления остатка, приведенную на рис. 5.48). Делимое и делитель, представленные в дополнительном коде, размещаются в регистре делимого (РгА) и делителя (РгВ) соответственно. Дальнейшие действия можно описать следующим образом.

1. Если в делителе B имеются k предшествующих нулей (при $B > 0$) или предшествующих единиц (при $B < 0$), то производится предварительный сдвиг содержимого РгА и РгВ влево на k разрядов.
2. Для i от 0 до n–1:
   - ☐ если три старшие цифры частичного остатка в РгА совпадают, то $q_i = 0$ и производится сдвиг содержимого РгА на один разряд влево;
   - ☐ если три старшие цифры частичного остатка в РгА не совпадают, а сам ЧО отрицателен, то $q_i = -1$, делается сдвиг содержимого РгА на один разряд влево и к ЧО прибавляется делитель;
   - ☐ если три старшие цифры частичного остатка в РгА не совпадают, а сам ЧО положителен, то $q_i = 1$, выполняется сдвиг содержимого РгА на разряд влево и из ЧО вычитается делитель.
3. Если после завершения пункта 2 остаток отрицателен, то производится коррекция (к остатку прибавляется делитель, а из частного вычитается единица).
4. Остаток сдвигается вправо на k разрядов.

Описанную процедуру иллюстрирует пример, приведенный на рис. 5.49.

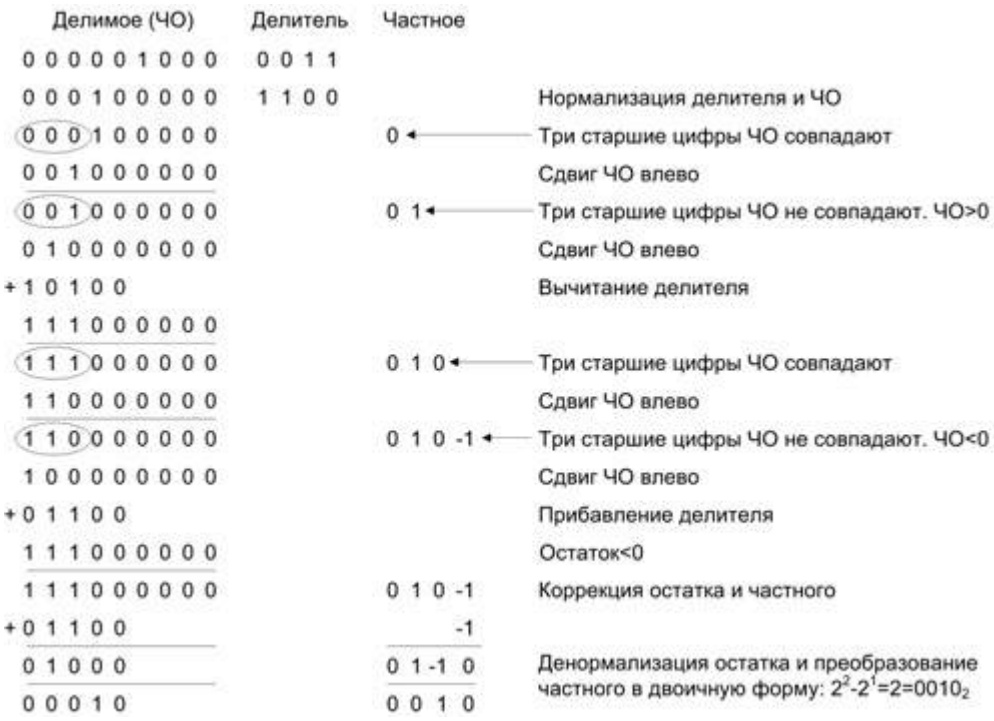


Рис. 5.49. Пример деления целых чисел по алгоритму SRT

На первом шаге для удаления предшествующих нулей делитель сдвигается на два разряда влево. Аналогично поступают и с ЧО, который вначале совпадает с делимым. Далее выполняется процедура, описанная выше в пункте 2. Операция вычитания B обеспечивается прибавлением делителя с противоположным знаком. Поскольку по завершении операции остаток отрицателен, производится его коррекция путем прибавления B. Одновременно частное уменьшается на единицу (эта операция показана в системе $\{1, 0, 1\}$, в которой представлено частное). Наконец, на последнем шаге форма представления частного меняется — переходит к представлению в стандартной двоичной системе.

В стандартном алгоритме деления без восстановления остатка помимо сдвига в каждой итерации выполняется операция сложения или вычитания. В варианте SRT, в зависимости от кодов операндов в отдельных итерациях, достаточно только сдвига, что, безусловно, ускоряет процесс деления. Согласно статистическим данным, в среднем число сложений и вычитаний при использовании этого алгоритма сокращается в 2,67 раза.

Представление частного в избыточной системе счисления

Наиболее распространенные методы ускорения операции деления основаны на применении алгоритмов, где частное представляется в избыточной системе счисления. Очередная цифра частного в избыточной системе счисления, в зависимости

от базы этой системы, соответствует двум или более цифрам в двоичном представлении частного, и для вычисления нужного количества двоичных цифр частного и остатка требуется меньше итераций. В то же время реализация такого подхода ведет к усложнению аппаратуры делителя, в частности надстраивается логика определения операции, выполняемой в очередной итерации. Для этой цели в состав устройства деления включается специальная память, хранящая таблицу для определения необходимых действий (в зависимости от текущей комбинации цифр в частичном остатке и делителе). Тем не менее выигрыш в быстродействии оказывается решающим моментом. Так, в микропроцессорах Pentium при делении мантисс чисел с плавающей запятой используется алгоритм SRT с базой 4 (Radix-4), то есть частное сначала вычисляется с использованием цифр –2, –1, 0, 1, 2 с последующим преобразованием результата к стандартному двоичному представлению. В этом варианте выбор очередной цифры частного производится с помощью таблицы, состоящей из отдельных секций. Конкретную секцию определяют четыре старшие цифры делителя (после его нормализации). Входом в секцию служат шесть старших цифр частичного остатка. ЧО в каждой итерации сдвигается не на один, а на два разряда, то есть число итераций сокращается вдвое. В микропроцессорах Nehalem и Penryn фирмы Intel деление как целых чисел, так и мантисс чисел с плавающей запятой реализуется блоком, получившим название *Fast Radix-16 Divider*. Поскольку используется система счисления с основанием 16, блок способен обрабатывать сразу 4 бита за такт, благодаря чему новые микропроцессоры выполняют операции деления целых и вещественных чисел примерно в два раза быстрее, чем в рассмотренном выше варианте, в которых использовалось основание 4 (Radix-4) и за один такт обрабатывались два бита.

Деление с использованием операции умножения

С появлением быстрых параллельных устройств умножения матричного и древовидного типа практическую значимость в плане ускорения операции деления получили алгоритмы, где эта операция сводится к умножениям, сложениям и сдвигам. Среди таких алгоритмов, прежде всего, следует упомянуть алгоритм Голдшмита (Robert Elliott Goldshmidt) и алгоритм Ньютона–Рафсона. Оба этих алгоритма особенно удобны при делении правильных дробей, в частности при делении мантисс в операциях с плавающей запятой.

Алгоритм Голдшмита

Этот алгоритм сводит операцию деления правильных дробей к последовательным операциям умножения. Представим операцию деления в следующем виде:

$$Q = \frac{A}{B} = \frac{A \times F_0 \times F_1 \times F_2 \times \ldots}{B \times F_0 \times F_1 \times F_2 \times \ldots}.$$

На каждом шаге числитель и знаменатель умножаются на очередной постоянный множитель F_i, при этом величина отношения не меняется. Последовательность множителей F_i выбирается таким образом, чтобы результирующий знаменатель сходился к единице:

$$B \times F_0 \times F_1 \times F_2 \times \ldots \Rightarrow 1.$$

Как следствие,

$$A \times F_0 \times F_1 \times F_2 \times \ldots \Rightarrow Q.$$

Поскольку речь идет о правильных дробях, то $0 < B < 1$, и в качестве первого множителя F_0 предлагается принять $F_0 = 2 - B$. Далее процесс деления сводится к рекуррентному вычислению очередной оценки числителя $A_i + 1$, знаменателя $B_i + 1$ и очередного множителя $F_i + 1$:

$$A_i + 1 = A_i \times F_i;\ B_i + 1 = B_i \times F_i;\ F_i + 1 = 2 - B_i,$$

где $i = 0, \ldots, k$, $A_0 = A$, $B_0 = B$.

Вычисления прекращаются на k-м шаге, когда выполнится условие $|1 - B_k| \le \Delta$ (Δ — *допустимая погрешность вычислений)*.

Проиллюстрируем алгоритм на примере деления десятичных дробей с допустимой погрешностью вычислений $\Delta = 0{,}001$. Для $A = 0{,}127$ и $B = 0{,}445$ при заданной погрешности имеем $Q = \frac{0{,}127}{0{,}445} = 0{,}285$. Деление в соответствии с алгоритмом Голдшмита можно описать следующим образом:

$A_0 = 0{,}127;$	$B_0 = 0{,}445;$	$F_0 = 1{,}555;$	$\lvert 1 - 0{,}445 \rvert = 0{,}555 \ge \Delta;$
$A_1 = 0{,}197;$	$B_1 = 0{,}692;$	$F_1 = 1{,}308;$	$\lvert 1 - 0{,}692 \rvert = 0{,}308 \ge \Delta;$
$A_2 = 0{,}258;$	$B_2 = 0{,}905;$	$F_2 = 1{,}095;$	$\lvert 1 - 0{,}905 \rvert = 0{,}095 \ge \Delta;$
$A_3 = 0{,}283;$	$B_3 = 0{,}991;$	$F_3 = 1{,}009;$	$\lvert 1 - 0{,}991 \rvert = 0{,}009 \ge \Delta;$
$A_4 \approx 0{,}285;$	$B_4 \approx 1{,}000;$		$\lvert 1 - 1 \rvert \approx 0{,}000 < \Delta.$

После 4-й итерации ($k = 4$) условие завершения вычислений выполнено, поэтому $Q = A_4 \approx 0{,}285$.

Метод Голдшмита используется в устройстве деления микропроцессора Athlon и последующих микропроцессорах фирмы AMD.

Алгоритм Ньютона—Рафсона

Идея алгоритма заключается в замене операции деления на B умножением делимого на величину, обратную делителю ($\frac{1}{B}$): $Q = \frac{A}{B} = A \times \frac{1}{B}$. В этом случае проблема сводится к эффективному вычислению $\frac{1}{B}$. Задача решается методом Ньютона–Рафсона путем нахождения корня уравнения

$$f(X) = \frac{1}{X} - B = 0,$$

то есть $X = \frac{1}{B}$. Решение может быть получено с привлечением рекуррентного соотношения: $X_{i+1} = X_i(2 - X_i B)$. Начальное значение оценки X_0 для нормализованного делителя ($0{,}5 \le B < 1$) вычисляется по формуле $X_0 = 2{,}9142 - 2B$. Вычисление X завершается, когда будет выполнено условие $|X_{i+1} - X_i| \le \Delta$. Для 32-разрядных чисел

обычно достаточно четырех итераций, а для 64-разрядных — пяти. Поэтому можно считать, что реализация метода для n-разрядных чисел требует $2\lfloor \log_2 n \rfloor - 1$ операций умножения.

Для иллюстрации метода воспользуемся примером из предыдущего раздела.

$A_0 = 0{,}127;$ $B_0 = 0{,}445;$ $X_0 = 2{,}024;$

$X_1 = 2{,}225;$ $|2{,}225 - 2{,}024| = 0{,}201 \geq \Delta;$

$X_2 = 2{,}247;$ $|2{,}247 - 2{,}225| = 0{,}022 \geq \Delta;$

$X_3 \approx 2{,}247;$ $|2{,}247 - 2{,}247| \approx 0{,}000 < \Delta;$

После третьей итерации имеем $\frac{1}{B} = X_3 = 2{,}247$ и $Q = 0{,}127 \times 2{,}247 = 0{,}285$.

Ускорение вычисления частичных остатков

Возможности данного подхода весьма ограничены и связаны в основном с ускорением операций сложения (вычитания). Способы достижения этой цели ничем не отличаются от тех, что применяются, например, при выполнении умножения. Это различные приемы для убыстрения распространения переноса, матричные схемы сложения и т. п.

Матричная схема деления

Для ускорения вычисления частичных остатков может быть применена матричная схема [7], показанная на рис. 5.50.

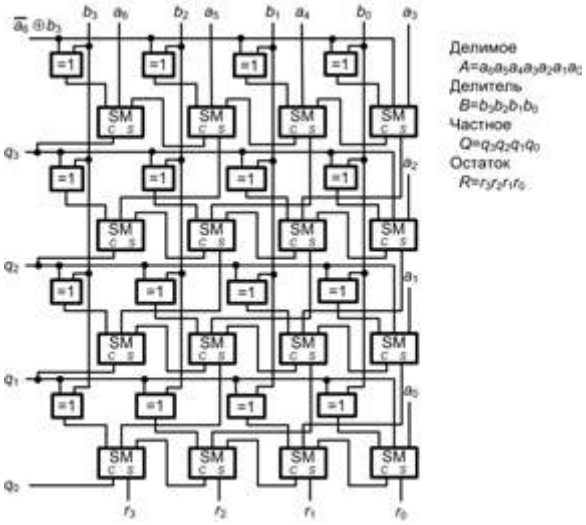


Рис. 5.50. Матричное устройство деления для алгоритма без восстановления остатка

В роли строк матрицы выступают n-разрядные параллельные сумматоры. В каждой следующей строке вычисляется очередной частичный остаток. Поскольку схема реализует алгоритм деления без восстановления остатка, сумматоры обеспечивают прибавление или вычитание делителя в зависимости от знака предыдущего ЧО. Как видно из рисунка, вычитание реализуется так же, как это было описано при рассмотрении операционного блока для сложения и вычитания (см. рис. 5.15), то есть путем прибавления делителя, взятого с обратным знаком и представленного в дополнительном коде. Результирующие частное и остаток будут представлены в дополнительном коде. Окончательный остаток не всегда верен и может нуждаться в коррекции.

Нетрудно заметить, что схеме присущ длинный тракт распространения переноса, что явно не способствует сокращению времени деления. Таким образом, матричное исполнение деления нельзя считать очень эффективным в плане быстродействия, хотя данный метод и выигрывает в сравнении с традиционным устройством деления. Основное достоинство матричной схемы заключается в ее регулярности, что удобно при реализации устройства в виде интегральной микросхемы.

Операционные устройства для чисел в форме с плавающей запятой

Операции над числами в форме с плавающей запятой (ПЗ) имеют существенные отличия от аналогичных операций в форме с фиксированной запятой, поэтому их обычно реализуют с помощью самостоятельного операционного устройства. Как и целочисленное ОПУ, операционное устройство для чисел в форме ПЗ как минимум должно обеспечивать выполнение четырех арифметических действий: сложения, вычитания, умножения и деления. Кроме того, в таких ОПУ в качестве стандартной предусмотрена также операция извлечения квадратного корня.

После принятия стандарта IEEE 754 негласным требованием ко всем ВМ является обеспечение операций с числами, представленными в форматах стандарта. Это требование не исключает использования в конкретной ВМ также иных форматов чисел с ПЗ, что обычно связано с сохранением совместимости с предыдущими моделями данной вычислительной машины.

Напомним основные положения записи чисел в стандарте IEEE 754 (рис. 5.51). Мантиссы чисел m представляются в нормализованном виде, при этом действует прием скрытого разряда, когда старшая цифра мантиссы, всегда равная единице, в записи числа отсутствует, то есть в поле мантиссы старшей является вторая цифра нормализованной мантиссы.

В отличие от общепринятого условия нормализации $\frac{1}{2} \leqslant |m| < 1$, в стандарте IEEE 754 используется условие $1 \leqslant |m| < 2$.

Запись числа содержит смещенный порядок, то есть порядок, увеличенный на величину смещения, которое в стандарте IEEE 754 для одинарного формата равно 127, а для двойного — 1023.

Рис. 5.51. Представление чисел в форме с плавающей запятой по стандарту IEEE 754 (одинарный формат)

С учетом перечисленных особенностей, арифметическую операцию над числами в форме с плавающей запятой можно записать в виде

$$\pm m_Z 2^{p_Z} = (\pm m_X 2^{p_X}) \lozenge (\pm m_Y 2^{p_Y}),$$

где m_X, m_Y, m_Z — нормализованные мантиссы операндов и результата; p_X, p_Y, p_Z — смещенные порядки операндов и результата; $\lozenge$ — знак арифметической операции.

При выполнении арифметических операций следует учитывать ряд особенностей стандарта. Так, поскольку нулевое значение не может быть представлено в нормализованном виде, оно обозначается кодом, содержащим нули во всех позициях, за исключением позиции знака (предусмотрены значения +0 и –0). Специальные коды используются и для представления значений $\pm \infty$, а также неопределенного результата, обычно называемого NAN (not a number). NAN применяют для обозначения, например, 0/0. В случае появления одного из этих значений в качестве операнда арифметической операции, результат ее определяется по правилам, также являющимся частью стандарта. Например,

- обычное число / $(+\infty) = \pm 0$;
- $(+\infty) \times$ обычное число $= \pm \infty$;
- NAN + обычное число = NAN.

Специальные коды позволяют вместо остановки вычислений оттянуть принятие решения до конца выполнения операции.

При всех различиях в выполнении разных арифметических операций подготовительный и заключительный этапы во всех случаях практически совпадают, в силу чего рассмотрим их отдельно.

Подготовительный этап

Первой особенностью операционных устройств для чисел с плавающей запятой является то, что в них операции над тремя составляющими чисел с ПЗ (знаками, мантиссами и порядками операндов) выполняются раздельно: блоком обработки знаков (БОЗ), блоком обработки порядков (БОП) и блоком обработки мантисс (БОМ). Для хранения операндов и результата в ОПУ предусмотрены соответствующие регистры. Хотя эти регистры могут быть физически реализованы в виде единых устройств, каждый из них логично рассматривать как совокупность трех регистров: знака, порядка и мантиссы. Таким образом, на этапе загрузки операндов в регистры ОПУ осуществляется «распаковка» чисел с ПЗ — разбиение на три составляющие. В ходе такой распаковки в старшем разряде регистра мантиссы

восстанавливается единица, которая в записи числа отсутствовала (была скрыта). Если операнды представлены в одном из дополнительных форматов, предусмотренных в стандарте IEEE 754, они преобразуются во внутренний формат операционного устройства.

На предварительном этапе выполняется также проверка на совпадение кодов одного или обоих операндов со специальными кодами, обозначающими такие значения, как ± 0, ± ∞ или NAN. Это позволяет исключить ненужные операции. Так, в операциях умножения и деления, если нулю равны множитель, множимое или делимое, в качестве результата сразу же можно принять нулевое значение, обойдя предписанные данными операциями действия.

Заключительный этап

Действия на завершающем этапе выполнения любой арифметической операции идентичны и сводятся к выявлению и учету *исключений*, нормализации мантиссы, округлению мантиссы и «упаковке» составляющих результата (знака, порядка и мантиссы).

Сначала остановимся на учете исключений. Так принято называть «нештатные» ситуации, возникающие в процессе выполнения операции или в процессе представления результата в стандартной форме.

Первая такая ситуация связана с нулевым значением мантиссы, которое может получиться в результате операции. В этом случае имеет место ситуация *потери значимости мантиссы* или *антипереполнение*, и результат операции принимается равным нулю. Это означает, что нормализацию мантиссы производить не нужно, а результат можно сразу представить специальным кодом 0 с соответствующим знаком.

Результатом арифметической операции может стать число, порядок которого превышает наибольшее возможное значение, которое можно записать в поле порядка, то есть имеет место *переполнение*. Такая ситуация может, например, возникнуть при сложении мантисс с одинаковым знаком или вычитании мантисс с разными знаками, когда результирующая мантисса не укладывается в поле мантиссы. В этом случае мантисса результата сдвигается вправо на один разряд, а порядок результата увеличивается на единицу, что, в свою очередь, чревато переполнением поля порядка. При возникновении переполнения в качестве результата берутся специальные коды, обозначающие $+\infty$ или $-\infty$ (в зависимости от знака мантиссы), и формируется признак *переполнения*.

При получении результата типа NAN формируется признак «*неправильная операция*». Такое возможно, например, в следующих случаях:

- Сложение: $(+\infty) + (-\infty)$;
- Умножение: $0 \times \infty$;
- Деление: $0/0$ или ∞/∞;
- Квадратный корень: Операнд < 0.

При получении «нормального» результата очередным шагом заключительного этапа является нормализация мантиссы. Она сводится к последовательному сдвигу мантиссы влево до тех пор, пока старшую позицию не займет единица. Каждый сдвиг сопровождается уменьшением на единицу порядка результата. Иногда с целью ускорения процесса нормализации операционное устройство содержит схему определения количества нулей, предшествующих старшей единице мантиссы. Если количество таких нулей равно k, то мантисса сдвигается влево сразу на k разрядов, а порядок результата уменьшается на k. Из-за того, что при нормализации порядок результата уменьшается, он может стать отрицательным, что для смещенных порядков свидетельствует о получении числа, непредставимого в данном формате. В такой ситуации результат принимается равным нулю, заменяется соответствующим специальным кодом, и одновременно формируется признак *потери значимости порядка*.

Следующий шаг заключительного этапа — усечение или округление результата, связанное с преобразованием значений, представленных с повышенной точностью, в формат меньшей точности, используемый при хранении и/или выводе результата. Округление может повлечь необходимость еще одного сдвига для устранения переполнения мантиссы и коррекции порядка со всеми описанными ранее последствиями.

Далее из нормализованной мантиссы удаляется старший разряд, который для всех чисел (кроме представленных специальными кодами) равен единице. Отметим, что в стандарте IEEE 754 для повышения точности представления маленьких чисел может использоваться ненормализованная запись мантиссы, где старшая единица не удаляется (не скрывается).

В последней фазе, если результат не относится к специальным, осуществляется «упаковка» всех его составляющих (знака, порядка и мантиссы), после чего сформированный результат заносится в выходной регистр ОПУ.

Сложение и вычитание

В арифметике с плавающей запятой сложение и вычитание — более сложные операции, чем умножение и деление. Обусловлено это необходимостью выравнивания порядков операндов. Алгоритм сложения и вычитания включает в себя следующие основные фазы.

1. Подготовительный этап.
2. Определение операнда, имеющего меньший порядок, и сдвиг его мантиссы вправо на число разрядов, равное разности порядков операндов.
3. Приравнивание порядка результата большему из порядков операндов.
4. Сложение или вычитание мантисс и определение знака результата.
5. Заключительный этап.

Структуру устройства для сложения/вычитания чисел с плавающей запятой иллюстрирует рис. 5.52 [126].

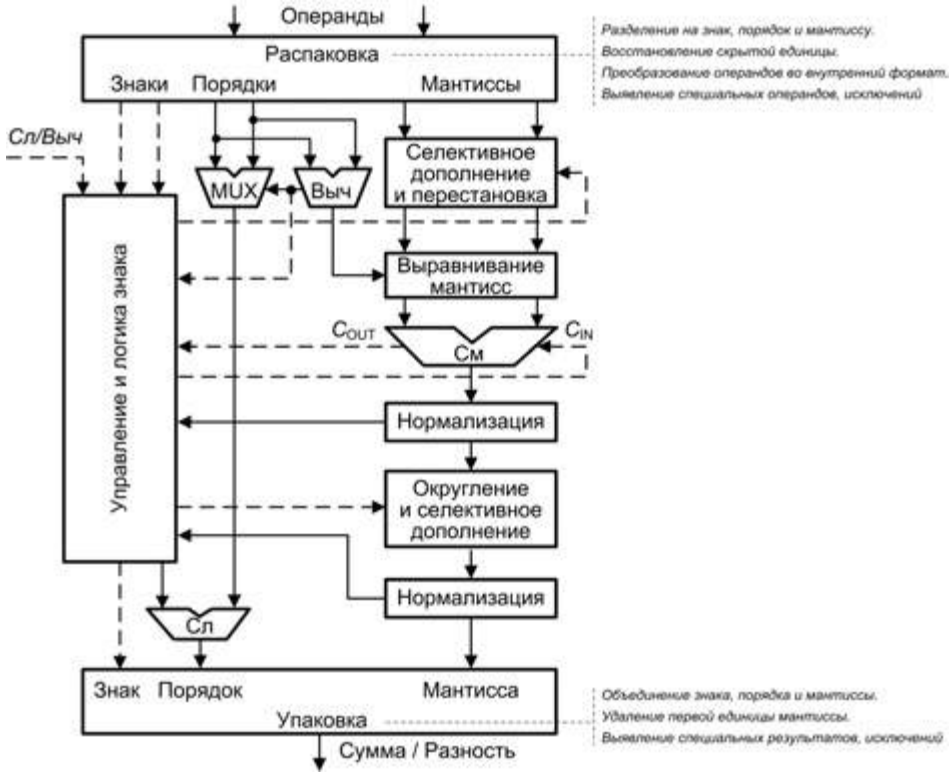


Рис. 5.52. Структура устройства сложения/вычитания чисел с плавающей запятой

Устройство сложения/вычитания состоит из сумматора для сложения выровненных мантисс и схем, обеспечивающих операции со знаками и порядками операндов, предварительный сдвиг мантисс (*пресдвиг*) при выравнивании мантисс, сдвиг мантиссы результата при нормализации (*постсдвиг*), формирование специальных кодов (0, ∞ и т. д.).

После завершения подготовительного этапа осуществляется такое преобразование одного из исходных чисел, чтобы порядки обоих операндов стали равны. Для пояснения рассмотрим пример сложения десятичных чисел с ПЗ:

$$123 \times 10^0 + 456 \times 10^{-2}.$$

Очевидно, что непосредственное сложение мантисс недопустимо, поскольку цифры мантисс, имеющие одинаковый вес, должны располагаться в эквивалентных позициях. Так, цифра 4 во втором числе должна суммироваться с цифрой 3 в первом. Этого можно добиться, если записать второе число так, чтобы порядки обоих чисел были равны:

$$123 \times 10^0 + 456 \times 10^{-2} = 123 \times 10^0 + 4{,}56 \times 10^0 = 127{,}56 \times 10^0.$$

Выравнивания порядков можно достичь сдвигом мантиссы меньшего из чисел вправо, с одновременным увеличением порядка этого числа, либо сдвигом манти-

ссы большего из чисел влево и уменьшением его порядка. Оба варианта сопряжены с потерей цифр мантиссы, но выгоднее сдвигать меньшее из чисел, так как при этом теряются младшие разряды мантиссы. Таким образом, выравнивание порядков операндов реализуется путем сдвига мантиссы меньшего из чисел на число разрядов, равное разности порядков (такой сдвиг часто называют пресдвигом). В качестве порядка результата берется больший из порядков операндов. Для экономии оборудования возможность пресдвига часто обеспечивается только для одного из операндов (первого или второго), при этом возможен взаимный обмен местами содержимого регистров мантисс. Это обеспечивается блоком селективного дополнения и перестановки (СДП).

Следующая фаза — сложение мантисс, осуществляемое с помощью параллельного сумматора/вычитателя, аналогичного сумматору для чисел с фиксированной запятой (см. рис. 5.15). Поскольку в процессе пресдвига младшие разряды сдвигаемой вправо мантиссы могут быть потеряны, разрядность сумматора обычно больше разрядности мантиссы. Это позволяет восстановить «вытолкнутые» разряды в ходе нормализации результата (при нормализации мантисса результата сдвигается влево).

Сложение мантисс производится с учетом их знаков. При сложении операндов с одинаковыми знаками их знак в процессе суммирования игнорируется и просто добавляется на заключительном этапе. Если один из операндов отрицателен, то его мантисса должна быть представлена в дополнительном коде. Такое преобразование осуществляется блоком СДП. Как и в случае пресдвига, для экономии оборудования возможность преобразования в дополнительный код может быть предусмотрена только для одного из операндов, например для второго (Y). Если отрицателен первый операнд ($-X$), то его знак игнорируется, и выполняется вычитание, что приводит к результату $X - Y$ вместо $-X + Y$. Разница в знаках учитывается в блоке управления и логики знака (УЛЗ) при определении правильного знака результата.

Так как суммирование мантисс производится в дополнительном коде, а в записи числа с плавающей запятой мантисса должна быть представлена в прямом коде, то в случае отрицательной мантиссы результата она должна быть преобразована в прямой код. Такое преобразование также обеспечивает УЛЗ. Кроме того, УЛЗ отвечает за выборочное (селективное) преобразование в дополнительный код одного из операндов и определение знака результата в зависимости от выполняемой операции (сложение или вычитание).

Процедуру выравнивания порядков и сложения мантисс для случая $p_X > p_Y$ формально можно описать следующим образом:

$$\pm m_Z q^{p_Z} = (\pm m_X q^{p_X}) + (\pm m_Y q^{p_Y}) = (\pm m_X q^{p_X}) + (\pm m_Y / q^{p_X - p_Y}) q^{p_X} =$$
$$= (\pm m_X \pm m_Y / q^{p_X - p_Y}) q^{p_X} = \pm m_Z q^{p_Z},$$

где q — основание системы счисления; m_X, m_Y и m_Z — мантиссы, а p_X, p_Y, p_Z — смещенные порядки операндов и результата соответственно.

Далее выполняется описанный выше заключительный этап.

В отличие от целочисленной арифметики, в операциях с ПЗ сложение и вычитание производятся приближенно, так как при выравнивании порядков может происходить потеря младших разрядов одного из слагаемых.

Умножение

ОПУ умножения чисел с плавающей запятой (рис. 5.53, *а*) состоит из устройства умножения чисел с фиксированной запятой для перемножения мантисс и дополнительных схем, обеспечивающих операции с порядками и учет специальных значений (0, ∞ и т. д.). Поскольку подготовительный и заключительный этапы для всех арифметических операций практически одинаковы, рассмотрим, главным образом, только ту часть ОПУ, которая связана с собственно умножением.

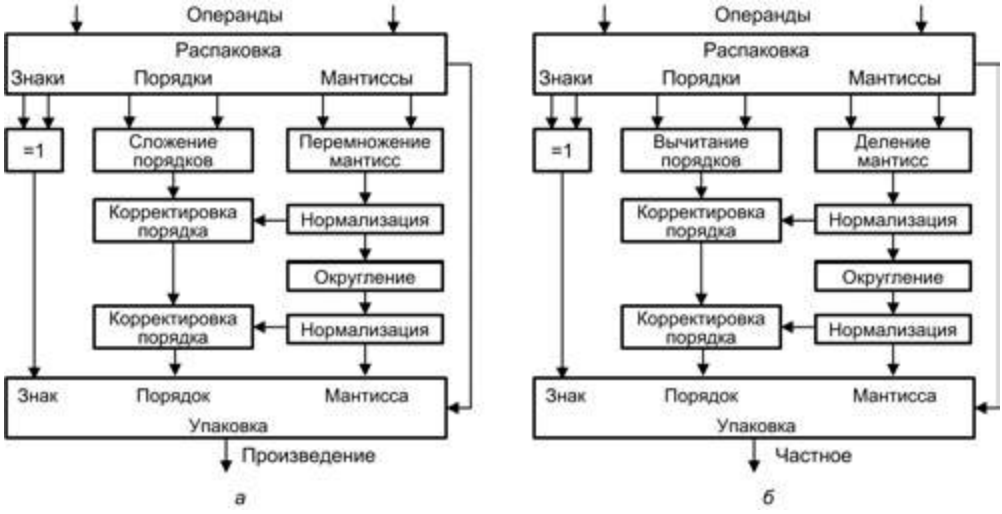


Рис. 5.53. Структура устройств умножения и деления чисел с плавающей запятой: *а* — устройство умножения; *б* — устройство деления

Знак произведения получается с помощью операции исключающего ИЛИ над знаками сомножителей.

Предварительный порядок результата вычисляется суммированием порядков сомножителей. Так как в стандарте IEEE 754 используется смещенный порядок, то в полученной сумме будет содержаться удвоенное смещение N, поэтому из нее необходимо вычесть величину смещения N. Результатом действий с порядками может стать как переполнение порядка, так и потеря значимости. В обоих случаях выполнение операции прекращается и выдается соответствующее сообщение (возникает прерывание).

Если порядок результата лежит в допустимых границах, на следующем шаге производится перемножение мантисс с учетом их знака, которое выполняется так же, как для чисел с фиксированной запятой. При размещении произведения мантисс в разрядной сетке необходимо учитывать, что мантиссы представлены не целыми

числами, а правильными дробями. Хотя результат умножения мантисс имеет удвоенную по сравнению с операндами длину, он округляется до длины поля мантиссы.

Описанные действия в случае смещенных порядков можно описать следующим образом:

$$\pm m_Z q^{p_Z} = (\pm m_X q^{p_X}) \times (\pm m_Y q^{p_Y}) = (\pm m_X \times m_Y) q^{p_X + p_Y - N}.$$

На последнем шаге выполняется типовая процедура заключительного этапа.

Деление

Устройство деления с плавающей запятой (рис. 5.53, *б*) имеет практически такую же структуру, что и устройство умножения. Операнды также проходят подготовительный этап, особенностью которого является проверка на случай деления на 0.

Знак частного определяется так же, как и при умножении — с помощью операции исключающего ИЛИ над знаками операндов.

Далее выполняется вычитание порядка делителя из порядка делимого, что приводит к удалению смещения из порядка результата (напомним, что в стандарте IEEE 754 используются смещенные порядки). Следовательно, для получения смещенного порядка результата к разности должно быть добавлено смещение. После выполнения этих действий необходима проверка на переполнение порядков и потерю значимости.

При использовании смещенных порядков процедура деления может быть формально описана как

$$m_Z q^{p_Z} = (\pm m_X q^{p_X}) / (\pm m_Y q^{p_Y}) = (\pm m_X / m_Y) q^{p_X - p_Y + N}.$$

Следующий шаг — деление мантисс как чисел с фиксированной запятой. Поскольку это наиболее медленная операция, выбору метода деления уделяется особое внимание. Так, в микропроцессорах Nehalem и Penryn фирмы Intel деление мантисс реализуется блоком, получившим название *Fast Radix-16 Divider*. В нем используется система счисления с основанием 16, поэтому блок способен обрабатывать сразу 4 бита за такт, благодаря чему новые микропроцессоры выполняют операции деления примерно в два раза быстрее, чем их предшественник (Pentium), где использовалось основание 4 и за один такт обрабатывались два бита.

Как и другие арифметические операции, деление завершается заключительным этапом.

Реализация логических операций

Помимо арифметических действий, ОПУ любой вычислительной машины предполагает выполнение основных логических операций и сдвигов. Чаще всего такие операции реализуются дополнительными схемами, входящими в состав целочисленных ОПУ.

К базовым логическим операциям относятся: логическое отрицание (НЕ), логическое сложение (ИЛИ) и логическое умножение (И). Этот набор, как правило, дополняют операцией сложения по модулю 2 (исключающее ИЛИ).

Булева переменная в ВМ представляется одним битом, однако на практике логические операции в ОПУ выполняются сразу над совокупностью логических переменных, объединенных в рамках одного байта или машинного слова, причем над всеми битами этого слова выполняется одна и та же операция. Поскольку каждый бит совокупности логических переменных рассматривается как независимая переменная, никакие переносы между разрядами не формируются. Операционный блок (ОПБ) для выполнения логических операций обеспечивает реализацию всех перечисленных логических операций. Возможная структура подобного ОПБ показана на рис. 5.54.

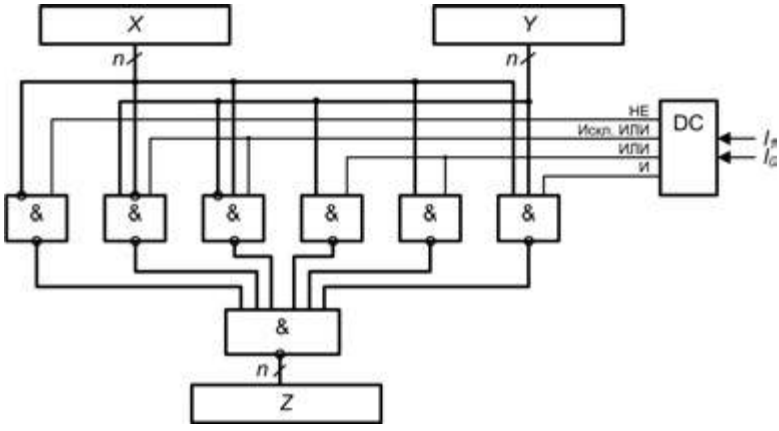


Рис. 5.54. Структура операционного блока для выполнения логических операций

Выбор нужной операции осуществляется с помощью двухразрядного управляющего кода L ($l_1 l_0$), с учетом которого выходная функция Z может быть описана выражением:

$$Z = \overline{l_1} \wedge \overline{l_0} \wedge \overline{X} \vee \overline{l_1} \wedge l_0 \wedge (\overline{X} \wedge Y \vee X \wedge \overline{Y}) \vee l_1 \wedge \overline{l_0} \wedge (X \vee Y) \vee l_1 \wedge l_0 \wedge (X \wedge Y).$$

Контрольные вопросы

1. Охарактеризуйте состав операционных устройств, входящих в АЛУ. Из каких соображений и каким образом он может изменяться?
2. Поясните понятие «операционные устройства с жесткой структурой». В чем заключается жесткость их структуры? Каковы их достоинства и недостатки?
3. Чем обусловлено название операционных устройств с магистральной структурой? Сравните магистральные структуры с жесткими структурами, выделяя достоинства, недостатки и область применения.
4. Дайте развернутую характеристику классификации операционных устройств с магистральной структурой. Поясните достоинства и недостатки «минимального» и «максимального» вариантов.
5. Поясните функциональный состав параллельного операционного блока магистрального ОПУ. Каким образом можно минимизировать количество внешних связей этого блока? Ответ сопроводите конкретным примером.

6. Чем обусловлена специфика целочисленного сложения и вычитания? Какую роль играет в них дополнительный код? К чему бы привел отказ от дополнительного кода? Ответ поясните на примерах. Как выявляется переполнение в этих операциях?
7. Какие вспомогательные системы счисления используются при создании операционных устройств умножения и деления? Чем обусловлено применение таких систем?
8. Каким образом в схеме операционного блока сложения/вычитания обеспечивается замена операции вычитания на операцию сложения?
9. В чем состоит различие между логическими и аппаратными методами ускорения умножения.
10. Попарно сравните алгоритм Бута, модифицированный алгоритм Бута, алгоритм Лемана.
11. Разработайте алгоритм умножения с обработкой за шаг трех разрядов множителя.
12. Поясните суть аппаратных методов ускорения умножения, выделив три возможных подхода.
13. Сравните умножители Брауна, Бо–Вули, Пезариса. Чем они схожи и в чем отличаются друг от друга?
14. В чем заключается основная идея древовидных умножителей? Каковы особенности их организации?
15. Сформулируйте, в чем состоит сходство и различие дерева Уоллеса, дерева Дадда и перевернутого ступенчатого дерева.
16. С какой целью и каким образом выполняется конвейеризация матричных и древовидных умножителей? Приведите (и поясните) конкретные примеры.
17. На конкретном примере объясните, как можно нарастить разрядность аппаратного умножителя.
18. Сравните организацию целочисленного деления с восстановлением остатка и без восстановления остатка. Как учитываются при делении знаки операндов?
19. Обоснуйте возможность совмещения структур умножителя и делителя. Опишите объединенную структуру.
20. Сформулируйте пути ускорения целочисленного деления. Сравните между собой их возможную реализацию.
21. Какие из операций с плавающей запятой считаются наиболее сложными? Ответ обоснуйте на конкретных примерах.
22. В чем состоит специфика умножения с плавающей запятой? Поясните эту специфику на примере.
23. Разработайте серию примеров для иллюстрации всех особенностей деления с плавающей запятой.
24. Создайте структуру операционного блока для выполнения как сложения/вычитания, так и базового набора логических операций. Обоснуйте каждый элемент этой структуры.

Глава 6

Память

В любой ВМ, вне зависимости от ее архитектуры, данные[1] хранятся в запоминающих устройствах (ЗУ), которые, учитывая их характеристики, место расположения и способ взаимодействия с процессором, относят к внутренней или внешней памяти. Внутренняя память располагается частично на общем кристалле с процессором (регистры и кэш-память), а частично — на системной плате (основная память и, возможно, кэш-память 3-го и более высоких уровней). Медленные ЗУ большой емкости (твердотельные, магнитные и оптические диски, магнитные ленты) называют внешней памятью, поскольку к ВМ они подключаются аналогично устройствам ввода/вывода. Основными функциями ЗУ являются прием, хранение и выдача данных в процессе работы ВМ. Процесс приема данных, в ходе которого осуществляется их занесение в ЗУ, называется *записью*, процесс выдачи данных — *чтением* или *считыванием*, а совместно их определяют как *процессы обращения* к ЗУ.

Характеристики запоминающих устройств внутренней памяти

Перечень основных характеристик, которые необходимо учитывать, рассматривая конкретный вид ЗУ, включает в себя:

- емкость;
- единицу пересылки;
- метод доступа;
- быстродействие;
- физический тип;
- физические особенности;
- стоимость.

[1] ГОСТ 15971-90 определяет данные как информацию, представленную в виде, пригодном для обработки автоматическими средствами при возможном участии человека.

Емкость ЗУ характеризуют числом битов, либо байтов, которое одновременно может храниться в запоминающем устройстве. На практике применяются более крупные единицы, а для их обозначения к словам «бит» или «байт» добавляют приставки: кило, мега, гига, тера, пета, экса, зетта, йотта (kilo, mega, giga, tera, peta, exa, zetta, yotta). Стандартно эти приставки означают умножение основной единицы измерений на 10^3, 10^6, 10^9, 10^{12}, 10^{15}, 10^{18}, 10^{21} и 10^{24} соответственно. В вычислительной технике, ориентированной на двоичную систему счисления, они соответствуют значениям, достаточно близким к стандартным, но представляющим собой целую степень числа 2, то есть 2^{10}, 2^{20}, 2^{30}, 2^{40}, 2^{50}, 2^{60}, 2^{70}, 2^{80}. Во избежание разночтений в 2000 году МЭК — Международная электротехническая комиссия (IEC — International Electrotechnical Comission) утвердила стандарт IEC 60027-2, предусматривающий новые обозначения, в которых к основной приставке добавляется слог bi (от английского binary — двоичный). Так, если единица измерения емкости кратна байту, предлагаются следующие названия и обозначения: kibibyte (KiB), mebibyte (MiB), gibibyte (GiB), tebibyte (TiB), pebibyte (PiB), exbibyte (EiB), zettabyte (ZiB), yottabyte (YiB) [100]. В русской транскрипции — кибибайт (КиБ), мебибайт (МиБ), гибибайт (ГиБ), тебибайт (ТиБ) и т. д.

Важной характеристикой ЗУ является *единица пересылки*. Для основной памяти (ОП) единица пересылки определяется шириной шины данных, то есть количеством битов, передаваемых по линиям шины параллельно. Обычно единица пересылки равна длине слова, но не обязательно. Так, при пересылке информации между основной памятью и кэш-памятью данные передаются единицами, превышающими размер слова, и такие единицы называются блоками.

При оценке быстродействия необходимо учитывать применяемый в данном типе *ЗУ метод доступа* к данным. Различают четыре основных метода доступа: последовательный, прямой, произвольный и ассоциативный, из которых для внутренней памяти характерны два последних. В ЗУ с *произвольным доступом* ячейка[1] имеет уникальный физический адрес. Обращение к любой ячейке занимает одно и то же время и может производиться в любой последовательности (произвольной очередности). Примером могут служить запоминающие устройства основной памяти. *Ассоциативный доступ* позволяет обращаться к ячейкам ЗУ в соответствии с признаками хранимых в них данных, он обеспечивает поиск ячеек, содержащих такую информацию, в которой значение отдельных битов совпадает с состоянием одноименных битов в заданном образце. Сравнение осуществляется параллельно для всех ячеек памяти. По ассоциативному принципу построены некоторые блоки кэш-памяти.

Быстродействие ЗУ является одним из важнейших его показателей. Для количественной оценки быстродействия обычно используют четыре параметра:

- *Время выборки данных.* Оно соответствует интервалу времени между началом операции считывания и выдачей считанных данных из запоминающего устройства.

[1] ЗУ внутренней памяти представляет собой массив ячеек, каждая из которых способна хранить единицу информации (обычно 1 байт).

- *Время хранения данных* — интервал времени, в течение которого запоминающее устройство в заданном режиме сохраняет данные без регенерации.
- *Цикл обращения к ЗУ* или *период обращения* ($T_Ц$). Им называют минимальный интервал времени между двумя последовательными доступами к данным запоминающего устройства. Период обращения включает в себя собственно время доступа плюс некоторое дополнительное время. Дополнительное время может требоваться для затухания сигналов на линиях, а в некоторых типах ЗУ, где считывание информации приводит к ее разрушению, — для восстановления считанной информации.
- *Скорость передачи данных* — количество данных, считываемых (записываемых) запоминающим устройством в единицу времени.

Говоря о *физическом типе* запоминающего устройства, необходимо отметить, что ЗУ внутренней памяти современных вычислительных машин базируются на полупроводниковой технологии.

В зависимости от примененной технологии следует учитывать и ряд *физических особенностей* ЗУ. Так, для полупроводниковой технологии приходится учитывать фактор энергозависимости. В энергозависимой памяти информация может быть искажена или потеряна при отключении источника питания, в то время как в энергонезависимых ЗУ записанная информация сохраняется и при отключении питающего напряжения. Полупроводниковая память может быть как энергозависимой, так и нет, в зависимости от ее типа. Помимо энергозависимости нужно учитывать, приводит ли считывание информации к ее разрушению.

Стоимость ЗУ принято оценивать отношением общей стоимости ЗУ к его емкости в битах, то есть стоимостью хранения одного бита информации.

Иерархия запоминающих устройств

Память часто называют «узким местом» фон-неймановских ВМ из-за ее серьезного отставания по быстродействию от процессоров, причем разрыв этот неуклонно увеличивается. Так, если производительность процессоров возрастает вдвое примерно каждые 1,5 года, то для микросхем памяти прирост быстродействия не превышает 9% в год (удвоение за 10 лет), что выражается в увеличении разрыва в быстродействии между процессором и памятью приблизительно на 50% в год.

Если проанализировать используемые в настоящее время типы ЗУ, выявляется следующая закономерность:

- чем меньше время выборки, тем выше стоимость хранения бита;
- чем больше емкость, тем ниже стоимость хранения бита, но больше время выборки.

При создании системы памяти постоянно приходится решать задачу обеспечения требуемой емкости и высокого быстродействия за приемлемую цену. Наиболее распространенным подходом здесь является построение системы памяти ВМ по иерархическому принципу. *Иерархическая память* состоит из ЗУ различных типов (рис. 6.1), которые, в зависимости от характеристик, относят к определенному

уровню иерархии. Более высокий уровень меньше по емкости, быстрее и имеет большую стоимость в пересчете на бит, чем более низкий уровень. Уровни иерархии взаимосвязаны: все данные на одном уровне могут быть также найдены на более низком уровне, и все данные на этом более низком уровне могут быть найдены на следующем нижележащем уровне и т. д.

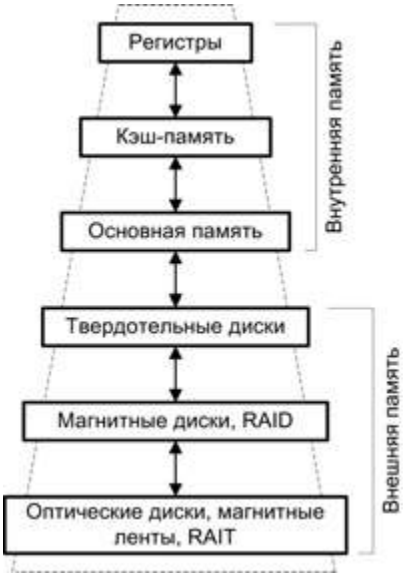


Рис. 6.1. Иерархия запоминающих устройств

Три верхних уровня иерархии образуют *внутреннюю память* ВМ, а все нижние уровни — это *внешняя* или *вторичная* память[1]. По мере движения вниз по иерархической структуре:

1) уменьшается соотношение «стоимость/бит».
2) возрастает емкость.
3) растет время выборки.
4) уменьшается частота обращения к памяти со стороны центрального процессора.

Если память организована в соответствии с пунктами 1–3, а характер размещения в ней данных удовлетворяет пункту 4, иерархическая организация ведет к уменьшению общей стоимости при заданном уровне производительности.

Справедливость этого утверждения вытекает из принципа *локальности по обращению* [76]. Если рассмотреть процесс выполнения большинства программ, то можно заметить, что с очень высокой вероятностью адрес очередной команды программы

[1] Самый нижний уровень иерархии, представленный ЗУ со сменными носителями, благодаря чему обеспечивается практически неограниченная емкость, часто называют третичной памятью.

либо следует непосредственно за адресом, по которому была считана текущая команда, либо расположен вблизи него. Такое расположение адресов называется *пространственной локальностью программы*. Обрабатываемые данные, как правило, структурированы, и такие структуры обычно хранятся в последовательности смежных ячеек памяти. Данная особенность программ называется *пространственной локальностью данных*. Кроме того, программы содержат множество небольших циклов и подпрограмм. Это означает, что небольшие наборы команд могут многократно повторяться в течение некоторого интервала времени, то есть имеет место *временная локальность*. Все три вида локальности объединяет понятие *локальность по обращению*. Принцип локальности часто облекают в численную форму и представляют в виде так называемого правила «90/10»: 90% времени работы программы связано с доступом к 10% адресного пространства этой программы.

Из свойства локальности вытекает, что программу разумно представить в виде последовательно обрабатываемых фрагментов — компактных групп команд и обрабатываемых ими данных. Помещая такие фрагменты в более быструю память, можно существенно снизить общие задержки на обращение, поскольку команды и исходные данные, будучи один раз переданы из медленного ЗУ в быстрое, затем могут использоваться многократно, и среднее время доступа к ним в этом случае определяется уже более быстрым ЗУ. Это позволяет хранить большие программы и массивы данных на медленных, емких, но дешевых ЗУ, а в процессе обработки активно использовать сравнительно небольшую быструю память, увеличение емкости которой сопряжено с высокими затратами.

На каждом уровне иерархии информация (данные) разбивается на *блоки*, выступающие в качестве наименьшей информационной единицы, пересылаемой между двумя соседними уровнями иерархии. Размер блоков может быть фиксированным либо переменным. При фиксированном размере блока емкость памяти обычно кратна его размеру. Размер блоков на каждом уровне иерархии чаще всего различен и увеличивается от верхних уровней к нижним.

При доступе к командам и исходным данным, например для их считывания, сначала производится поиск в памяти верхнего уровня. Факт обнаружения нужной информации называют *попаданием* (hit), в противном случае говорят о *промахе* (miss). При промахе производится поиск в ЗУ следующего более низкого уровня, где также возможны попадание или промах. После обнаружения необходимой информации выполняется пересылка блока, содержащего искомую информацию, с нижних уровней на верхние.

При оценке эффективности подобной организации памяти обычно используют следующие характеристики:

- *коэффициент попаданий* (hit rate) — отношение числа обращений к памяти, при которых произошло попадание, к общему числу обращений к ЗУ данного уровня иерархии;
- *коэффициент промахов* (miss rate) — отношение числа обращений к памяти, при которых имел место промах, к общему числу обращений к ЗУ данного уровня иерархии;

- *время обращения при попадании* (hit time) — время, необходимое для поиска нужной информации в памяти верхнего уровня (включая выяснение, является ли обращение попаданием), плюс время на фактическое считывание данных;
- *потери на промах* (miss penalty) — время, требуемое для замены блока в памяти более высокого уровня на блок с нужными данными, расположенный в ЗУ следующего (более низкого) уровня. Потери на промах включают в себя: *время доступа* (access time) — время обращения к первому слову блока при промахе и *время пересылки* (transfer time) — дополнительное время для пересылки оставшихся слов блока. Время доступа обусловлено задержкой памяти более низкого уровня, в то время как время пересылки связано с полосой пропускания канала между ЗУ двух смежных уровней.

Самый быстрый, но и минимальный по емкости тип памяти — это внутренние регистры ЦП, которые иногда объединяют понятием *сверхоперативное запоминающее устройство* — СОЗУ или *регистровый файл*. Как правило, количество регистров невелико, хотя в архитектурах с сокращенным набором команд их число может доходить до нескольких сотен. Основная память (ОП), значительно большей емкости, располагается ниже. Между регистрами ЦП и основной памятью часто размещают кэш-память, которая по емкости ощутимо проигрывает ОП, но существенно превосходит последнюю по быстродействию, уступая в то же время СОЗУ. Все виды внутренней памяти реализуются на основе полупроводниковых технологий и в основном являются энергозависимыми.

Долговременное хранение больших объемов данных обеспечивается внешними ЗУ, среди которых наиболее распространены запоминающие устройства на основе магнитных и оптических дисков, а также магнитоленточные ЗУ. В последнее время все большую популярность получают твердотельные диски на базе флэш-памяти.

Еще один уровень иерархии может быть добавлен между основной памятью и магнитными дисками. Этот уровень носит название дисковой кэш-памяти и реализуется в виде самостоятельного ЗУ, включаемого в состав магнитного диска. Дисковая кэш-память существенно улучшает производительность при обмене информацией между дисками и основной памятью.

Основная память

Основная память (ОП) представляет собой единственный вид памяти, к которой ЦП может обращаться непосредственно (исключение составляют лишь регистры центрального процессора). Информация, хранящаяся на внешних ЗУ, становится доступной процессору только после того, как будет переписана в основную память.

Основную память образуют запоминающие устройства с произвольным доступом. Такие ЗУ образованы как массив ячеек, а «произвольный доступ» означает, что обращение к любой ячейке занимает одно и то же время и может производиться в произвольной последовательности. Каждая ячейка содержит фиксированное число

запоминающих элементов и имеет уникальный адрес, позволяющий отличать ее от других ячеек.

Основная память может включать в себя два типа устройств: *оперативные запоминающие устройства* (ОЗУ) и *постоянные запоминающие устройства* (ПЗУ).

Преимущественную долю основной памяти образует ОЗУ, допускающее как запись, так и считывание информации, причем обе операции выполняются однотипно, практически с одной и той же скоростью. В англоязычной литературе ОЗУ соответствует аббревиатура RAM — *Random Access Memory*, то есть «память с произвольным доступом», что не совсем корректно, поскольку памятью с произвольным доступом являются также ПЗУ и регистры процессора. Для большинства типов полупроводниковых ОЗУ характерна энергозависимость — даже при кратковременном прерывании питания хранимая информация теряется. Микросхема ОЗУ должна быть постоянно подключена к источнику питания и поэтому может использоваться только как временная память.

Вторую группу полупроводниковых ЗУ основной памяти образуют энергонезависимые микросхемы ПЗУ (ROM — *Read-Only Memory*). ПЗУ обеспечивает считывание информации, но либо вообще не допускает ее изменения, либо процесс такого изменения (запись) сильно отличается от считывания и требует значительно большего времени.

Блочная организация основной памяти

Емкость основной памяти современных ВМ слишком велика, чтобы ее можно было реализовать на базе единственной интегральной микросхемы (ИМС). Необходимость объединения нескольких ИМС ЗУ возникает также, когда разрядность ячеек в микросхеме ЗУ меньше разрядности слов ВМ.

Увеличение разрядности ЗУ реализуется за счет объединения адресных входов объединяемых ИМС ЗУ. Информационные входы и выходы микросхем являются входами и выходами модуля ЗУ увеличенной разрядности (рис. 6.2). Полученную совокупность микросхем называют *модулем памяти*. Модулем можно считать и единственную микросхему, если она уже имеет нужную разрядность. Один или несколько модулей образуют *банк памяти*.

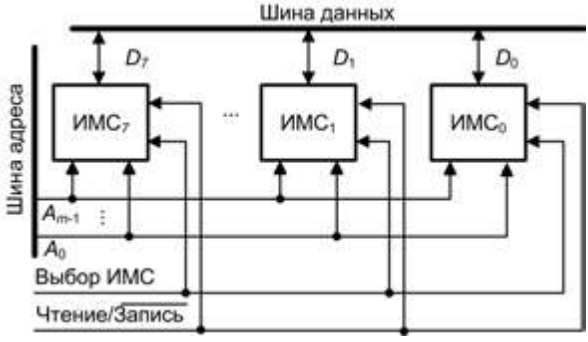


Рис. 6.2. Увеличение разрядности памяти

Для получения требуемой емкости ЗУ нужно определенным образом объединить несколько банков памяти меньшей емкости. В общем случае основная память ВМ практически всегда имеет блочную структуру, то есть содержит несколько банков.

При использовании блочной памяти, состоящей из B банков, адрес ячейки A преобразуется в пару (b, w), где b — номер банка, w — адрес ячейки внутри банка. Известны три схемы распределения разрядов адреса A между b и w:

- блочная (номер банка b определяет старшие разряды адреса);
- циклическая ($b = A \bmod B$; $w = A \operatorname{div} B$);
- блочно-циклическая (комбинация двух предыдущих схем).

Рассмотрение основных структур блочной ОП будем проводить на примере памяти емкостью 512 слов (2^9), построенной из четырех банков по 128 слов в каждом. Типовая структура памяти, организованная в соответствии с блочной структурой, показана на рис. 6.3.

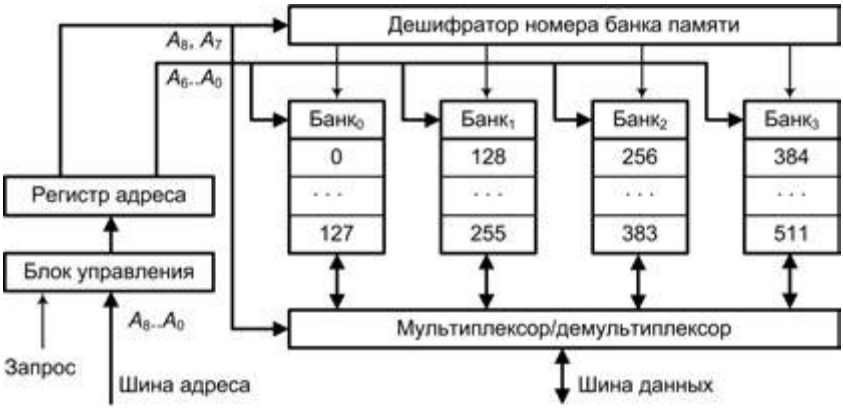


Рис. 6.3. Структура основной памяти на основе блочной схемы

Адресное пространство памяти разбито на группы последовательных адресов, и каждая такая группа обеспечивается отдельным банком памяти. Для обращения к ОП используется 9-разрядный адрес, семь младших разрядов которого (A_6–A_0) поступают параллельно на все банки памяти и выбирают в каждом из них одну ячейку. Два старших разряда адреса (A_8, A_7) содержат номер банка. Выбор банка обеспечивается либо с помощью дешифратора номера банка памяти, либо путем мультиплексирования информации (на рис. 6.3 показаны оба варианта). В функциональном отношении такая ОП может рассматриваться как единое ЗУ, емкость которого равна суммарной емкости составляющих, а быстродействие — быстродействию отдельного банка.

Расслоение памяти

Помимо податливости к наращиванию емкости, блочное построение памяти обладает еще одним достоинством — позволяет сократить время доступа к информации. Это возможно благодаря потенциальному параллелизму, присущему блочной

организации. Большей скорости доступа можно достичь за счет одновременного доступа ко многим банкам памяти. Одна из используемых для этого методик называется *расслоением памяти*. В ее основе лежит так называемое *чередование адресов* (address interleaving), заключающееся в изменении системы распределения адресов между банками памяти.

Прием чередования адресов базируется на ранее рассмотренном свойстве локальности по обращению, согласно которому последовательный доступ в память обычно производится к ячейкам, имеющим смежные адреса. Иными словами, если в данный момент выполняется обращение к ячейке с адресом 5, то следующее обращение, вероятнее всего, будет к ячейке с адресом 6, затем 7 и т. д. Чередование адресов обеспечивается за счет циклического разбиения адреса. В нашем примере (рис. 6.4) для выбора банка используются два младших разряда адреса (A_1, A_0), а для выбора ячейки в банке — 7 старших разрядов (A_8–A_2).

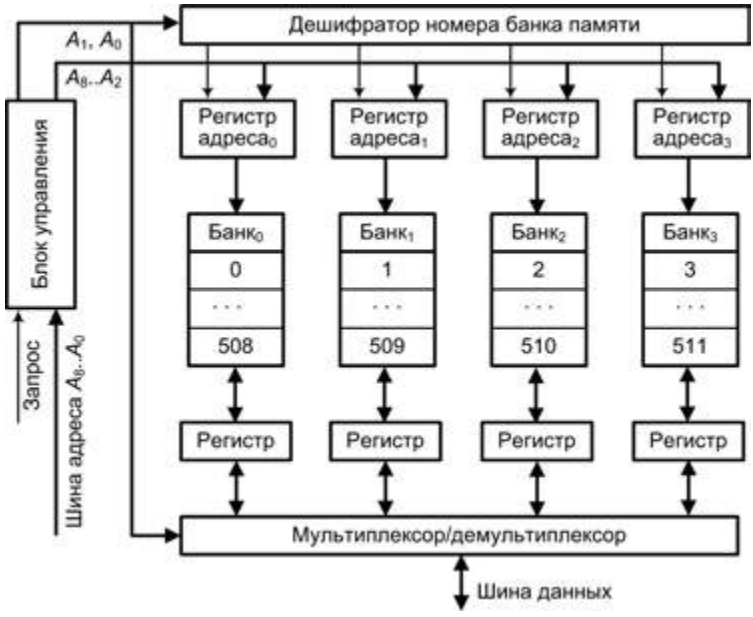


Рис. 6.4. Блочная память с чередованием адресов по циклической схеме

Поскольку в каждом такте на шине адреса может присутствовать адрес только одной ячейки, параллельное обращение к нескольким банкам невозможно, однако оно может быть организовано со сдвигом на один такт. Адрес ячейки запоминается в индивидуальном регистре адреса, и дальнейшие операции по доступу к ячейке в каждом банке протекают независимо. При большом количестве банков среднее время доступа к ОП сокращается почти в B раз (B — количество банков), но при условии, что ячейки, к которым производится последовательное обращение, относятся к разным банкам. Если же запросы к одному и тому же банку следуют друг за другом, каждый следующий запрос должен ожидать завершения обслуживания

предыдущего. Такая ситуация называется *конфликтом по доступу*. При частом возникновении конфликтов по доступу метод становится неэффективным.

В блочно-циклической схеме расслоения памяти каждый банк состоит из нескольких модулей, адресуемых по круговой схеме. Адреса между банками распределены по блочной схеме. Таким образом, адрес ячейки разбивается на три части. Старшие биты определяют номер банка, следующая группа разрядов адреса указывает на ячейку в модуле, а младшие биты адреса выбирают модуль в банке. Схему иллюстрирует рис. 6.5.

Рис. 6.5. Блочно-циклическая схема расслоения памяти

Традиционные способы расслоения памяти хорошо работают в рамках одной задачи, для которой характерно свойство локальности. В многопроцессорных системах с общей памятью, где запросы на доступ к памяти достаточно независимы, не исключен иной подход, который можно рассматривать как развитие идеи расслоения памяти. Для этого в систему включают несколько контроллеров памяти, что позволяет отдельным банкам работать совершенно автономно. Эффективность данного приема зависит от частоты независимых обращений к разным банкам. Лучшего результата можно ожидать при большом числе банков, что уменьшает вероятность последовательных обращений к одному и тому же банку памяти.

Синхронные и асинхронные запоминающие устройства

В качестве первого показателя, по которому можно классифицировать запоминающие устройства основной памяти, рассмотрим способ синхронизации. С этих позиций известные типы ЗУ подразделяются на синхронные и асинхронные.

В микросхемах, где реализован *синхронный принцип*, процессы чтения и записи (если это ОЗУ) выполняются одновременно с тактовыми сигналами контроллера памяти.

Асинхронный принцип предполагает, что момент начала очередного действия определяется моментом завершения предшествующей операции. Перенося этот принцип на систему памяти, необходимо принимать во внимание, что контроллер памяти всегда работает синхронно. В *асинхронных ЗУ* цикл чтения начинается только при поступлении запроса от контроллера памяти, и если память не успевает выдать данные в текущем такте, контроллер может считать их только в следующем такте, поскольку очередной шаг контроллера начинается с приходом очередного тактового импульса. В последнее время асинхронная схема активно вытесняется синхронной.

Организация микросхем памяти

Интегральные микросхемы (ИМС) памяти представляют собой массив запоминающих элементов (ЗЭ). Запоминающий элемент способен хранить 1 бит информации. Для ЗЭ любой полупроводниковой памяти характерны следующие свойства:

- два стабильных состояния, представляющие двоичные 0 и 1;
- в ЗЭ (хотя бы однажды) может быть произведена запись информации, посредством перевода его в одно из двух возможных состояний;
- для определения текущего состояния ЗЭ его содержимое может быть считано.

Каждый ЗЭ в микросхеме памяти имеет определенный адрес, на основании которого осуществляется доступ к данному ЗЭ. На физическую организацию массива однобитовых ЗЭ накладывается логическая организация памяти, то есть разрядность микросхемы *n*. Разрядность микросхемы определяет количество ЗЭ, имеющих один и тот же адрес (такая совокупность запоминающих элементов называется *ячейкой*).

Массив ЗЭ организован в виде совокупности из *n* матриц. Таким образом, *i*-й разряд всех ячеек хранится в *i*-й матрице массива.

При матричной организации ИМС памяти (рис. 6.6) реализуется координатный принцип адресации ячеек: адрес ячейки, поступающий по шине адреса ВМ, пропускается через логику выбора, где он разделяется на две составляющих: адрес строки и адрес столбца. Адреса строки и столбца запоминаются соответственно в регистре адреса строки и регистре адреса столбца микросхемы. Регистры соединены каждый со своим дешифратором. Выходы дешифраторов образуют систему горизонтальных и вертикальных линий, к которым подсоединены запоминающие элементы матрицы, при этом набор ЗЭ, образующий ячейку, расположен на пересечении одной горизонтальной и одной вертикальной линии.

Запоминающие элементы, объединенные общим «горизонтальным» проводом, принято называть *строкой* (row). ЗЭ, подключенные к общему «вертикальному» проводу, называют *столбцом* (column). Кроме того, к каждому ЗЭ необходимо подключить линию, по которой будет передаваться считанная и записываемая информация.

Совокупность запоминающих элементов и логических схем, связанных с выбором строк и столбцов, называют *ядром* микросхемы памяти. Помимо ядра, в ИМС имеется еще интерфейсная логика, обеспечивающая взаимодействие ядра с внешним миром. В ее задачи, в частности, входят коммутация нужного столбца на выход при считывании и на вход — при записи.

Для уменьшения числа контактов ИМС адреса строки и столбца в большинстве микросхем подаются в микросхему через одни и те же контакты последовательно во времени (мультиплексируются) и запоминаются соответственно в регистре адреса строки и регистре адреса столбца микросхемы. Мультиплексирование обычно реализуется внешней по отношению к ИМС схемой.

Для синхронизации обработки адресной информации внутри ИМС адрес строки сопровождается сигналом RAS (Row Address Strobe — строб строки), а адрес

столбца — сигналом CAS (Column Address Strobe — строб столбца). Вторую букву в аббревиатурах RAS и CAS иногда расшифровывают как Access — «доступ», то есть имеется строб доступа к строке и строб доступа к столбцу. Чтобы стробирование было надежным, эти сигналы подаются с задержкой, достаточной для завершения переходных процессов на шине адреса и в адресных цепях микросхемы.

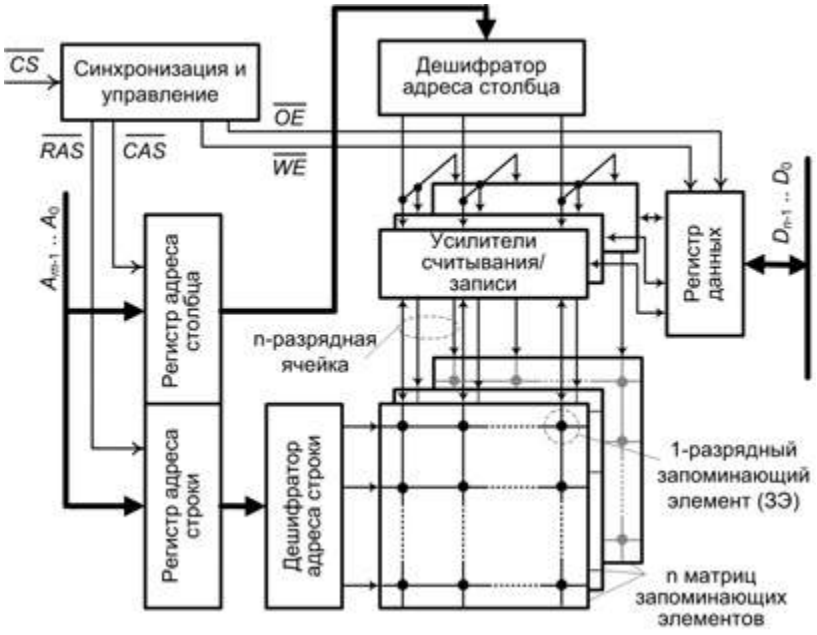


Рис. 6.6. Структура микросхемы памяти

Сигнал выбора микросхемы CS (Chip Select) разрешает работу ИМС и используется для выбора определенной микросхемы в системах, состоящих из нескольких ИМС. Вход WE (Write Enable — разрешение записи) определяет вид выполняемой операции (считывание или запись).

Записываемая информация, поступающая по шине данных, первоначально заносится во входной регистр данных, а затем — в выбранную ячейку. При выполнении операции чтения информация из ячейки до ее выдачи на шину данных буферизируется в выходном регистре данных. Обычно роль входного и выходного выполняет один и тот же регистр. Усилители считывания/записи (УСЗ) служат для электрического согласования сигналов на линиях данных и внутренних сигналов ИМС. Обычно число УСЗ равно числу запоминающих элементов в строке матрицы, и все они при обращении к памяти подключаются к выбранной горизонтальной линии. Каждая группа УСЗ, образующая ячейку, подключена к одному из столбцов матрицы, то есть выбор нужной ячейки в строке обеспечивается активизацией одной из вертикальных линий. На все время, пока ИМС памяти не использует шину данных, информационные выходы микросхемы переводятся в третье (высокоимпедансное) состояние, что эквивалентно отключению микросхемы от шины

данных. Управление переключением в третье состояние обеспечивается сигналом OE (Output Enable — разрешение выдачи выходных сигналов). Этот сигнал активизируется при выполнении операции чтения.

Для большинства перечисленных выше управляющих сигналов активным обычно считается их низкий уровень, что и показано на рис. 6.6.

Управление операциями с основной памятью осуществляется контроллером памяти, который может входить в состав центрального процессора либо реализуется в виде внешнего по отношению к памяти устройства. В последних типах ИМС памяти часть функций контроллера возлагается на микросхему памяти. Хотя работа ИМС памяти может быть организована как по синхронной, так и по асинхронной схеме, контроллер памяти — устройство синхронное, то есть срабатывающее исключительно по тактовым импульсам. По этой причине операции с памятью принято описывать с привязкой к тактам. В общем случае на каждую такую операцию требуется, как минимум, пять тактов, которые используются следующим образом:

1. Указание типа операции (чтение или запись) и установка адреса строки.
2. Формирование сигнала RAS.
3. Установка адреса столбца.
4. Формирование сигнала CAS.
5. Возврат сигналов RAS и CAS в неактивное состояние.

Данный перечень учитывает далеко не все необходимые действия, например регенерацию содержимого памяти в динамических ОЗУ.

«Классическую» процедуру доступа к памяти рассмотрим на примере чтения из ИМС с мультиплексированием адресов строк и столбцов (рис. 6.7). Сначала на входе WE устанавливается уровень, соответствующий операции чтения, а на адресные контакты ИМС подается адрес строки, сопровождаемый сигналом RAS. По заднему фронту этого сигнала адрес запоминается в регистре адреса строки микросхемы, после чего дешифрируется. После стабилизации процессов, вызванных сигналом RAS, выбранная строка подключается к УСЗ. Далее на вход ИМС подается адрес столбца, который по заднему фронту сигнала CAS заносится в регистр адреса столбца. Одновременно подготавливается выходной регистр данных, куда после стабилизации сигнала CAS загружается информация с выбранных УСЗ.

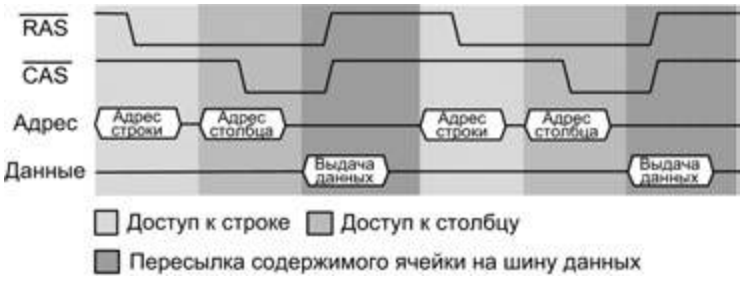


Рис. 6.7. Временная диаграмма «классической» процедуры чтения

Разработчики микросхем памяти тратят значительные усилия на повышение быстродействия ИМС, которое принято характеризовать четырьмя параметрами:

- t_{RAS} — минимальное время от перепада сигнала RAS (с высокого уровня к низкому) до момента появления и стабилизации считанных данных на выходе ИМС. Среди приводившихся в начале главы характеристик быстродействия это соответствует *времени выборки данных*;
- t_{RC} — минимальное время от начала доступа к одной строке микросхемы памяти до начала доступа к следующей строке. Этот параметр также упоминался в начале главы как *цикл обращения к ЗУ* $T_Ц$;
- t_{CAS} — минимальное время от перепада сигнала CAS с высокого уровня к низкому до момента появления и стабилизации считанных данных на выходе ИМС;
- T_{PC} — минимальное время от начала доступа к одному столбцу микросхемы памяти до начала доступа к следующему столбцу.

Возможности «ускорения» ядра микросхемы ЗУ весьма ограничены и связаны в основном с миниатюризацией запоминающих элементов. Наибольшие успехи достигнуты в интерфейсной части ИМС, касаются они, главным образом, операции чтения, то есть способов доставки содержимого ячейки на шину данных. Наибольшее распространение получили следующие фундаментальные подходы:

- последовательный;
- регистровый;
- быстрый постраничный;
- пакетный;
- конвейерный;
- удвоенной скорости.

Последовательный режим

Последовательный режим (Flow through Mode) представляет собой вариант реализации классического доступа к памяти в синхронных микросхемах. Адрес и управляющие сигналы подаются на микросхему до поступления синхроимпульса. В момент прихода синхроимпульса вся входная информация запоминается во внутренних регистрах — по его переднему фронту, и начинается цикл чтения. Через некоторое время, но в пределах того же цикла данные появляются на внешней шине, причем момент этот определяется только моментом прихода синхронизирующего импульса и скоростью внутренних цепей микросхемы.

Регистровый режим

Регистровый режим (Register to Latch) используется относительно редко и отличается от последовательного режима наличием регистра на выходе микросхемы. Считанные данные заносятся в промежуточный выходной регистр и хранятся там до появления отрицательного фронта (спада) синхроимпульса и в этот момент передаются на шину. Изменяя ширину импульса синхронизации, можно менять время появления данных на шине, что оказывается весьма полезным при проектировании специализированных ВМ. По быстродействию микросхемы с регистровым режимом идентичны ИМС с последовательным режимом.

Быстрый постраничный режим

Под страницей понимается строка матрицы ЗЭ. Последовательность элементов страницы соответствует ячейкам памяти с последовательно возрастающими адресами. Это позволяет при доступе к ячейкам со смежными адресами (согласно принципу локальности такая ситуация наиболее вероятна) существенно ускорить доступ ко второй и последующим ячейкам. Для доступа к очередной ячейке достаточно подавать на ИМС лишь адрес нового столбца, сопровождая его сигналом CAS. Отметим, что обращение к первой ячейке в последовательности производится стандартным образом — поочередным заданием адреса строки и адреса столбца, то есть здесь время доступа уменьшить практически невозможно. Рассмотренный режим называется *режимом постраничного доступа* или просто *постраничным режимом* (Page Mode). В интегральных микросхемах DRAM распространена модификация режима, известная как *быстрый постраничный режим* (FPM — Fast Page Mode). Особенность модификации — в способе занесения новой информации в регистр адреса столбца. Полный адрес (строки и столбца) передается только при первом обращении к строке. Активизация регистра адреса столбца производится не по сигналу CAS, а по заднему фронту сигнала RAS. Сигнал RAS остается активным на протяжении всего страничного цикла и позволяет заносить в регистр адреса столбца новую информацию не по спадающему фронту CAS, а как только адрес на входе ИМС стабилизируется, то есть практически по переднему фронту сигнала CAS. Временная диаграмма операции чтения для рассматриваемого режима показана на рис. 6.8.

Рис. 6.8. Временная диаграмма операции чтения в быстром постраничном режиме

В целом потери времени сокращаются на два такта, но реальный выигрыш наблюдается лишь при передаче блоков данных, хранящихся в одной и той же строке микросхемы. Если же программа часто обращается к разным областям памяти, переходя с одной строки ИМС на другую, преимущества метода теряются.

Микросхемы, где реализуется постраничный режим и его модификации, принято характеризовать формулой *x-y-y-y*. Первое число *x* представляет количество тактов системной шины, необходимое для доступа к первой ячейке последовательности, а числа *y* — количество тактов для доступа к каждой из последующих ячеек. Так, выражение 7-3-3-3 означает, что для обработки первого слова необходимо 7 тактовых периодов системной шины (в течение шести из которых шина простаивает

в ожидании), а для обработки последующих слов — по три периода, из которых два системная шина также простаивает.

Пакетный режим

Пакетный режим (Burst Mode) — режим, при котором на запрос по конкретному адресу память возвращает пакет данных, хранящихся не только по этому адресу, но и по нескольким последующим адресам.

Разрядность ячейки памяти современных ВМ обычно равна одному байту. Если ширина шины данных равна четырем байтам, то одно обращение к памяти требует последовательного доступа к четырем смежным ячейкам — пакету[1]. С учетом этого обстоятельства, в ИМС памяти часто используется модификация страничного режима, носящая название *группового* или *пакетного режима*. При его реализации адрес столбца заносится в ИМС только для первой ячейки пакета, а переход к очередному столбцу производится уже внутри микросхемы. Это позволяет для каждого пакета исключить три из четырех операций занесения в ИМС адреса столбца и тем самым еще более сократить среднее время доступа (рис. 6.9).

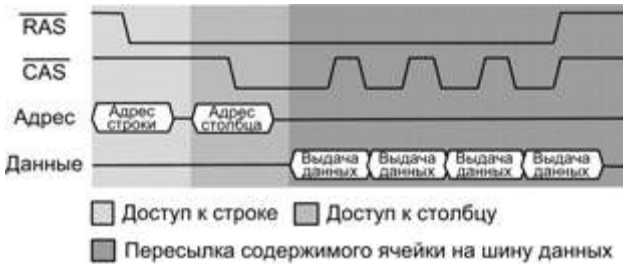


Рис. 6.9. Временная диаграмма операции чтения в пакетном режиме

Конвейерный режим

В *конвейерном режиме* (pipeline mode) процесс разбивается на два этапа. Пока данные из предыдущего цикла чтения передаются на внешнюю шину, происходит запрос на следующую операцию чтения. Таким образом, два цикла чтения перекрываются во времени. Из-за усложнения схемы передачи данных на внешнюю шину время считывания увеличивается на один такт, и данные поступают на выход только в следующем такте, но такое запаздывание наблюдается лишь при первом чтении в последовательности операций считывания из памяти. Все последующие данные поступают на выход друг за другом, хотя и с запаздыванием на один такт относительно запроса на чтение. Так как циклы чтения перекрываются, микросхемы с конвейерным режимом могут использоваться при частотах шины, вдвое превышающих допустимую для ИМС с последовательным режимом чтения. На рис. 6.10 показана временная диаграмма операции чтения, в которой конвейерный режим сочетается с пакетным.

[1] Строго говоря, количество ячеек, считываемое за один раз без дополнительного указания адреса и называемое длиной пакета (burst length), в большинстве случаев может программироваться. Помимо упомянутых четырех, это могут быть 1, 2 или 8 ячеек подряд.

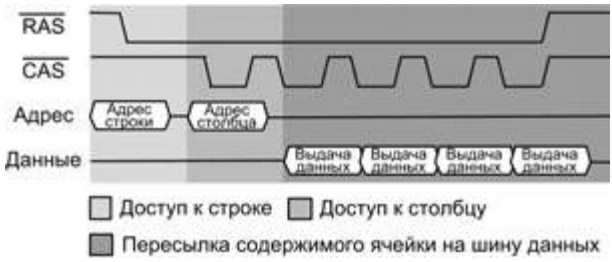


Рис. 6.10. Временная диаграмма операции чтения в пакетном режиме с конвейеризацией

Режим удвоенной скорости

Важным этапом в дальнейшем развитии технологии микросхем синхронной памяти стал режим DDR (Double Data Rate) — удвоенная скорость передачи данных. Сущность метода заключается в передаче данных по обоим фронтам импульса синхронизации, то есть дважды за период. Таким образом, пропускная способность увеличивается в те же два раза. Ввиду того, что данный режим в микросхемах динамической памяти в настоящее время является наиболее распространенным, далее он будет рассмотрен подробнее.

Помимо упомянутых, используются и другие приемы повышения быстродействия ИМС памяти, такие как включение в состав микросхемы вспомогательной кэш-памяти и независимые тракты данных, позволяющие одновременно производить обмен с шиной данных и обращение к матрице ЗЭ и т. д.

Оперативные запоминающие устройства

Большинство из применяемых в настоящее время типов микросхем оперативной памяти[1] не в состоянии сохранять данные без внешнего источника энергии, то есть являются энергозависимыми (volatile memory). Широкое распространение таких устройств связано с рядом их достоинств по сравнению с энергонезависимыми типами ОЗУ (non-volatile memory): большей емкостью, низким энергопотреблением, более высоким быстродействием и невысокой себестоимостью хранения единицы информации.

Энергозависимые ОЗУ можно подразделить на две основные подгруппы: динамическую память (DRAM — Dynamic Random Access Memory) и статическую память (SRAM — Static Random Access Memory).

Статические оперативные запоминающие устройства

В *статических ОЗУ* запоминающий элемент может хранить записанную информацию неограниченно долго (при наличии питающего напряжения). Роль запоминающего элемента в статическом ОЗУ исполняет триггер. Такой триггер представляет собой схему с двумя устойчивыми состояниями, обычно состоящую из шести транзисторов (рис. 6.11).

---

[1] Оперативной называют память, в которой размещаются данные, над которыми непосредственно производятся операции процессора.

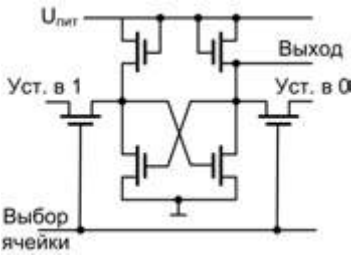


Рис. 6.11. Запоминающий элемент статического ОЗУ

Статические ОЗУ на настоящий момент — наиболее быстрый, правда, и наиболее дорогостоящий вид оперативной памяти. Известно достаточно много различных вариантов реализации SRAM, отличающихся по технологии и способам организации. Основная сфера применения статических ОЗУ — кэш-память. Несмотря на то что статические ОЗУ могут быть построены как по асинхронной, так и по синхронной схеме, в настоящее время предпочтение отдается синхронным ЗУ. В рамках группы *синхронных статических ОЗУ* выделяют ИМС типа SSRAM и более совершенные PB SRAM.

Как и в любой синхронной памяти, все события в SSRAM происходят с поступлением внешних тактовых импульсов. Отличительная особенность SSRAM — регистры, где фиксируется входная информация. Рассматриваемый вид памяти обеспечивает работу в пакетном режиме с формулой 3-1-1-1, но лишь до определенных значений тактовой частоты шины. При более высоких частотах формула изменяется на 3-2-2-2. В варианте с пакетным конвейерным доступом (PB SRAM — Pipelined Burst SRAM) реализована внутренняя конвейеризация, за счет которой скорость обмена пакетами данных возрастает примерно вдвое. Память данного типа хорошо работает при повышенных частотах системной шины. Время доступа к PB SRAM составляет от 4,5 до 8 нс, при этом формула 3-1-1-1 сохраняется даже при более высоких частотах системной шины.

Важным моментом, характеризующим SRAM, является технология записи. Известны два варианта записи: *стандартная* и *запаздывающая*. В стандартном режиме адрес и данные выставляются на соответствующие шины в одном и том же такте. В режиме запаздывающей записи данные для нее передаются в следующем такте после выбора адреса нужной ячейки, что напоминает режим конвейерного чтения, когда данные появляются на шине в следующем такте. Оба рассматриваемых варианта позволяют производить запись данных с частотой системной шины. Различия сказываются только при переключении между операциями чтения и записи.

В развитие идеи записи с запаздыванием компания IDT (Integrated Device Technology) предложила технологию, получившую название ZBT SRAM (Zero Bus Turnaround) — нулевое время переключения шины. Идея ее состоит в том, чтобы запись с запаздыванием производить с таким же интервалом, какой требуется для чтения. Так, если SRAM с конвейерным чтением требует три тактовых периода для чтения данных из ячейки, то данные для записи нужно передавать с таким же промедлением относительно адреса. В результате перекрывающиеся циклы чтения

и записи идут один за другим, позволяя выполнять операции чтения/записи в каждом такте без каких-либо задержек[1].

Динамические оперативные запоминающие устройства

Динамические ЗУ, как и статические, энергозависимы. Запоминающий элемент *динамического ОЗУ* способен хранить информацию только в течение достаточно короткого промежутка времени, после которого информацию нужно восстанавливать заново, иначе она будет потеряна. Такой ЗЭ состоит из одного конденсатора и запирающего транзистора (рис. 6.12).

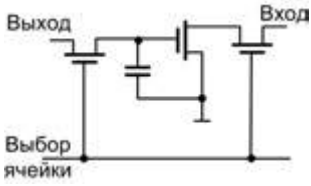


Рис. 6.12. Запоминающий элемент динамического ОЗУ

Наличие или отсутствие заряда в конденсаторе интерпретируется как 1 или 0 соответственно. Простота схемы позволяет достичь высокой плотности размещения ЗЭ и в итоге снизить стоимость. Плотность упаковки ЗЭ в микросхемах динамических ОЗУ по сравнению со статическими выше в 4–8 раз, а отношение стоимость/бит — ниже в 8–16 раз. С другой стороны, считать значение, хранящееся в ячейке DRAM, невозможно, не изменив эту информацию, поэтому после операции считывания заряд нужно восстановить. Такое восстановление производится путем занесения считанной информации в специальный буфер, с последующей перезаписью из буфера на то же место, благодаря чему происходит перезаряд конденсаторов. Кроме того, накапливаемый на конденсаторе заряд из-за паразитных утечек со временем теряется. Среднее время утечки заряда ЗЭ динамической памяти составляет сотни или даже десятки миллисекунд, поэтому заряд необходимо успеть восстановить в течение данного отрезка времени, иначе хранящаяся информация будет утеряна. Периодическое восстановление заряда путем перезаписи информации называется *регенерацией* и осуществляется каждые 2–8 мс. Операции разрядки-перезарядки могут занимать до 5% от общего времени работы с памятью, снижая скорость работы DRAM. В среднем быстродействие DRAM в 8–16 раз ниже, чем у статической памяти.

В различных типах ИМС динамической памяти нашли применение три основных метода регенерации:

- одним сигналом RAS (ROR — RAS Only Refresh);
- сигналом CAS, предваряющим сигнал RAS (CBR — CAS Before RAS);
- автоматическая регенерация (SR — Self Refresh).

[1] Сходную с ZBT SRAM технологию предложила также фирма Cypress Semiconductor. Эта технология получила название NoBL SRAM (No Bus Latency — дословно «нет задержек шины»).

Регенерация одним RAS использовалась еще в первых микросхемах DRAM. На шину адреса выдается адрес регенерируемой строки, сопровождаемый сигналом RAS. При этом выбирается строка ячеек, и хранящиеся там данные поступают на внутренние цепи микросхемы, после чего записываются обратно. Так как сигнал CAS не появляется, цикл чтения/записи не начинается. В следующий раз на шину адреса подается адрес следующей строки и т. д., пока не восстановятся все ячейки, после чего цикл повторяется. К недостаткам метода можно отнести занятость шины адреса в момент регенерации, из-за чего доступ к другим устройствам ВМ блокирован.

Особенность метода CBR в том, что если в обычном цикле чтения/записи сигнал RAS всегда предшествует сигналу CAS, то при появлении первым сигнала CAS начинается специальный цикл регенерации. В этом случае адрес строки не передается, а микросхема использует свой внутренний счетчик, содержимое которого увеличивается на единицу при каждом очередном CBR-цикле. Режим позволяет регенерировать память, не занимая шину адреса, то есть более эффективен.

Автоматическая регенерация памяти связана с энергосбережением, когда система переходит в режим «сна» и тактовый генератор перестает работать. При отсутствии внешних сигналов RAS и CAS обновление содержимого памяти методами ROR или CBR невозможно, и микросхема производит регенерацию самостоятельно, запуская собственный генератор, который тактирует внутренние цепи регенерации.

Область применения статической и динамической памяти определяется скоростью и стоимостью. Как уже отмечалось, главным преимуществом SRAM является более высокое быстродействие, однако из-за малой емкости микросхем и высокой стоимости применение статической памяти, как правило, ограничено относительно небольшой по емкости кэш-памятью.

Динамической памяти в вычислительной машине значительно больше, чем статической, поскольку именно DRAM используется в качестве основной памяти ВМ. Как и SRAM, динамическая память состоит из ядра (массива ЗЭ) и интерфейсной логики (буферных регистров, усилителей чтения данных, схемы регенерации и др.). Хотя количество видов DRAM уже превысило два десятка, ядро у них организовано практически одинаково. Главные различия связаны с интерфейсной логикой, причем различия эти обусловлены также и областью применения микросхем — помимо основной памяти ВМ, ИМС динамической памяти входят, например, в состав видеоадаптеров. Чтобы оценить различия между видами DRAM, предварительно остановимся на «классическом» алгоритме работы с динамической памятью. Для этого воспользуемся рис. 6.6.

Адрес ячейки DRAM передается в микросхему за два шага — вначале адрес столбца, а затем строки. Для указания, какая именно часть адреса передается в определенный момент, служат два вспомогательных сигнала RAS и CAS. При обращении к ячейке памяти на шину адреса выставляется адрес строки. После стабилизации процессов на шине подается сигнал RAS, и адрес записывается во внутренний регистр микросхемы памяти. Затем на шину адреса выставляется адрес столбца и выдается сигнал CAS. В зависимости от состояния линии WE производится чтение данных из ячейки или их запись в ячейку (перед записью данные должны быть

помещены на шину данных). Интервал между установкой адреса и выдачей сигнала RAS (или CAS) оговаривается техническими характеристиками микросхемы, но обычно адрес выставляется в одном такте системной шины, а управляющий сигнал — в следующем. Таким образом, для чтения или записи одной ячейки динамического ОЗУ требуется пять тактов, в которых происходит соответственно: выдача адреса строки, выдача сигнала RAS, выдача адреса столбца, выдача сигнала CAS, выполнение операции чтения/записи (в статической памяти процедура занимает лишь от двух до трех тактов).

Следует также помнить о необходимости регенерации данных. Но наряду с естественным разрядом конденсатора ЗЭ со временем к потере заряда приводит также считывание данных из DRAM, поэтому после каждой операции чтения данные должны быть восстановлены. Это достигается за счет повторной записи тех же данных сразу после чтения. При считывании информации из одной ячейки фактически выдаются данные сразу всей выбранной строки, но используются только те, которые находятся в интересующем столбце, а все остальные игнорируются. Таким образом, операция чтения из одной ячейки приводит к разрушению данных всей строки и их нужно восстанавливать. Регенерация данных после чтения выполняется автоматически интерфейсной логикой микросхемы, и происходит это сразу же после считывания строки.

Асинхронные динамические ОЗУ. Работа таких микросхем не привязана жестко к тактовым импульсам. В течение достаточно длительного периода времени основная память ВМ строилась на базе микросхем ОЗУ асинхронного типа. На этом этапе микросхемы постоянно совершенствовались, главным образом, за счет перехода к более эффективным режимам чтения (рис. 6.13). На смену «классическим» DRAM с последовательным режимом чтения пришли микросхемы с быстрым постраничным режимом (FPM DRAM). Достаточно быстро их сменили микросхемы типа EDO DRAM с усовершенствованным страничным (гиперстраничным) режимом. Линия асинхронных динамических ОЗУ фактически завершилась микросхемами типа BEDO DRAM, в которых сочетались пакетный и конвейерный режим чтения. Несмотря на достаточно высокие скоростные характеристики BEDO (5-1-1-1) и этим микросхемам свойственен общий недостаток асинхронной схемы — дополнительные затраты времени на взаимодействие микросхем памяти и контроллера. Это обстоятельство, а также успехи в технологии синхронных DRAM (главным образом — DDR) обусловили практически повсеместный переход к построению основной памяти на базе синхронных динамических ОЗУ.

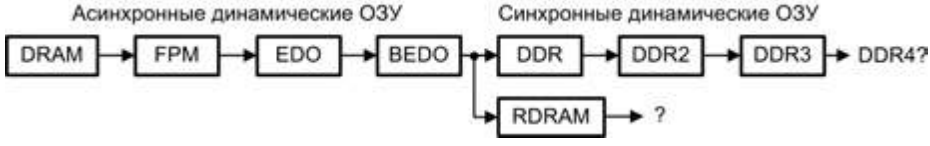


Рис. 6.13. Эволюция микросхем динамической оперативной памяти

Синхронные динамические ОЗУ. В синхронных DRAM обмен информацией синхронизируется внешними тактовыми сигналами и происходит в строго определенные

моменты времени, что позволяет взять все от пропускной способности шины «процессор-память» и избежать циклов ожидания. Адресная и управляющая информация фиксируется в микросхеме памяти, после чего ответная реакция микросхемы произойдет через четко определенное число тактовых импульсов, и это время процессор может использовать для других действий, не связанных с обращением к памяти. В случае синхронной динамической памяти вместо продолжительности цикла доступа говорят о минимально допустимом периоде тактовой частоты, и речь уже идет о времени порядка единиц наносекунд.

Для обозначения микросхем «обычных» синхронных динамических ОЗУ используется аббревиатура SDRAM (Synchronous DRAM — синхронная DRAM). В настоящее время наиболее распространенным типом динамической памяти персональных ВМ являются микросхемы, реализующие технологию DDR SDRAM (Double Data Rate SDRAM — SDRAM с удвоенной скоростью передачи данных). Основная особенность технологии в том, что передача данных синхронизируется как передним, так и задним фронтом тактового импульса. Таким образом, при сохранении тактовой частоты шины памяти запрошенные данные передаются вдвое быстрее (рис. 6.14) — по два пакета за один тактовый период (память работает в пакетном режиме). Как следствие, получается двукратное увеличение пропускной способности шины памяти.

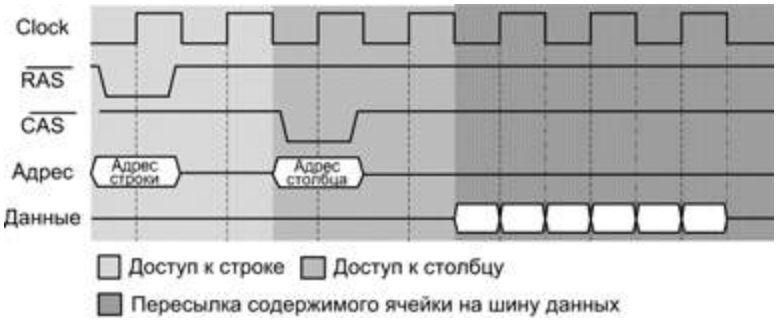


Рис. 6.14. Иллюстрация идеи синхронизации по обоим фронтам тактового импульса

С момента появления технологии сменилось несколько поколений DDR (DDR, DDR2, DDR3). С каждым следующим поколением увеличивается рабочая частота микросхем и снижается потребляемая мощность за счет снижения рабочего напряжения.

Обозначение микросхем типа DDR имеет вид DDR*x*-*yyyy*, где *x* — поколение DDR, а *yyyy* — эффективная частота DDR, которая вдвое больше реальной максимальной тактовой частоты. Например, микросхемы DDR2-800 могут работать на частоте шины памяти 400 МГц. Следует иметь в виду, что указанное число отражает лишь максимальное значение частоты, на которой может работать микросхема, но «автоматической» подстройки частоты не происходит. Тактовая частота шины памяти задается контроллером памяти, внешним по отношению к микросхеме памяти.

В случае персональных ВМ, где микросхемы памяти объединяются в модули, используется несколько иная система обозначений: PC*x*-*zzzz*. Здесь по-прежнему *x* — поколение DDR, а *zzzz* — максимальная теоретическая скорость передачи (полоса пропускания). Полоса пропускания определяет количество байтов данных, которое может быть передано в секунду между контроллером памяти и модулем памяти, в предположении, что данные будут передаваться с каждым тактовым импульсом. Для расчета максимальной скорости передачи используется следующая формула: *тактовая частота шины памяти* × 2 (передача данных дважды за такт) × 8 (число байтов, передающихся за такт при ширине шины памяти 64 бита). Например, модуль памяти на микросхемах DDR2-800 обладает максимальной теоретической скоростью передачи 6400 Мбайт/с (мегабайт в секунду), и такой модуль обозначается как PC2-6400. В ряде случаев число округляется. Например, модули на базе DDR3-1333, имеющие максимальную теоретическую скорость передачи 10666 МБ/с, в зависимости от производителя, могут обозначаться как PC3-10666 либо как PC3-10600.

Микросхемы памяти DDR, DDR2 и DDR3 классифицируют не только по скорости, с которой они могут работать. Помимо скорости используется иная информация, говорящая о временных соотношениях, присущих микросхеме. Внешне — это выражения вида 2-2-2-6-T1 (не нужно их путать с ранее приводившейся и внешне похожей формулой для микросхем, реализующих постраничный режим). Именно эти показатели объясняют, почему два модуля памяти с одинаковой теоретической максимальной скоростью пересылки данных обеспечивают разные уровни производительности.

Числа, входящие в подобные выражения, означают количество тактовых периодов, требуемых для выполнения микросхемой определенной операции. Чем меньше число, тем быстрее память.

Приводимые в обозначении числа определяют лишь теоретический максимум, который никогда не может быть достигнут, поскольку предполагается, что микросхема в каждом тактовом периоде пересылает в контроллер памяти исключительно данные. Однако некоторые периоды не могут быть использованы для передачи данных, поскольку в эти периоды контроллер и память обмениваются служебной информацией, обеспечивающей их взаимодействие.

В упрощенном варианте структуру DDR SDRAM можно представить в виде, показанном на рис. 6.15.

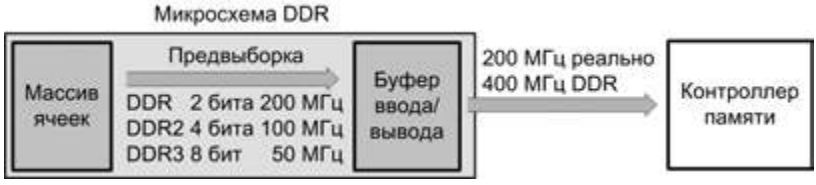


Рис. 6.15. Структура DDR SDRAM с иллюстрацией идеи *n*-битовой предвыборки

Чтобы обеспечить передачу данных дважды за такт, используется специальная архитектура с предвыборкой, в которой присутствует буфер ввода/вывода (буфер

предвыборки). В каждом тактовом периоде шины из массива ячеек в этот буфер передаются два бита данных. Это называется 2*n*-битовой предвыборкой. В DDR2 внутренняя шина между массивом ячеек и буфером увеличена до 4 битов, а в DDR3 — до 8 битов.

Для лучшего понимания этой идеи сравним микросхемы DDR-400, DDR2-400 и условную DDR3-400 (реально такая микросхема не существует). Формально эти микросхемы работают на частоте шины памяти 200 МГц, передавая в каждом тактовом периоде два пакета данных, что внешне эквивалентно работе на тактовой частоте 400 МГц. В то же время внутри микросхемы DDR за тактовый период между массивом ячеек и буфером ввода/вывода за один тактовый период передаются 2 бита, поэтому соответствующий тракт должен работать на частоте 200 МГц (200 МГц × 2 = = 400 МГц). Так как в DDR2 ширина тракта увеличена до 4 битов, то для получения той же производительности этот тракт может работать на вдвое меньшей частоте (100 МГц × 4 = 400МГц), а в DDR3 — частота тракта может быть уменьшена еще вдвое (50 МГц × 8 = 400 МГц).

Удвоение ширины внутреннего тракта означает, что каждое новое поколение предположительно может включать микросхемы с удвоенной максимальной тактовой частотой, по сравнению с достигнутой в предыдущем поколении. Например, микросхемы DDR-400, DDR2-800 и DDR3-1600 внутренне работают на той же тактовой частоте 200 МГц.

В табл. 6.1 приведены основные характеристики поколений DDR.

Таблица 6.1. Характеристики микросхем DDR

	DDR	**DDR2**	**DDR3**
Буфер предвыборки, бит	2	4	8
Уровень сигналов, В	2,5	1,8	1,5
Частота шины	200, 266, 333, 400	400, 533, 677, 800	800, 1066, 1330, 1600

На момент написания учебника происходит активный процесс перехода от DDR2 к DDR3. Этому способствует поддержка со стороны ведущих производителей ВМ. Так, поддержка DDR3 заложена в ряде новых процессоров компании Intel. Например, процессоры Nehalem содержат интегрированный контроллер памяти DDR3. Очередным этапом в эволюции технологии DDR должна стать DDR4, появление которой ожидается в 2012 году. «Дорожная карта» развития технологии показана на рис. 6.16.

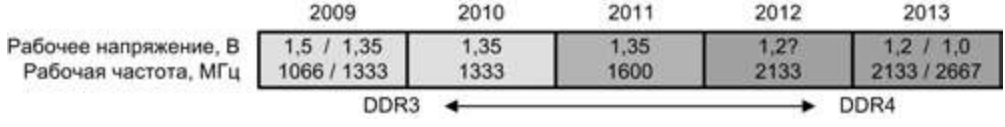


Рис. 6.16. «Дорожная карта» развития технологии DDR

Микросхемы RDRAM, DRDRAM. Наиболее очевидные способы повышения эффективности работы процессора с памятью — увеличение тактовой частоты шины либо

ширины выборки (количества одновременно пересылаемых разрядов). К сожалению, попытки совмещения обоих вариантов наталкиваются на существенные технические трудности (с повышением частоты усугубляются проблемы электромагнитной совместимости, труднее становится обеспечить одновременность поступления потребителю всех параллельно пересылаемых битов информации). В большинстве синхронных DRAM (SDRAM, DDR) применяется широкая выборка (64 бита) при ограниченной частоте шины.

Помимо микросхем с идеологией DDR следует упомянуть еще одну ветвь в эволюции синхронных динамических ОЗУ. Речь идет о принципиально отличном подходе к построению SDRAM, предложенном компанией Rambus в 1997 году. В нем упор сделан на повышение тактовой частоты до 400 МГц при одновременном уменьшении ширины выборки до 16 битов. Новая память известна как RDRAM (Rambus Direct RAM). Главным отличием от других типов SDRAM является оригинальная система обмена данными между ядром и контроллером памяти, в основе которой лежит так называемый «канал Rambus», применяющий асинхронный блочно-ориентированный протокол. На логическом уровне информация между контроллером и памятью передается пакетами. Различают три вида пакетов: пакеты данных, пакеты строк и пакеты столбцов. Пакеты строк и столбцов служат для передачи от контроллера памяти команд управления соответственно линиями строк и столбцов массива запоминающих элементов. Эти команды заменяют обычную систему управления микросхемой с помощью сигналов RAS, CAS, WE и CS. Несмотря на перспективность описанного подхода, в силу ряда обстоятельств, не имеющих отношения к технической стороне вопроса, большинство наиболее известных фирм отказались от использования RDRAM в своих разработках.

Многоканальная динамическая память

В условиях непрерывно возрастающей скорости и производительности процессоров память все в большей степени становится «узким местом» ВМ. Это происходит из-за того, что процессор работает быстрее основной памяти и вынужден ожидать поступления данных из ОП. Многоканальная архитектура — это технология, теоретически позволяющая увеличить пропускную способность тракта передачи данных между памятью и *контроллером памяти*. Контроллером памяти называется узел, управляющий обменом данными между основной памятью и центральным процессором. Контроллер физически может быть частью процессора либо частью микросхем чипсета — комплекса микросхем на материнской плате ВМ, обеспечивающих взаимодействие всех частей ВМ как единой системы. В современных ВМ данные между основной памятью и контроллером памяти передаются по 64-разрядной шине. Многоканальность — это способность некоторых контроллеров памяти увеличить ширину шины данных пропорционально числу каналов (дальнейшее изложение будет вестись применительно к двухканальному варианту). Учитывая, что тактовая частота шины остается неизменной, максимальная теоретическая скорость передачи данных удваивается.

Двухканальная архитектура требует использования двухканального контроллера (или двух независимых одноканальных) и наличия двух или более модулей памяти, построенных на микросхемах типа DDR, DDR2, либо DDR3. Модули памяти

устанавливаются в разъемы материнской платы, каждый из которых обозначен своим цветом (цвет указывает: какой из банков памяти считается банком 0 или банком 1). Отдельные каналы позволяют каждому модулю независимо обращаться к контроллеру, тем самым увеличивается полоса пропускания. Модули не обязательно должны быть идентичными. При различных модулях общая скорость памяти будет определяться скоростью более медленного модуля.

Вопрос эффективности двухканальной архитектуры памяти был предметом ряда исследований. Согласно одних результатов различие между одноканальной и двухканальной организацией не превысило 5%. По другим данным применение двухканальной технологии приводит к выигрышу в скорости порядка 15,3%. Такой разброс объясняют разницей в способе реализации двухканальной идеи в различных материнских платах.

В последних разработках компании Intel (процессоры Core i7) и AMD (Sao Paulo) заложено использование трехканальной памяти на микросхемах типа DDR3. Идея такой памяти аналогична двухканальной, но число каналов между ОП и контроллером памяти увеличено до трех, с соответствующим увеличением теоретической пропускной способности. Перспективные разработки AMD ориентированы на четырехканальную память на микросхемах DDR3.

В зависимости от производителя и типа модули трехканальной памяти имеют выражения 9-9-9-27 или 9-9-9-24 (Kingston), 8-8-8-24 (A-Data).

Оперативные запоминающие устройства для видеоадаптеров

Использование памяти в видеоадаптерах имеет свою специфику, и для реализации дополнительных требований прибегают к несколько иным типам микросхем. Так, при создании динамичных изображений часто достаточно просто изменить расположение уже хранящейся в видеопамяти информации. Вместо того чтобы многократно пересылать по шине одни и те же данные, лишь несколько изменив их расположение, выгоднее заставить микросхему памяти переместить уже хранящиеся в ней данные из одной области ядра в другую. На ИМС памяти можно также возложить операции по изменению цвета точек изображения.

Краткое описание некоторых из типов ОЗУ, ориентированных на применение в качестве видеопамяти, можно найти в [35]. К настоящему моменту они уже утратили актуальность. Из применяемых в современных графических картах ведущих производителей свои позиции сохранили микросхемы типа VRAM. Одновременно с этим широкое распространение получают микросхемы динамической памяти, в основе которых лежит идеология DDR: GDDR2, GDDR3, GDDR4, GDDR5.

Микросхемы VRAM. ОЗУ типа VRAM (Video RAM) отличается высокой производительностью и предназначено для мощных графических систем. При разработке ставилась задача обеспечить постоянный поток данных при обновлении изображения на экране. Для типовых значений разрешения и частоты обновления изображения интенсивность потока данных приближается к 200 Мбит/с. В таких условиях процессору трудно получить доступ к видеопамяти для чтения или записи. Чтобы разрешить эту проблему, в микросхеме сделаны существенные архитектурные изменения, позволяющие обособить обмен между процессором и ядром VRAM (для

чтения/записи информации) и операции по выдаче информации на схему формирования видеосигнала.

GDDR (Graphics Double Data Rate) — семейство микросхем видеопамяти с идеологией DDR. Эволюцию семейства отражает ряд: GDDR2, GDDR3, GDDR4, GDDR5. От обычных микросхем типа DDR это семейство отличают некоторые особенности, обусловленные спецификой применения. Так, в основе GDDR3 лежит та же технологическая база, что и в DDR2, однако в видеоварианте снижены требования к потребляемой мощности и рассеиваемому теплу. Это и ряд других усовершенствований привели к повышению производительности модулей памяти и упрощению системы охлаждения. В свою очередь, в GDDR4 реализована 8-битовая предвыборка взамен 4-битовой в GDDR3. GDDR5 отличается от своего предшественника (GDDR4) тем, что в микросхеме реализована технология DDR3 (а не DDR2). Это изменение позволило использовать графические карты с GDDR5 в приложениях, особо критичных к полосе пропускания.

Специальные типы оперативной памяти

В ряде практических задач более выгодным оказывается использование специализированных архитектур ОЗУ, где стандартные функции (запись, хранение, считывание) сочетаются с некоторыми дополнительными возможностями или учитывают особенности применения памяти. Такие виды ОЗУ называют специализированными и к ним причисляют:

- память с множественным доступом (многопортовые ОЗУ);
- память типа очереди (ОЗУ типа FIFO).

Оба типа относятся к статическим ОЗУ.

Многопортовые ОЗУ

Стандартное однопортовое ОЗУ имеет по одной шине адреса, данных и управления и в каждый момент времени обеспечивает доступ к ячейке памяти только одному устройству. Структура запоминающего элемента (ЗЭ) такого ОЗУ приведена на рис. 6.17, *а*.

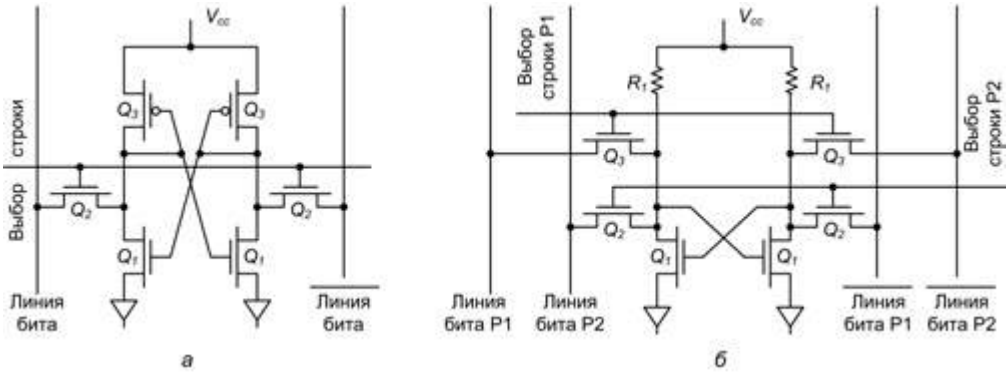


Рис. 6.17. Запоминающие элементы статического ОЗУ: *а* — однопортового; *б* — двухпортового

В отличие от стандартного в *n*-портовом ОЗУ имеется *n* независимых наборов шин адреса, данных и управления, гарантирующих одновременный и независимый доступ к ОЗУ *n*-устройствам. Данное свойство позволяет существенно упростить создание многопроцессорных и многомашинных вычислительных систем, где многопортовое ОЗУ выступает в роли общей или совместно используемой памяти. В настоящее время серийно выпускаются двух- и четырехпортовые микросхемы, среди которых наиболее распространены первые. Поскольку архитектурные решения в обоих случаях схожи, дальнейшее изложение будет вестись применительно к двухпортовым ОЗУ.

ЗЭ двухпортового ОЗУ (рис. 6.17, *б*) также содержит шесть транзисторов, но, в отличие от стандартного запоминающего элемента (ЗЭ), транзисторы Q_3 служат не в качестве резисторов, а предоставляют доступ к элементу с двух направлений.

В двухпортовой памяти имеются два набора адресных, информационных и управляющих сигнальных шин, каждый из которых обеспечивает доступ к общему массиву ЗЭ (рис. 6.18). Поскольку двухпортовому ОЗУ свойственна симметричная структура, в дальнейшем наборы шин будем называть «левым» (Л) и «правым» (П). В целом организация матрицы ЗЭ остается традиционной.

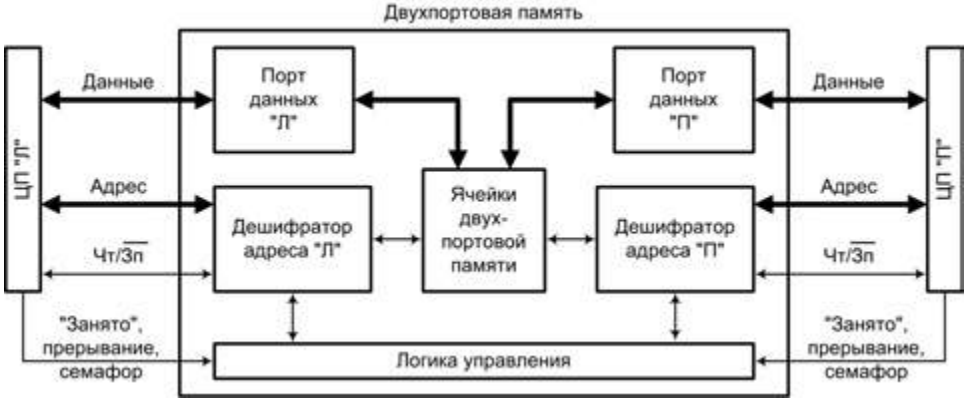


Рис. 6.18. Структура двухпортового ОЗУ

Доступ к ячейкам возможен как через левую, так и через правую группу шин, причем если Л- и П-адреса различны, никаких конфликтов не возникает. Проблемы потенциально возможны, когда Л- и П-устройства одновременно обращаются по одному и тому же адресу и хотя бы одно из этих устройств пытается выполнить операцию записи. В этом случае, если один из портов читает информацию, а другой производит запись в ту же ячейку, вероятно считывание недостоверной информации. При попытке единовременной записи в ячейку с двух направлений в нее может быть занесена неопределенная комбинация из записываемых слов. Несмотря на то что вероятность подобных ситуаций по оценкам не превышает 0,1%, такой вариант необходимо учитывать, для чего в двухпортовой памяти имеется схема арбитража с использованием сигналов «Занято». Логика арбитража в микросхеме реализована аппаратными средствами.

Помимо возможности доступа к ячейкам с двух направлений, двухпортовая память снабжается средствами для обмена сообщениями между подключенными к ней устройствами: системой прерывания и системой семафоров. Первую из них называют аппаратной, а вторую — программной.

В системе прерываний двухпортовой памяти две последних ячейки микросхемы (с наибольшими адресами) используются в качестве «почтовых ящиков» для обмена сообщениями между устройствами, подключенными к Л- и П-портам. Сообщению от левого устройства выделена ячейка с четным адресом, а от правого — с нечетным. Когда устройство записывает информацию в свой «почтовый ящик», формируется запрос прерывания к устройству, подключенному к противоположному порту. Этот сигнал автоматически сбрасывается, когда адресат считывает информацию из своего «почтового ящика».

Система семафоров — это имеющийся в двухпортовой памяти набор из восьми триггеров, состояние которых может быть прочитано и изменено со стороны любого из портов. Триггеры играют роль программных семафоров или флагов, с помощью которых Л- и П-устройства могут извещать друг друга о каких-то событиях. Сущность этих событий не зафиксирована и определяется реализуемыми программами. Обычно семафоры нужны для предоставления одному из процессоров монопольного права работы с определенным блоком данных (до завершения всех необходимых операций с этим блоком). В этом случае процессор, монополизирующий блок данных, устанавливает один из семафоров в состояние 1, а по завершении — в 0. Второй процессор, прежде чем обратиться к данному блоку, считывает семафор и при единичном состоянии последнего повторяет считывание и анализ семафора до тех пор, пока первый процессор не установит его в состояние 0. Естественно, что в программном обеспечении Л- и П-процессоров распределение и правила использования семафоров должны быть согласованы.

Зачастую одной микросхемы многопортовой памяти не хватает из-за недостаточной емкости одной ИМС или ввиду малой разрядности ячеек. В обоих случаях необходимо соединить несколько микросхем, соответственно параллельно или последовательно. Если несколько микросхем объединяются в цепочку для достижения нужной разрядности слова, возникает проблема с арбитражем при одновременном обращении к одной и той же ячейке. В этих случаях в разных ИМС цепочки, в силу разброса их параметров, предпочтение может быть отдано разным портам, в то время как решение должно быть единым. Для исключения подобной ситуации микросхемы многопортовой памяти выпускаются в двух вариантах: ведущие (master) и ведомые (slave). Принятие решения производится только в ведущих микросхемах, а ведомые функционируют в соответствии с инструкцией, полученной от ведущего. Таким образом, в цепочке используется только одна микросхема типа «ведущий», а все прочие ИМС должны иметь тип «ведомый».

Память типа FIFO

Во многих случаях ОЗУ применяется для буферизации потока данных, когда данные считываются из памяти в той же последовательности, в которой они туда заносились, но поступление и считывание происходят с различной скоростью. Часто для этой цели применяют обычное ОЗУ, однако здесь одновременная запись

и считывание информации невозможны. Более эффективным видом ОЗУ, где оба действия могут вестись одновременно, служит память типа FIFO. Микросхема представляет собой двухпортовое ОЗУ, где один порт предназначен для занесения информации, а второй — для считывания. Для FIFO-памяти характерны все технологические приемы, свойственные двухпортовой памяти, в частности способы арбитража при одномоментном обращении к одной и той же ячейке. В то же время есть и существенные отличия.

Первое состоит в том, что у микросхемы нет входов для указания адреса ячейки, занесение и считывание данных производится в порядке их поступления через одну входную точку и одну выходную.

Второе отличие связано с необходимостью слежения за состоянием очереди. Для этого в микросхеме имеются регистры-указатели адресов начала и конца очереди, а также специальные флаги, которые указывают на две ситуации: отсутствие данных (в этом случае блокируется считывание из микросхемы) и полное заполнение памяти (блокируется запись).

Постоянные запоминающие устройства

Слово «постоянные» в названии этого вида запоминающих устройств относится к их свойству хранить информацию при отсутствии питающего напряжения. Микросхемы ПЗУ также построены по принципу матричной структуры накопителя, где в узлах расположены перемычки в виде проводников, полупроводниковых диодов или транзисторов, одним концом подключенные к адресной линии, а другим — к разрядной линии считывания. В такой матрице наличие перемычки может означать 1, а ее отсутствие — 0. В некоторых типах ПЗУ элемент, расположенный на перемычке, исполняет роль конденсатора. Тогда заряженное состояние конденсатора означает 1, а разряженное — 0.

Основным режимом работы ПЗУ является считывание информации, которое мало отличается от аналогичной операции в ОЗУ как по организации, так и по длительности. Именно это обстоятельство подчеркивает общепризнанное название постоянных ЗУ — ROM (Read-Only Memory — память только для чтения). В то же время запись в ПЗУ по сравнению с чтением обычно сложнее и связана с большими затратами времени и энергии. Занесение информации в ПЗУ называют программированием или «прошивкой». Последнее название напоминает о том, что первые ПЗУ выполнялись на базе магнитных сердечников, а данные в них заносились путем прошивки соответствующих сердечников проводниками считывания. Полупроводниковые микросхемы ПЗУ по возможностям и способу программирования разделяют на:

- программируемые при изготовлении;
- однократно программируемые после изготовления;
- многократно программируемые.

С появлением многократно программируемых ПЗУ популярность первых двух групп существенно снизилась. Тем не менее для полноты картины в области технологий ПЗУ кратко рассмотрим и эти две группы.

Группу *ПЗУ, программируемых при изготовлении,* образуют так называемые масочные устройства, и именно к ним принято применять аббревиатуру ПЗУ. В литературе более распространено обозначение различных вариантов постоянных ЗУ сокращениями от английских названий, поэтому в дальнейшем будем также использовать аналогичную систему. Для масочных ПЗУ таким обозначением является ROM, совпадающее с общим названием всех типов ПЗУ. Иногда такие микросхемы именуют MROM (Mask Programmable ROM — ПЗУ, программируемые с помощью маски). Занесение информации в масочные ПЗУ составляет часть производственного процесса и заключается в подключении или не подключении запоминающего элемента к разрядной линии считывания. В зависимости от этого из ЗЭ будет всегда извлекаться 1 или 0. В роли перемычки выступает транзистор, расположенный на пересечении адресной и разрядной линий. Какие именно ЗЭ должны быть подключены к выходной линии, определяет маска, «закрывающая» определенные участки кристалла.

Создание масок для ROM оправдано при производстве большого числа копий. Если требуется относительно небольшое количество микросхем с данной информацией, разумной альтернативой являются *однократно программируемые ПЗУ*. Такие микросхемы обозначают аббревиатурой PROM (Programmable ROM — *программируемые ПЗУ)*. Информация в них может быть записана только однократно. Первыми PROM стали микросхемы памяти на базе плавких предохранителей. В исходной микросхеме во всех узлах адресные линии соединены с разрядными. Занесение информации в PROM производится электрически, путем пережигания отдельных перемычек, и может быть выполнено поставщиком или потребителем спустя какое-то время после изготовления микросхемы. Подобные ПЗУ выпускались в рамках серий К556 и К1556. Позже появились ИМС, где в перемычку входили два диода, соединенные навстречу. В процессе программирования удалить перемычку можно было с помощью электрического пробоя одного из этих диодов. В любом варианте для записи информации требуется специальное оборудование — программаторы. Основными недостатками данного вида ПЗУ были большой процент брака и необходимость специальной термической тренировки после программирования, без которой надежность хранения данных была невысокой.

Многократно программируемые ПЗУ

Процедура программирования таких ПЗУ обычно предполагает два этапа: сначала производится стирание содержимого всех или части ячеек, а затем производится запись новой информации.

В этом классе постоянных запоминающих устройств выделяют несколько групп:

- EPROM (Erasable Programmable ROM — стираемые программируемые ПЗУ);
- EEPROM (Electrically Erasable Programmable ROM — электрически стираемые программируемые ПЗУ);
- флэш-память;
- PCM (Phase Change Memory) — фазовая память.

Микросхемы EPROM. В EPROM запись информации производится электрическими сигналами, так же как в PROM, однако перед операцией записи содержимое

всех ячеек должно быть приведено к одинаковому состоянию (стерто) путем воздействия на микросхему ультрафиолетовым облучением[1]. Кристалл заключен в керамический корпус, имеющий небольшое кварцевое окно, через которое и производится облучение. Чтобы предотвратить случайное стирание информации, после облучения кварцевое окно заклеивают непрозрачной пленкой. Процесс стирания может выполняться многократно. Каждое стирание занимает порядка 20 мин. Данные хранятся в виде зарядов плавающих затворов МОП-транзисторов, играющих роль конденсаторов с очень малой утечкой заряда. Заряженный ЗЭ соответствует логическому нулю, а разряженный — логической единице. Цикл программирования занимает нескольких сотен миллисекунд. Время считывания близко к показателям ROM и DRAM.

По сравнению с PROM микросхемы EPROM дороже, но возможность многократного перепрограммирования часто является определяющей. Данный вид ИМС выпускался в рамках серии К573 (зарубежный аналог — серия 27xxx).

Микросхемы EEPROM. Более привлекательным вариантом многократно программируемой памяти является электрически стираемая программируемая постоянная память EEPROM. Стирание и запись информации в эту память производятся по байтам, причем стирание — не отдельный процесс, а лишь этап, происходящий автоматически при записи. Операция записи занимает существенно больше времени, чем считывание — несколько сотен микросекунд на байт. В микросхеме используется тот же принцип хранения информации, что и в EPROM. Программирование EPROM не требует специального программатора и реализуется средствами самой микросхемы.

Выпускаются два варианта микросхем: с последовательным и параллельным доступом, причем на долю первых приходится 90% всех выпускаемых ИМС этого типа. В EEPROM с доступом по последовательному каналу (SEEPROM — Serial EEPROM) адреса, данные и управляющие команды передаются по одному проводу и синхронизируются импульсами на тактовом входе. Преимуществом SEEPROM являются малые габариты и минимальное число линий ввода/вывода, а недостатком — большое время доступа. SEEPROM выпускаются в рамках серий микросхем 24Cxxx, 25Cxxx и 93Cxxx, а параллельные EEPROM — в серии 28Cxxx.

В целом, EEPROM дороже, чем EPROM, а микросхемы имеют менее плотную упаковку ячеек, то есть меньшую емкость.

Флэш-память. Флэш-память — это энергонезависимая перепрограммируемая полупроводниковая память. Слово flash, которое можно перевести как «быстрый, мгновенный», подчеркивает относительно высокую скорость перепрограммирования. Впервые анонсированная компанией Toshiba в 1984 году, флэш-память во многом похожа на EEPROM, но использует особую технологию построения ЗЭ. Логически ЗЭ представлен элементом ИЛИ-НЕ (NOR), либо И-НЕ (NAND), поэтому можно говорить о флэш-памяти с NOR-ячейками и NAND-ячейками.

[1] Данный вид микросхем иногда обозначают как UV-EPROM (Ultra-Violet EPROM — EPROM, стираемые ультрафиолетовым облучением).

Запоминающим элементом флэш-памяти служит полевой транзистор, у которого под управляющим затвором располагается так называемый плавающий затвор, который может хранить заряд в виде электронов (рис. 6.19), причем этот заряд, даже при отсутствии питающего напряжения, может сохраняться длительное время (несколько лет).

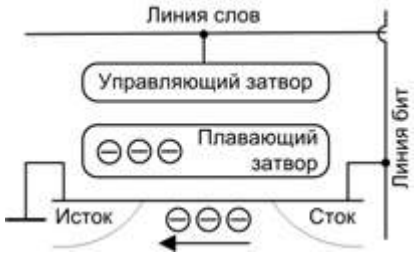


Рис. 6.19. Запоминающий элемент флэш-памяти

Состояние ЗЭ зависит от количества электронов в плавающем затворе, которое при чтении может быть измерено косвенно через величину порогового напряжения из ЗЭ. Если в плавающем затворе менее 5000 электронов, считается, что в ЗЭ записана логическая единица. Если заряд более 30 000 электронов — логический ноль.

Записи информации в ЗЭ предшествует стирание заряда с плавающего затвора (запись в ЗЭ логической 1). Для этого на исток подается высокое положительное, а на управляющий затвор — высокое отрицательное напряжение. В результате электроны с плавающего затвора «переходят» на исток (см. рис. 6.20, *а*).

Программирование заключается в записи логических нулей, то есть заряде плавающего затвора тех ЗЭ, где должен храниться логический 0. Это обеспечивается подачей на сток высокого положительного напряжения, а на управляющий затвор вдвое большего положительного напряжения. В этих условиях электроны из канала между истоком и стоком переходят на плавающий затвор (см. рис. 6.20, *б*).

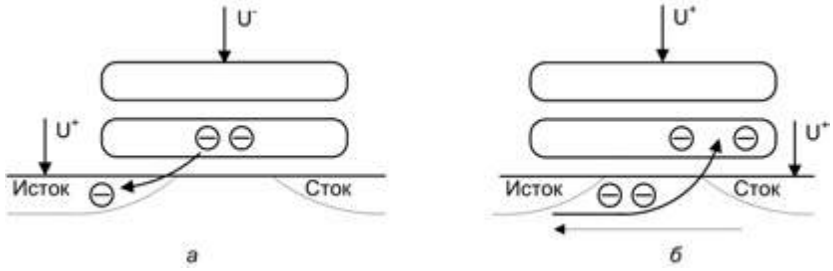


Рис. 6.20. Стирание и программирование (запись) информации: *а* — стирание информации; *б* — программирование

Полностью содержимое флэш-памяти может быть очищено за одну или несколько секунд, что значительно быстрее, чем у EEPROM. Программирование

(запись) байта занимает время порядка 10 мкс, а время доступа при чтении составляет 35–200 нс.

Флэш-память с NOR ячейками позволяет работать с отдельными байтами, аналогично EEPROM. Во флэш-памяти типа NAND ячейки группируются в небольшие блоки, и в качестве наименьшей единицы информации при записи выступает блок. В силу этих различий флэш-память с ячейками NOR выгодно использовать в случае произвольного доступа, например, для хранения BIOS в персональных компьютерах. Память с ячейками NAND более подходит при работе с большими массивами информации, например в твердотельных дисках.

Отметим, что независимо от типа ячеек флэш-память имеет ограничение по количеству циклов перезаписи (в лучшем случае — несколько миллионов), что не позволяет использовать ее как энергонезависимое ОЗУ. Кроме того, флэш-память уступает микросхемам ОЗУ по быстродействию.

Фазовая память PCM. Носителем информации служат микроскопические частицы халькогенидного стекла, которое может находиться в одном из двух состояний[1]. Эти состояния (фазы) различаются по оптическим и электрическим характеристикам[2]. Под воздействием тепла материал может переходить из одного состояния в другое. В оптических ЗУ такой нагрев достигается с помощью луча лазера. В случае PCM нагрев достигается пропусканием через материал электрического тока, причем состояние, в которое переходит нагреваемый участок, зависит от величины тока, приложенного напряжения и длительности нагревания. В обычном (холодном) состоянии материал представляет собой аморфную стеклообразную структуру с высоким электрическим сопротивлением. При воздействии высокой температуры (до 600 градусов по Цельсию) материал кристаллизуется и будет обладать очень низким сопротивлением. Высокое сопротивление в аморфной фазе используется для представления двоичного «0», а низкое сопротивление в кристаллической фазе — «1». Потенциально фазовый переход может осуществляться менее чем за 5 нс, хотя в экспериментах пока удалось добиться величины порядка 16 нс. В целом, PCM обеспечивает стирание содержимого ячеек почти в 10 раз быстрее флэш-памяти, а темп перезаписи превышает аналогичный показатель для флэш-памяти в семь раз.

PCM выгодно использовать там, где требуется быстрая запись информации, поскольку технология позволяет изменять значение отдельных битов без предварительного стирания целого блока ячеек. Кроме того, запоминающие элементы PCM обладают высоким быстродействием. Еще одно преимущество PCM перед флэш-памятью — количество циклов перезаписи без деградации микросхемы достигает 100 миллионов.

---

[1] В последних версиях удалось добавить еще два четко различимых состояния частичной кристаллизации, что позволило в одном физическом запоминающем элементе хранить два бита, то есть удвоить информационную емкость микросхем.

[2] Такой же материал используется в оптических дисках с многократной записью, но там используются оптические свойства материала, тогда как в фазовой памяти – электрические.

Энергонезависимые оперативные запоминающие устройства

Под понятие *энергонезависимое ОЗУ* (NVRAM — Non-Volatile RAM) подпадает несколько типов памяти. От перепрограммируемых постоянных ЗУ их отличает отсутствие этапа стирания, предваряющего запись новой информации, поэтому вместо термина «программирование» для них употребляют стандартный термин «запись».

Микросхемы BBSRAM. К рассматриваемой группе относятся обычные статические ОЗУ со встроенным литиевым аккумулятором и усиленной защитой от искажения информации в момент включения и отключения внешнего питания. Для их обозначения применяют аббревиатуру BBSRAM (Battery-Back SRAM).

Микросхемы NVRAM. Другой подход реализован в микросхеме, разработанной компанией Simtec. Особенность ее в том, что в одном корпусе объединены статическое ОЗУ и перепрограммируемая постоянная память типа EEPROM. При включении питания данные копируются из EEPROM в SRAM, а при выключении — автоматически перезаписываются из SRAM в EEPROM. Благодаря такому приему данный вид памяти можно считать энергонезависимым.

Микросхемы FRAM. FRAM (Ferroelectric RAM — ферроэлектрическая память) разработана компанией Ramtron и представляет собой еще один вариант энергонезависимой памяти. По быстродействию данное ЗУ несколько уступает динамическим ОЗУ и пока рассматривается лишь как альтернатива флэш-памяти. Причисление FRAM к оперативным ЗУ обусловлено отсутствием перед записью явно выраженного цикла стирания информации.

Запоминающий элемент FRAM похож на ЗЭ динамического ОЗУ, то есть состоит из конденсатора и транзистора. Отличие заключено в диэлектрических свойствах материала между обкладками конденсатора. В FRAM этот материал обладает большой диэлектрической постоянной и может быть поляризован с помощью электрического поля. Поляризация сохраняется вплоть до ее изменения противоположно направленным электрическим полем, что и обеспечивает энергонезависимость данного вида памяти. Данные считываются за счет воздействия на конденсатор электрического поля. Величина возникающего при этом тока зависит от того, изменяет ли приложенное поле направление поляризации на противоположное или нет, что может быть зафиксировано усилителями считывания. В процессе считывания содержимое ЗЭ разрушается и должно быть восстановлено путем повторной записи, то есть, как и DRAM, данный тип ЗУ требует регенерации. Количество циклов перезаписи для FRAM обычно составляет 10 млрд.

Главное достоинство данной технологии в значительно более высокой скорости записи по сравнению с EEPROM. В то же время относительная простота ЗЭ позволяет добиться высокой плотности размещения элементов на кристалле, сопоставимой с DRAM. FRAM выпускаются в виде микросхем, полностью совместимых с последовательными и параллельными EEPROM. Примером может служить серия 24Cxx.

В какой-то мере к энергонезависимым ОЗУ можно отнести и фазовую память, поскольку запись новых данных не требует предварительного удаления старых.

Сказанное подтверждает и другая аббревиатура, используемая для обозначения фазовой памяти — PRAM (Phase-change Random Access Memory), что можно перевести как ОЗУ на базе эффекта фазового перехода.

Обнаружение и исправление ошибок

При работе с полупроводниковой памятью не исключено возникновение различного рода отказов и сбоев. Причиной *отказов* могут быть производственные дефекты, повреждение микросхем или их физический износ. Проявляются отказы в том, что в отдельных разрядах одной или нескольких ячеек постоянно считывается 0 или 1, вне зависимости от реально записанной туда информации. *Сбой* — это случайное событие, выражающееся в неверном считывании или записи информации в отдельных разрядах одной или нескольких ячеек, не связанное с дефектами микросхемы. Сбои обычно обусловлены проблемами с источником питания или с воздействием альфа-частиц, возникающих в результате распада радиоактивных элементов, которые в небольших количествах присутствуют практически в любых материалах. Как отказы, так и сбои крайне нежелательны, поэтому в большинстве систем основной памяти содержатся схемы, служащие для обнаружения и исправления ошибок.

Вне зависимости от того, как именно реализуется контроль и исправление ошибок, в основе их всегда лежит введение избыточности. Это означает, что контролируемые разряды дополняются контрольными разрядами, благодаря которым и возможно детектирование ошибок, а в ряде методов — их коррекция. Общую схему обнаружения и исправления ошибок иллюстрирует рис. 6.21.

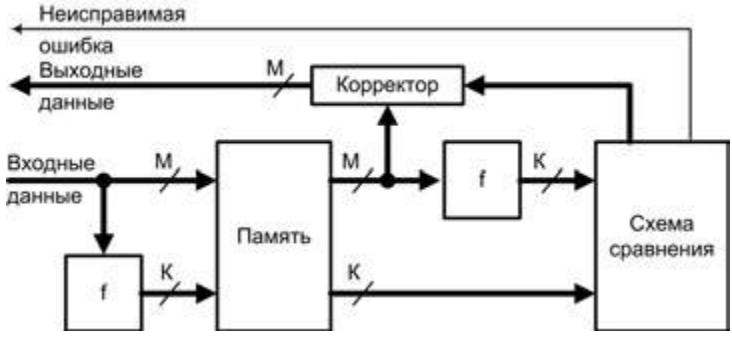


Рис. 6.21. Общая схема обнаружения и исправления ошибок [143]

Перед записью *M*-разрядных данных в память производится их обработка, обозначенная на схеме функцией «f», в результате которой формируется добавочный *K*-разрядный код. В память заносятся как данные, так и этот вычисленный код, то есть (*M* + *K*)-разрядная информация. При чтении информации повторно формируется *K*-разрядный код, который сравнивается с аналогичным кодом, считанным из ячейки. Сравнение приводит к одному из трех результатов:

- *Не обнаружено ни одной ошибки*. Извлеченные из ячейки данные подаются на выход памяти.

- *Обнаружена ошибка, и она может быть исправлена.* Биты данных и добавочного кода подаются на схему коррекции. После исправления ошибки в M-разрядных данных они поступают на выход памяти.
- *Обнаружена ошибка, и она не может быть исправлена.* Выдается сообщение о неисправимой ошибке.

Коды, используемые для подобных операций, называют *корректирующими кодами* или *кодами с исправлением ошибок*.

Простейший вид такого кода основан на добавлении к каждому байту информации одного *бита паритета*. Бит паритета — это дополнительный бит, значение которого устанавливается таким, чтобы суммарное число единиц в данных, с учетом этого дополнительного разряда, было четным (или нечетным). В некоторых системах за основу берется четность, в иных — нечетность. Для 64-разрядного слова требуется восемь битов паритета, то есть ячейка памяти должна хранить 36 разрядов. При записи слова в память для каждого байта формируется бит паритета. Это может быть сделано с помощью схемы в виде дерева, составленного из схем сложения по модулю 2. При чтении из памяти выполняется аналогичная операция над считанными информационными битами, а ее результат сравнивается с битом паритета, вычисленным при записи и хранившимся в памяти. Метод позволяет обнаружить ошибку, если исказилось нечетное количество битов. При четном числе ошибок метод неработоспособен. К сожалению, фиксируя ошибку, данный способ кодирования не может указать на ее местоположение, что позволило бы внести исправления, в силу чего его называют *кодом с обнаружением ошибки* (EDC — Error Detection Code).

В основе корректирующих кодов лежит достаточно простая идея [30]. Для контроля двоичного информационного кода длиной M битов добавим к ней K дополнительных контрольных разрядов так, что общая длина последовательности теперь будет равна $M + K$ разрядам. В этом случае из возможных $N = 2^{M+K}$ комбинаций интерес представляют только $L = 2^M$ последовательностей, которые называют *разрешенными*. Оставшиеся $N - L$ последовательностей назовем *запрещенными*. Если при обработке (записи в память, считывании или передаче) разрешенной кодовой последовательности произойдут ошибки и возникнет одна из запрещенных последовательностей, то тем самым эти ошибки обнаруживаются. Если же ошибки превратят одну разрешенную последовательность в другую, то такие ошибки не могут быть обнаружены. Для исправления ошибок необходимо произвести разбиение множества запрещенных последовательностей на L непересекающихся подмножеств и каждому подмножеству поставить в соответствие одну из разрешенных последовательностей. Тогда, если была принята некоторая запрещенная последовательность, входящая в одно из подмножеств, считается, что передана разрешенная последовательность, соответствующая этому подмножеству, производится замена, чем и исправляется возникшая ошибка.

Простейший вариант корректирующего кода также может быть построен на базе битов паритета. Для этого биты данных представляются в виде матрицы, к каждой строке и столбцу которой добавляется бит паритета. Для 64-разрядных данных этот подход иллюстрирует табл. 6.2 [143]. Здесь D — биты данных, C — столбец битов паритета строк, K — строка битов паритета столбцов, P — бит паритета,

контролирующий столбец *C* и строку *K*. Таким образом, к 64 битам данных нужно добавить 17 битов паритета: по 8 битов на строки и столбцы и один дополнительный бит для контроля строки и столбца битов паритета. Если в одной строке и одном столбце обнаружено нарушение паритета, для исправления ошибки достаточно просто инвертировать бит на пересечении этих строки и столбца. Если ошибка паритета выявлена только в одной строке или только одном столбце либо одновременно в нескольких строках и столбцах, фиксируется многобитовая ошибка и формируется признак невозможности коррекции.

Таблица 6.2. Формирование корректирующего кода для 64-битовых данных

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	
0	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7	C_0
1	D_8	D_9	D_{10}	D_{11}	D_{12}	D_{13}	D_{14}	D_{15}	C_1
2	D_{16}	D_{17}	D_{18}	D_{19}	D_{20}	D_{21}	D_{22}	D_{23}	C_2
3	D_{24}	D_{25}	D_{26}	D_{27}	D_{28}	D_{29}	D_{30}	D_{31}	C_3
4	D_{32}	D_{33}	D_{34}	D_{35}	D_{36}	D_{37}	D_{38}	D_{39}	C_4
5	D_{40}	D_{41}	D_{42}	D_{43}	D_{44}	D_{45}	D_{46}	D_{47}	C_5
6	D_{48}	D_{49}	D_{50}	D_{51}	D_{52}	D_{53}	D_{54}	D_{55}	C_6
7	D_{56}	D_{57}	D_{58}	D_{59}	D_{60}	D_{61}	D_{62}	D_{63}	C_7
	K_0	K_1	K_2	K_3	K_4	K_5	K_6	K_7	P

Недостаток рассмотренного приема в том, что он требует большого числа дополнительных разрядов. Более эффективным представляется код, предложенный Ричардом Хэммингом и носящий его имя (*код Хэмминга*).

Для пояснения концепции, положенной в основу кода Хэмминга, построим код, обнаруживающий и исправляющий однобитовые ошибки в 8-разрядных словах (пример взят из [42]).

Сначала определим требуемую длину корректирующего кода. В соответствии с рис. 6.21, на вход схемы сравнения поступают два *K*-разрядных значения. Сравнение производится путем поразрядной операции «исключающее ИЛИ» (сложение по модулю 2) над входными кодами. Результатом является так называемое *слово синдрома*. В зависимости от того, было ли совпадение входных кодов или нет, соответствующий бит синдрома будет равен 0 или 1.

Слово синдрома состоит из *K* разрядов, то есть его возможные значения лежат в диапазоне от 0 до $2^K - 1$. Значение 0 соответствует случаю, когда ошибки не обнаружено, остальные $2^K - 1$ случая свидетельствуют о наличии ошибки и указывают на ее местоположение. Поскольку ошибка может возникнуть в любом из *M* битов данных или *K* контрольных битов, мы должны иметь $2^K - 1 \geq M + K$. Это выражение позволяет определить число битов, необходимое для исправления одиночной ошибки в *M*-разрядных данных.

В табл. 6.3 приведено количество корректирующих разрядов, нужное для контроля данных различной разрядности. Из таблицы видно, что для 8-разрядных слов требуется четыре корректирующих разряда.

Для удобства будем формировать четырехразрядный синдром со следующими характеристиками.

- Если синдром содержит все нули, значит, не обнаружено ни одной ошибки.
- Если синдром содержит единственную единицу в одном из разрядов, это означает, что выявлена ошибка в одном из четырех корректирующих разрядов и никакой коррекции не требуется.
- Если в синдроме в единичное состояние установлены несколько битов, то численное значение синдрома соответствует позиции ошибки в данных, для исправления которой необходимо инвертировать бит в этой позиции.

Таблица 6.3. Количество корректирующих разрядов, используемое в коде Хэмминга

Разрядность	**Исправление одиночной ошибки**		**Исправление одиночной ошибки Обнаружение двойной ошибки**	
	Контрольные биты	**% увеличения длины кода**	**Контрольные биты**	**% увеличения длины кода**
8	4	50	5	62,5
16	5	31,25	6	37,5
32	6	18,75	7	21,875
64	7	10,94	8	12,5
128	8	6,25	9	7,03
256	9	3,52	10	3,91

Под контрольные разряды отводятся те биты, чьи позиционные номера представляют собой степень числа 2 (табл. 6.4). Отдельный контрольный разряд отвечает за определенные биты данных. Так, разрядная позиция n контролируется теми битами P_i, которые делают справедливым соотношение $\Sigma_i = n$. Например, разряд данных с позиционным номером 7_{10} (0111_2) контролируется битами 4, 2 и 1 ($7 = 4 + 2 + 1$), а разряд с номером 10_{10} (1010_2) — битами 8 и 2 ($10 = 8 + 2$).

Таблица 6.4. Распределение разрядов 12-разрядного слова между данными и контрольным кодом

Десятичный номер позиции	12	11	10	9	8	7	6	5	4	3	2	1
Двоичный код номера позиции	1100	1011	1010	1001	1000	0111	0110	0101	0100	0011	0010	0001
Биты данных	D_8	D_7	D_6	D_5		D_4	D_3	D_2		D_1		
Контрольные биты					P_8				P_4		P_2	P_1

Таким образом, для рассматриваемого случая значения контрольных разрядов вычисляются по следующим правилам:

$$P_1 = D_1 \oplus D_2 \oplus D_4 \oplus D_5 \oplus D_7;$$
$$P_2 = D_1 \oplus D_3 \oplus D_4 \oplus D_6 \oplus D_7;$$
$$P_4 = D_2 \oplus D_3 \oplus D_4 \oplus D_8;$$
$$P_8 = D_5 \oplus D_6 \oplus D_7 \oplus D_8.$$

Проверим корректность такой схемы на примере. Пусть входной код равен 00101011, где разряду D_1 соответствует правая цифра. Контрольные разряды вычисляются следующим образом:

$$P_1 = D_1 \oplus D_2 \oplus D_4 \oplus D_5 \oplus D_7 = 1 \oplus 1 \oplus 1 \oplus 0 \oplus 0 = 1;$$
$$P_2 = D_1 \oplus D_3 \oplus D_4 \oplus D_6 \oplus D_7 = 1 \oplus 0 \oplus 1 \oplus 1 \oplus 0 = 1;$$
$$P_4 = D_2 \oplus D_3 \oplus D_4 \oplus D_8 = 1 \oplus 0 \oplus 1 \oplus 0 = 0;$$
$$P_8 = D_5 \oplus D_6 \oplus D_7 \oplus D_8 = 0 \oplus 1 \oplus 0 \oplus 0 = 1.$$

Предположим, что данные в бите 3 содержат ошибку и там вместо 0 находится 1. После пересчета контрольных разрядов имеем

$$P_1' = D_1 \oplus D_2 \oplus D_4 \oplus D_5 \oplus D_7 = 1 \oplus 1 \oplus 1 \oplus 0 \oplus 0 = 1;$$
$$P_2' = D_1 \oplus D_3 \oplus D_4 \oplus D_6 \oplus D_7 = 1 \oplus 1 \oplus 1 \oplus 1 \oplus 0 = 0;$$
$$P_4' = D_2 \oplus D_3 \oplus D_4 \oplus D_8 = 1 \oplus 1 \oplus 1 \oplus 0 = 1;$$
$$P_8' = D_5 \oplus D_6 \oplus D_7 \oplus D_8 = 0 \oplus 1 \oplus 0 \oplus 0 = 1.$$

Путем сложения по модулю два результатов старой и новой проверок получим слово синдрома ($S_8S_4S_2S_1$):

$$S_1 = P_1 \oplus P_1' = 1 \oplus 1 = 0;$$
$$S_2 = P_2 \oplus P_2' = 1 \oplus 1 = 0;$$
$$S_4 = P_4 \oplus P_4' = 1 \oplus 1 = 0;$$
$$S_8 = P_8 \oplus P_8' = 1 \oplus 1 = 0.$$

Результат 0110_2 (6_{10}) означает, что в разряде 6, содержащем третий бит данных, присутствует ошибка.

Описанный код называется *кодом с исправлением одиночной ошибки* (SEC — Single Error Correcting). В большинстве микросхем памяти используется *код с исправлением одиночной и обнаружением двойной ошибки* (*SECDED* — Single Error Correcting, Double Error Detecting). Из табл. 6.3 видно, что, по сравнению с SEC, такой код требует введения одного дополнительного контрольного разряда.

Коды с исправлением ошибок применяются в большинстве ВМ. Например, в основной памяти ВМ типа IBM 30xx используется 8-разрядный код SECDED на каждые 64 бита данных, то есть емкость памяти примерно на 12% больше, чем имеет в своем распоряжении пользователь. В ВМ типа VAX на каждые 32 разряда данных добавлен 7-разрядный код SECDED, следовательно, избыточность составляет 22%.

Структура одного из вариантов построения устройства для обнаружения одинарных и двойных ошибок с коррекцией одинарных ошибок приведена на рис. 6.22. Схема предназначена для контроля 16-разрядных данных и размещается между процессором и памятью. Из табл. 6.3 видно, что код SECDED предполагает шесть дополнительных разрядов. Таким образом, из процессора и в процессор поступают 16-разрядные коды (UD_{15} ... UD_0), а в память заносятся 22-разрядные данные (M_{21} ... M_0). Хранящаяся в ячейках памяти информация состоит из 16 битов информации (MD_{15} ... MD_0) и 6 контрольных битов (MP_5 ... MP_0). В последующем первые буквы в обозначении разрядов могут быть опущены, при этом *D* будет означать информационный разряд кода, а *P* — контрольный разряд.

Система размещения основных и контрольных разрядов была рассмотрена ранее, и для данной схемы она приведена в табл. 6.5.

Таблица 6.5. Распределение информационных и контрольных разрядов в 22-разрядном слове

Номер позиции	22	21	20	19	18	17	16	15	14	13	12
Биты данных		D_{15}	D_{14}	D_{13}	D_{12}	D_{11}		D_{10}	D_9	D_8	D_7
Контрольные биты	P_5						P_4				
Номер позиции	11	10	9	8	7	6	5	4	3	2	1
Биты данных	D_6	D_5	D_4		D_3	D_2		D_1	D_0		
Контрольные биты				P_3				P_2		P_1	P_0

Контрольные разряды $P_4 ... P_0$ вычисляются по стандартной схеме кода Хэмминга:

$$P_0 = D_{15} \oplus D_{13} \oplus D_{11} \oplus D_{10} \oplus D_8 \oplus D_6 \oplus D_4 \oplus D_3 \oplus D_1 \oplus D_0;$$
$$P_1 = D_{13} \oplus D_{12} \oplus D_{10} \oplus D_9 \oplus D_6 \oplus D_5 \oplus D_3 \oplus D_2 \oplus D_0;$$
$$P_2 = D_{15} \oplus D_{14} \oplus D_{10} \oplus D_9 \oplus D_8 \oplus D_7 \oplus D_3 \oplus D_2 \oplus D_1;$$
$$P_3 = D_{10} \oplus D_9 \oplus D_8 \oplus D_7 \oplus D_6 \oplus D_5 \oplus D_4;$$
$$P_4 = D_{15} \oplus D_{14} \oplus D_{13} \oplus D_{12} \oplus D_{11}.$$

Особенность рассматриваемой схемы состоит в способе формирования контрольного разряда P_5. Он вычисляется путем суммирования по модулю 2 всех остальных 21 разрядов кода (D_{15} ... D_0 и MP_4 ... MP_0). По мнению авторов рассматриваемой схемы, это облегчает фиксацию факта неисправимой двойной ошибки.

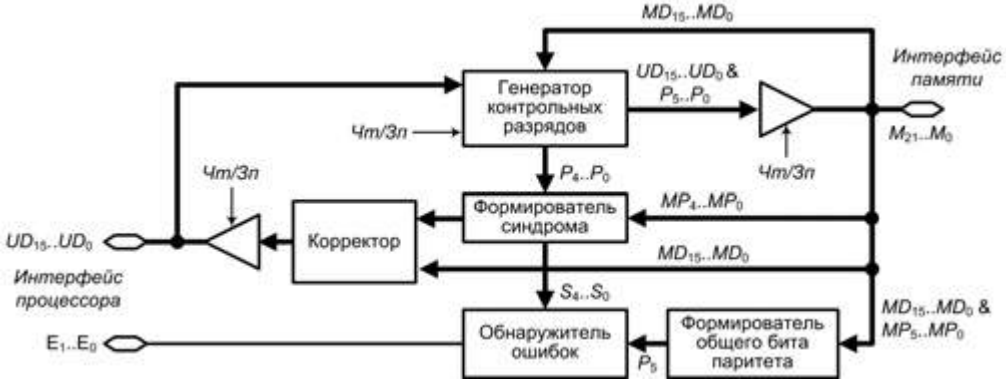


Рис. 6.22. Схема обнаружения одинарных и двойных ошибок с коррекцией одинарных, использующая код SECDED

При чтении из памяти формирователем синдрома вычисляется синдром $S_4 ... S_0$:

$$S_0 = P_0 \oplus MP_0;$$
$$S_1 = P_1 \oplus MP_1;$$
$$S_2 = P_2 \oplus MP_2;$$
$$S_3 = P_3 \oplus MP_3;$$
$$S_4 = P_4 \oplus MP_4.$$

Здесь $P_4 ... P_0$ — контрольные разряды, вычисленные генератором контрольных разрядов на основании информационных битов считанного из памяти кода ($MD_{15} ... MD_0$). $MP_5 ... MP_0$ — такие же контрольные разряды, полученные тем же генератором, но перед записью информации в память и хранившиеся там вместе с основной информацией. Если синдром не равен нулю, то он указывает на позицию исказившегося бита при одиночной ошибке. С учетом контрольного разряда P_5 возможны четыре ситуации, показанные в табл. 6.6.

Таблица 6.6. Обнаружение ошибок

Синдром	P5	Тип ошибки	Примечания
0	0	Ошибки нет	
≠0	1	Одиночная ошибка	Корректируемая: синдром указывает на позицию искаженного разряда
≠0	0	Двойная ошибка	Неисправимая
0	1	Ошибка в контрольном разряде	Искажен контрольный разряд P_5, и он может быть откорректирован

Сигнал Чт/Зп на схеме определяет выполняемую операцию. Чт/Зп = 1 означает чтение из памяти, а Чт/Зп = 0 — запись в память.

Проанализировав полученный синдром и разряд P_5, обнаружитель ошибки формирует двухразрядный код ошибки E_1E_0 (табл. 6.7).

Таблица 6.7. Кодирование ошибок

Код ошибки (E1, E0)	Описание
00	Ошибки нет
01	Одиночная ошибка — исправимая
10	Двойная ошибка — неисправимая
11	Ошибка в контрольном разряде — исправимая

Стековая память

Стековая память обеспечивает такой режим работы, когда информация записывается и считывается по принципу «*последним записан — первым считан*» (LIFO — Last In First Out). Память с подобной организацией широко применяется для запоминания и восстановления содержимого регистров процессора при обработке подпрограмм и прерываний. Работу стековой памяти поясняет рис. 6.23, *а*.

Когда слово A заносится в стек, оно располагается в первой свободной ячейке. Каждое следующее записываемое слово перемещает все содержимое стека на одну ячейку вверх и занимает освободившуюся ячейку. Запись очередного кода, после H, приводит к переполнению стека и потере кода A. Считывание кодов из стека осуществляется в обратном порядке, то есть начиная с кода H, который был записан последним. Отметим, что доступ к произвольному коду в стеке формально недопустим до извлечения всех кодов, записанных позже.

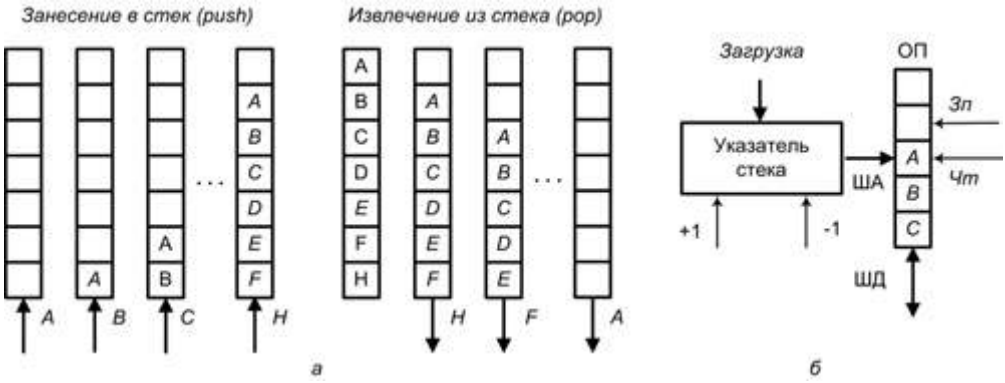


Рис. 6.23. Организация стековой памяти: *а* — логика работы; *б* — аппаратно-программный стек

Наиболее распространенным в настоящее время является внешний или аппаратно-программный стек, в котором для хранения информации используется область ОП. Обычно для этих целей отводится участок памяти с наибольшими адресами, а стек расширяется в сторону уменьшения адресов. Так как программа обычно загружается, начиная с меньших адресов, такой прием во многих случаях позволяет избежать перекрытия областей программы и стека. Адресация стека обеспечивается специальным регистром — *указателем стека* (SP — stack pointer), в который предварительно помещается наибольший адрес области основной памяти, отведенной под стек (рис. 6.23, *б*).

При занесении в стек очередного слова сначала производится уменьшение на единицу содержимого указателя стека (УС), которое затем используется как адрес ячейки, куда и производится запись, то есть указатель стека хранит адрес той ячейки, к которой было произведено последнее обращение. Это можно описать в виде: УС := УС – 1; ОП[(УС)] := ШД.

При считывании слова из стека в качестве адреса этого слова берется текущее содержимое указателя стека, а после того как слово извлечено, содержимое УС увеличивается на единицу. Таким образом, при извлечении слова из стека реализуются следующие операции: ШД := ОП[(УС)]; УС := УС + 1.

Ассоциативная память

В рассмотренных ранее видах запоминающих устройств доступ к информации требовал указания адреса ячейки. Зачастую значительно удобнее искать информацию не по адресу, а опираясь на какой-нибудь характерный признак, содержащийся в самой информации. Такой принцип лежит в основе ЗУ, известного как *ассоциативное запоминающее устройство* (АЗУ). В литературе встречаются и иные названия подобного ЗУ: память, адресуемая по содержанию (content addressable memory); память, адресуемая по данным (data addressable memory); память с параллельным поиском (parallel search memory); каталоговая память (catalog memory);

информационное ЗУ (information storage); тегированная память (tag memory). Ассоциативное ЗУ — это устройство, способное хранить информацию, сравнивать ее с некоторым заданным образцом и указывать на их соответствие или несоответствие друг другу. Признак, по которому производится поиск информации, будем называть *ассоциативным признаком*, а кодовую комбинацию, выступающую в роли образца для поиска, — *признаком поиска*. Ассоциативный признак может быть частью искомой информации или дополнительно придаваться ей. В последнем случае его принято называть *тегом* или *ярлыком*.

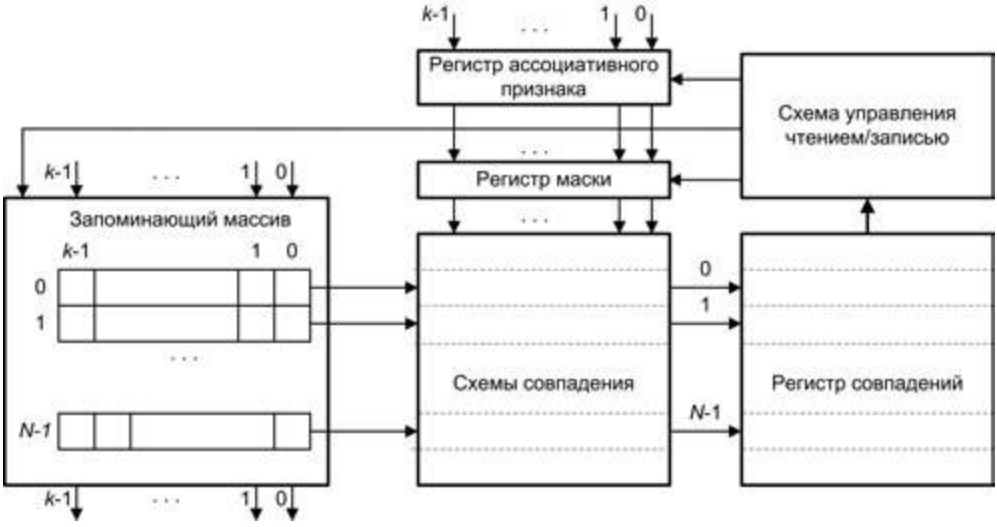


Рис. 6.24. Структура ассоциативного запоминающего устройства

Один из вариантов построения ассоциативной памяти показан на рис. 6.24. АЗУ включает в себя:

- запоминающий массив для хранения N m-разрядных слов, в каждом из которых несколько младших разрядов занимает служебная информация;
- регистр ассоциативного признака, куда помещается код искомой информации (признак поиска);
- схемы совпадения, используемые для параллельного сравнения каждого бита всех хранимых слов с соответствующим битом признака поиска и выработки сигналов совпадения;
- регистр совпадений, где каждой ячейке запоминающего массива соответствует один разряд, в который заносится единица, если все разряды соответствующей ячейки совпали с одноименными разрядами признака поиска;
- регистр маски, позволяющий запретить сравнение определенных битов;
- схему управления чтением/записью (СУЧЗ), которая на основании анализа содержимого регистра совпадений формирует сигналы, характеризующие результаты поиска информации, а также обеспечивает выбор ячейки для записи новой информации.

При обращении к АЗУ сначала в регистре маски обнуляются разряды, которые не должны учитываться при поиске информации. Все разряды регистра совпадений устанавливаются в единичное состояние. После этого в регистр ассоциативного признака заносится код искомой информации (признак поиска) и начинается ее поиск, в процессе которого схемы совпадения одновременно сравнивают первый незамаскированный бит всех ячеек запоминающего массива с первым битом признака поиска. Те схемы, которые зафиксировали несовпадение, формируют сигнал, переводящий соответствующий бит регистра совпадений в нулевое состояние. Так же происходит процесс поиска и для остальных незамаскированных битов признака поиска. В итоге единицы сохраняются лишь в тех разрядах регистра совпадений, которые соответствуют ячейкам, где находится искомая информация. Конфигурация единиц в регистре совпадений используется в качестве адресов, по которым производится считывание из запоминающего массива.

Из-за того что результаты поиска могут оказаться неоднозначными, содержимое регистра совпадений подается на СУЧЗ, где формируются сигналы, извещающие о том, что искомая информация:

- не найдена;
- содержится в одной ячейке;
- содержится более чем в одной ячейке.

Формирование содержимого регистра совпадений и упомянутых сигналов носит название *операции контроля ассоциации*. Она является составной частью операций считывания и записи, хотя может иметь и самостоятельное значение.

При считывании сначала производится контроль ассоциации по аргументу поиска. Если информация не найдена, считывание отменяется. При обнаружении лишь одного совпадения считывается слово, на которое указывает единица в регистре совпадений, а при большем числе совпадений сбрасывается самая старшая единица в регистре совпадений, и извлекается соответствующее ей слово. Повторяя эту операцию, можно последовательно считать все слова. В любом случае, чтение заканчивается, когда все разряды регистра совпадений будут содержать нули.

Запись в АП производится без указания конкретного адреса, в первую свободную ячейку. Для отыскания свободной ячейки СУЧЗ организует операцию считывания, в которой не замаскированы только служебные разряды, показывающие, как давно производилось обращение к данной ячейке, и свободной считается либо пустая ячейка, либо та, которая дольше всего не использовалась.

Главное преимущество ассоциативных ЗУ определяется тем, что время поиска информации зависит только от числа разрядов в признаке поиска и скорости опроса разрядов и не зависит от числа ячеек в запоминающем массиве.

Общность идеи ассоциативного поиска информации отнюдь не исключает разнообразия архитектур АЗУ. Конкретная архитектура определяется сочетанием четырех факторов: вида поиска информации; техники сравнения признаков; способа считывания информации при множественных совпадениях и способа записи информации.

В каждом конкретном применении АЗУ задача поиска информации может формулироваться по-разному (рис. 6.25).

Рис. 6.25. Классификация АЗУ по виду поиска информации в запоминающем массиве

При простом поиске требуется полное совпадение всех разрядов признака поиска с одноименными разрядами слов, хранящихся в запоминающем массиве.

При соответствующей организации АЗУ способно к более сложным вариантам поиска, с частичным совпадением. Можно, например, ставить задачу поиска слов с максимальным или минимальным значением ассоциативного признака. Многократная выборка из АЗУ слова с максимальным или минимальным значением ассоциативного признака (с исключением его из дальнейшего поиска), по существу, представляет собой упорядоченную выборку информации. Упорядоченную выборку можно обеспечить и другим способом, если вести поиск слов, ассоциативный признак которых по отношению к признаку опроса является ближайшим большим или меньшим значением.

Еще одним вариантом сложного поиска может быть нахождение слов, численное значение признака в которых больше или меньше заданной величины. Подобный подход позволяет вести поиск слов с признаками, лежащими внутри или вне заданных пределов.

Очевидно, что реализация сложных методов поиска связана с соответствующими изменениями в архитектуре АЗУ, в частности с усложнением схемы ЗУ и введением в нее дополнительной логики.

При построении АЗУ выбирают из четырех вариантов организации опроса содержимого памяти (рис. 6.26).

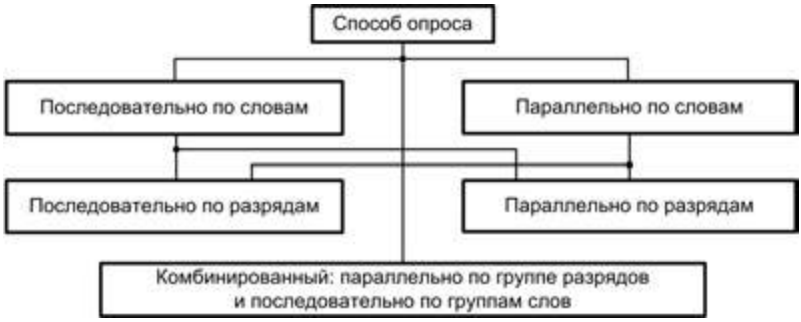


Рис. 6.26. Классификация АЗУ по способу опроса содержимого запоминающего массива

Варианты эти могут комбинироваться параллельно по группе разрядов и последовательно по группам. В плане времени поиска наиболее эффективным можно считать параллельный опрос как по словам, так и по разрядам, но не все виды запоминающих элементов допускают такую возможность.

Важным моментом является организация считывания из АЗУ информации, когда с признаком поиска совпадают ассоциативные признаки нескольких слов. В этом случае применяется один из двух подходов (рис. 6.27).

Рис. 6.27. Классификация АЗУ по способу выборки совпавших слов при множественных совпадениях

Цепь очередности реализуется с помощью достаточно сложного устройства, где фиксируются слова, образующие многозначный ответ. Цепь очередности позволяет производить считывание слов в порядке возрастания номера ячейки АЗУ независимо от величины ассоциативных признаков.

При алгоритмическом способе извлечения многозначного ответа выборка производится в результате серии опросов. Серия опросов может формироваться путем упорядочивания тех разрядов, которые были замаскированы и не участвовали в признаке поиска. В другом варианте для этих целей выделяются специальные разряды. Существует целый ряд алгоритмов, позволяющих организовать упорядоченную выборку из АЗУ. Подробное их описание и сравнительный анализ можно найти в [13].

Существенные отличия в архитектурах АЗУ могут быть связаны с выбранным принципом записи информации. Ранее был описан вариант с записью в незанятую ячейку с наименьшим номером. На практике применяются и иные способы (рис. 6.28), из которых наиболее сложный — запись с сортировкой информации на входе АЗУ по величине ассоциативного признака. Здесь местоположение ячейки, куда будет помещено новое слово, зависит от соотношения ассоциативных признаков вновь записываемого слова и уже хранящихся в АЗУ слов.

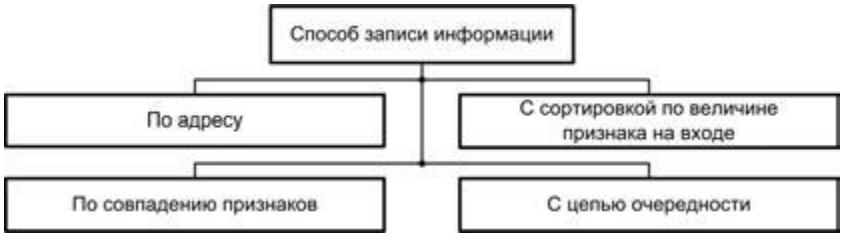


Рис. 6.28. Классификация АЗУ по способу записи информации

Из-за относительно высокой стоимости АЗУ редко используется как самостоятельный вид памяти.

Кэш-память

В большинстве ВМ основная память строится на базе микросхем динамических ОЗУ, на порядок уступающих по быстродействию центральному процессору. Замена динамических запоминающих устройств на статические ведет к весьма существенному удорожанию ОП. Экономически приемлемое решение этой проблемы, предложенное М. Уилксом, заключается в том, что между ОП и процессором размещается небольшая, но быстродействующая буферная память, куда в процессе работы копируются те участки ОП, к которым производится обращение со стороны процессора. Выигрыш достигается за счет ранее рассмотренного свойства локальности – если скопировать содержимое участка ОП в более быстродействующую буферную память и переадресовать на нее все обращения в пределах скопированного участка, то можно существенно сократить среднее время доступа к информации. Упомянутая буферная память получила название *кэш-память* (от английского слова *cache* — убежище, тайник), поскольку она обычно скрыта от программиста в том смысле, что он не может ее адресовать и может даже вообще не знать о ее существовании.

В общем виде использование кэш-памяти (КП) поясним следующим образом. Когда ЦП пытается прочитать слово из основной памяти, сначала осуществляется поиск копии этого слова в КП. Если такая копия существует, обращение к ОП не производится, а в ЦП передается слово, извлеченное из кэш-памяти. Данную ситуацию принято называть успешным обращением или *попаданием* (hit). При отсутствии слова в кэш-памяти, то есть при неуспешном обращении (*промахе* miss), блок данных, содержащий это слово, сначала пересылается из ОП в кэш-память, а затем уже из КП затребованное слово передается в ЦП.

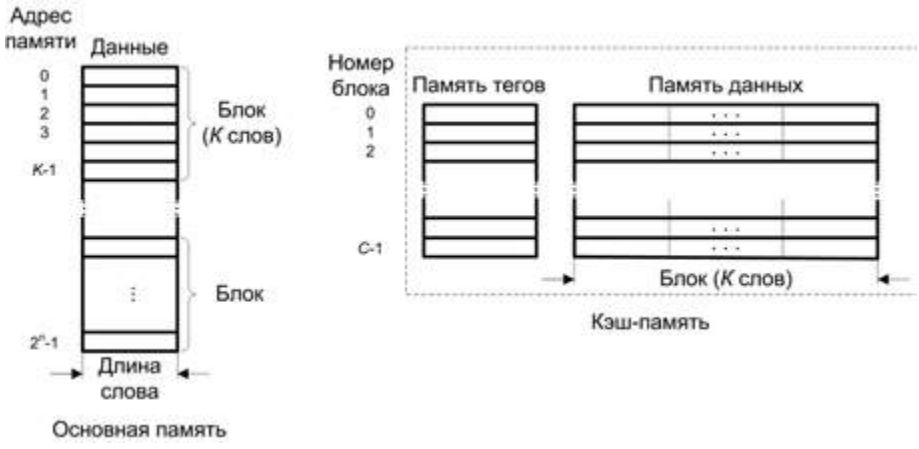


Рис. 6.29. Структура системы с основной и кэш-памятью

На рис. 6.29 приведена структура системы памяти, включающая как основную, так и кэш-память. ОП емкостью 2^n слов при взаимодействии с кэш-памятью рассматривается как состоящее из M блоков фиксированной длины по K слов в каждом

($M = 2^n/K$). Кэш-память также условно разбивается на C блоков аналогичного размера, причем $C << M$. При обращении к какой-либо ячейке ОП блок, содержащий данную ячейку, копируется в какой-то из блоков кэш-памяти, и все последующие обращения к блоку переадресовываются на его копию в КП. Ввиду малой емкости в кэш-памяти в разное время могут находиться копии различных блоков ОП. По этой причине каждую копию необходимо снабдить так называемым *тегом* (признаком), указывающим на то, копия какого блока ОП в настоящий момент хранится в данном блоке КП. В результате кэш-память представляет собой совокупность двух ЗУ, одно из которых предназначено для хранения копий данных из ОП (память данных), а второе — для хранения тегов этих копий (память тегов). Каждому блоку в памяти данных соответствует одна ячейка в памяти тегов.

Эффективность применения кэш-памяти в иерархической системе памяти зависит от целого ряда факторов, к наиболее существенным из которых следует отнести:

- емкость кэш-памяти;
- размер блока;
- способ отображения основной памяти на кэш-память;
- алгоритм замещения информации в заполненной кэш-памяти;
- алгоритм согласования содержимого основной и кэш-памяти;
- число уровней кэш-памяти.

Емкость кэш-памяти

Выбор емкости кэш-памяти — это всегда определенный компромисс. С одной стороны, кэш-память должна быть достаточно мала, чтобы ее стоимостные показатели были близки к величине, характерной для ОП. С другой — она должна быть достаточно большой, чтобы среднее время доступа в системе, состоящей из основной и кэш-памяти, определялось временем доступа к кэш-памяти.

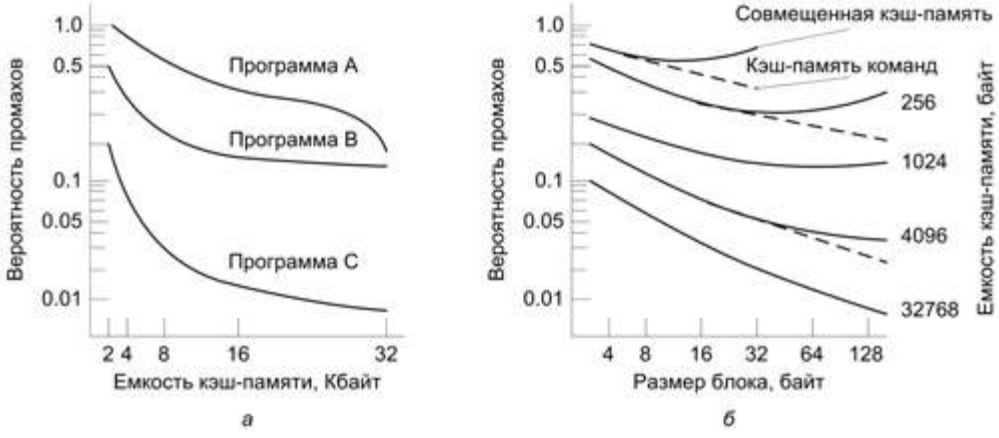


Рис. 6.30. Зависимость вероятности промахов от: *а* — емкости кэш-памяти; *б* — размера блока

Кроме того, чем вместительнее кэш-память, тем больше логических схем должно участвовать в ее адресации. Как следствие, микросхемы повышенной емкости работают медленнее по сравнению с микросхемами меньшей емкости, даже если они выполнены по одной и той же технологии.

Реальная эффективность использования кэш-памяти зависит от характера решаемых задач, и невозможно заранее определить, какая ее емкость будет действительно оптимальной. Рисунок 6.30, *а* иллюстрирует зависимость вероятности промахов от емкости кэш-памяти для трех программ A, B и C [140]. Несмотря на очевидные различия, просматривается и общая тенденция: по мере увеличения емкости кэш-памяти вероятность промахов сначала существенно снижается, но при достижении определенного значения эффект сглаживается и становится несущественным. В настоящее время считается, что для большинства задач близкой к оптимальной является емкость кэш-памяти от 1 до 512 Кбайт.

Размер блока

Еще одним важным фактором, влияющим на эффективность использования кэш-памяти, служит размер блока. Когда в кэш-память помещается блок, вместе с требуемым словом туда попадают и соседние слова. По мере увеличения размера блока вероятность промахов сначала падает, так как в кэш, согласно принципу локальности, попадает все больше данных, которые понадобятся в ближайшее время. Однако когда размер блока становится излишне большим, вероятность промахов начинает расти (рис. 6. 30 *б*). Объясняется это тем, что:

- большие размеры блока уменьшают общее количество блоков, которые можно загрузить в кэш-память, а малое число блоков приводит к необходимости частой их смены;
- по мере увеличения размера блока каждое дополнительное слово оказывается дальше от запрошенного, поэтому такое дополнительное слово менее вероятно понадобится в ближайшем будущем.

Зависимость между размером блока и вероятностью промахов во многом определяется характеристиками конкретной программы, из-за чего трудно рекомендовать определенное значение величины блока. Исследования [131, 140] показывают, что наиболее близким к оптимальному можно признать размер блока, равный 4–8 адресуемым единицам (словам или байтам). На практике его обычно выбирают сравнительно небольшим, равным ширине шины данных, связывающей кэш-память с основной памятью.

Способы отображения оперативной памяти на кэш-память

Сущность отображения блока основной памяти на кэш-память состоит в его копировании в какой-либо из блоков кэш-памяти, после чего все обращения к блоку в ОП должны переадресовываться на копию в кэш-памяти.

С целью облегчения понимания комплекса вопросов, возникающих при выборе способа отображения оперативной памяти на кэш-память, будем рассматривать систему, состоящую из основной памяти емкостью 256К слов и кэш-памяти емкостью

2К слова. Для адресации произвольного слова в основной памяти необходим 18-разрядный адрес (2^{18} = 256К), а в кэш-памяти — 11-разрядный адрес (2^{11} = 2К). Как ОП, так и кэш-память условно разбиваются на блоки по 16 слов в каждом. При такой организации 18-разрядный адрес ОП можно условно разделить на две части: младшие 4 разряда определяют адрес слова в пределах блока, а старшие 14 — номер одного из 16 384 блоков. Эти старшие 14 разрядов в дальнейшем будем называть *адресом блока основной памяти*. 11-разрядный адрес слова в кэш-памяти также можно представить состоящим из двух частей: адреса слова в блоке (4 младших разряда) и *адреса блока кэш-памяти* (7 старших разрядов).

Центральный процессор (ЦП) всегда обращается к информации в предположении, что она хранится в основной памяти и выставляет на шину адреса 18-разрядный адрес ячейки ОП. Если в кэш-памяти уже находится копия блока, куда входит ячейка, необходимо переадресовать обращение ЦП на кэш-память. Иными словами, нужно преобразовать 18-разрядный адрес ячейки ОП в 11-разрядный адрес ее копии в кэш-памяти. Фактически задача сводится к преобразованию 14-разрядного адреса блока основной памяти в 7-разрядный адрес блока кэш-памяти.

Задача отображения сводится к выбору подходящего места в кэш-памяти для очередного копируемого блока ОП. Удачным может быть признан лишь такой способ отображения, который одновременно отвечает трем требованиям: обеспечивает быструю проверку кэш-памяти на наличие в ней копии блока основной памяти; обеспечивает быстрое преобразование адреса блока ОП в адрес блока КП; реализует достижение первых двух требований наиболее экономными средствами.

Известные варианты отображения основной памяти на кэш можно свести к трем видам: прямому, полностью ассоциативному и частично-ассоциативному.

Прямое отображение

При *прямом отображении* множество блоков основной памяти условно представляется в виде матрицы, в которой количество строк равно числу блоков в кэш-памяти m (рис. 6.31). В блок кэш-памяти с номером i может быть помещен любой блок ОП, но только из i-й строки матрицы.

Номер строки (i) и столбца (k) матрицы, на пересечении которых располагается блок основной памяти с адресом j, определяются выражениями $i = j \bmod m$ и $k = j \operatorname{div} m$. В двоичной записи адреса блока основной памяти номер строки представлен $\log_2 m$ младшими разрядами, а номер столбца — оставшимися старшими разрядами. Такая система позволяет по адресу блока основной памяти, поступившему из ЦП, сразу же однозначно определить адрес блока кэш-памяти, куда должен быть отображен данный блок ОП (если его копия в КП отсутствует) или где следует искать соответствующую копию (если она присутствует в КП).

В качестве указателя того, какой из блоков i-й строки отображен или должен быть отображен на i-й блок КП (тега), используется номер столбца матрицы, где расположен отображенный (отображаемый) блок ОП.

Применительно к системе, предложенной в качестве примера, это означает, что 14-разрядный адрес блока основной памяти может рассматриваться как состоящий из 7-разрядного номера столбца (тега) и 7-разрядного номера строки. При

выставлении процессором адреса ячейки основной памяти сначала проверяется наличие в кэш-памяти копии блока, содержащего эту ячейку. Для этого из исполнительного адреса ячейки выделяется адрес блока ОП, к которому относится данная ячейка (обозначим его *j*). Младшие разряды *j* указывают на тот единственный блок кэш-памяти *i*, где может находиться копия, и одновременно на соответствующую ячейку памяти тегов. Тег сравнивается с 7 старшими разрядами поступившего из ЦП адреса. Совпадение означает, что в кэш-памяти находится копия затребованного блока ОП, и процессор адресуется к кэш-памяти. Несовпадение порождает копирование затребованного блока в кэш-память с одновременной записью в память тегов номера колонки, из которой был скопирован блок.

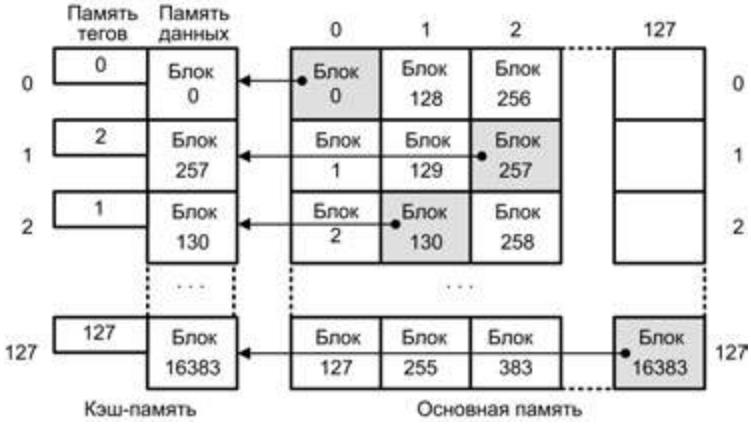


Рис. 6.31. Организация кэш-памяти с прямым отображением

Прямое отображение — простой и недорогой в реализации способ отображения. К его достоинствам можно отнести малую разрядность тега (для нашего примера она равна 7 битам). Кроме того, при проверке наличия в КП нужной копии достаточно проверить лишь одну определенную ячейку памяти тегов. Основной его недостаток — жесткое закрепление за определенными блоками ОП одного блока в КП. Поэтому если программа поочередно обращается к словам из двух различных блоков, отображаемых на одну и тут же строку кэш-памяти, постоянно станет происходить обновление данной строки и вероятность попадания будет низкой.

Полностью ассоциативное отображение

Полностью ассоциативное отображение позволяет преодолеть недостаток прямого, разрешая загрузку любого блока ОП в любой блок кэш-памяти. В этих условиях тегом служит полный адрес блока в основной памяти, то есть для нашего примера длина тега составляет 14 битов. Для проверки наличия копии блока в кэш-памяти необходимо сравнить теги всех блоков кэш-памяти на совпадение со старшими разрядами адреса, поступившего из ЦП. Такое сравнение желательно выполнять одновременно для всех ячеек памяти тегов. Этому требованию наилучшим образом отвечает ассоциативная память, то есть память тегов должна быть ассоциативной. Логика ассоциативного отображения иллюстрируется рис. 6.32.

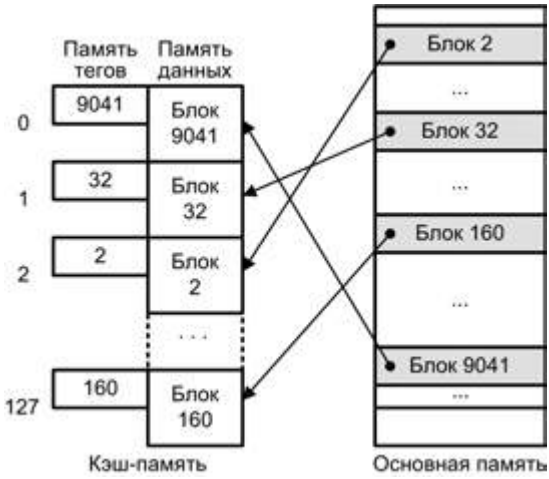


Рис. 6.32. Кэш-память с полностью ассоциативным отображением

Полностью ассоциативное отображение обеспечивает гибкость при выборе строки для вновь записываемого блока. Принципиальный недостаток этого способа — необходимость использования дорогостоящей ассоциативной памяти, причем разрядность ячейки этой памяти достаточно велика (в примере — 14 битов против 7 битов в случае прямого отображения).

Частично-ассоциативное отображение

Частично-ассоциативное отображение, известное также как множественно-ассоциативное (set-associative), является одним из возможных компромиссов, сочетающим достоинства прямого и полностью ассоциативного способов отображения. В определенной мере, оно свободно от их недостатков. Кэш-память разбивается на v подмножеств (в дальнейшем будем называть такие подмножества модулями), каждое из которых содержит k блоков. Иными словами, память тегов и память данных условно представляется матрицами, имеющими v строк и k столбцов. Кэш-память с такой организацией принято называть k-канальной (k-way) или кэш-памятью со степенью ассоциативности k. Основная память также рассматривается как матрица с тем же числом строк v. Зависимость между отдельным модулем и блоками ОП такая же, как и при прямом отображении: на блоки, входящие в модуль i, могут быть отображены только вполне определенные блоки основной памяти, в соответствии с соотношением $i = j \bmod v$, где j — адрес блока ОП. Это означает, что блоки из i-й строки матрицы блоков ОП могут отображаться лишь на блоки из i-й строки матрицы блоков кэш-памяти, причем на любой из этих блоков. Так как копия блока ОП может находиться в любом из блоков модуля кэш-памяти, для ускорения поиска нужного блока в пределах модуля используется ассоциативный принцип.

На рис. 6.33 показан пример четырехканальной кэш-памяти с частично-ассоциативным отображением. Память данных кэш-памяти разбита на 32 модуля по 4 блока в каждом. Память тегов содержит 32 ячейки, в каждой из которых может храниться 4 значения тегов (по одному на каждый блок модуля). 14-разрядный адрес

блока ОП представляется в виде двух полей: 9-разрядного поля тега и 5-разрядного поля номера модуля. Номер модуля однозначно указывает на один из модулей кэш-памяти. Он также позволяет определить номера тех блоков ОП, которые можно отображать на этот модуль. Такими являются блоки ОП, номера которых при делении на 2^5 дают в остатке число, совпадающее с номером данного модуля кэш-памяти. Так, блоки 0, 32, 64, 96 и т. д. основной памяти отображаются на модуль с номером 0; блоки 1, 33, 65, 97 и т. д. отображаются на модуль 1 и т. д. Любой из блоков в последовательности может быть загружен в любой из четырех блоков соответствующего модуля.

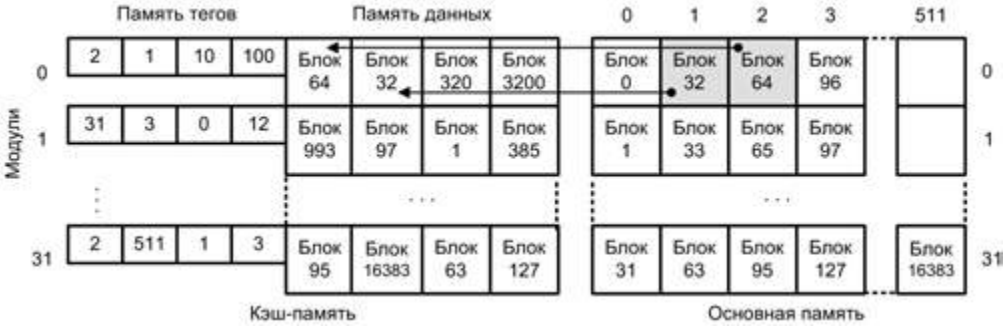


Рис. 6.33. Кэш-память с четырехканальным частично-ассоциативным отображением

При такой постановке роль тега выполняют 9 старших разрядов адреса блока ОП, в которых содержится порядковый номер блока в последовательности блоков, отображаемых на один и тот же модуль кэш-памяти. Например, блок 65 в последовательности блоков, отображаемых на модуль 1, имеет порядковый номер 2 (отсчет ведется от 0).

При обращении к кэш-памяти 5-разрядный номер модуля указывает на конкретную ячейку памяти тегов (это соответствует прямому отображению). Далее производится параллельное сравнение каждого из четырех тегов, хранящихся в этой ячейке, с полем тега поступившего адреса, то есть поиск нужного тега среди четырех возможных осуществляется ассоциативно.

В предельных случаях, когда $v = m$, $k = 1$, частично-ассоциативное отображение сводится к прямому, а при $v = 1$, $k = m$ — к полностью ассоциативному.

Наиболее общий вид организации частично-ассоциативного отображения — использование двух блоков на модуль ($v = m/2$, $k = 2$). Четырехканальная частично-ассоциативная кэш-память ($v = m/4$, $k = 4$) дает дополнительное улучшение за сравнительно небольшую дополнительную цену [96, 118]. Дальнейшее увеличение числа блоков в модуле существенного эффекта не дает.

Следует отметить, что именно этот способ отображения наиболее широко распространен в современных микропроцессорах. Наряду с 2- и 4-канальной кэш-памятью первого уровня в последних моделях микропроцессоров применяется и 16-канальная организация, в частности, при построении кэш-памяти второго уровня.

Алгоритмы замещения информации в заполненной кэш-памяти

Ввиду малой емкости в процессе вычислений все блоки кэш-памяти обычно заняты. Занесение в кэш-память копии еще одного блока ОП возможно лишь путем замещения содержимого одного из занятых блоков кэш-памяти. В этой ситуации возникает вопрос оптимального выбора замещаемого блока. Естественно, что наилучшим решением была бы замена блока, обращение к которому произойдет в более отдаленном времени или вообще не случится.

При прямом отображении каждому блоку основной памяти соответствует только один определенный блок в кэш-памяти, и никакой иной выбор удаляемого блока здесь невозможен. При полностью и частично ассоциативных способах отображения требуется какой-либо алгоритм замещения (выбора удаляемого из кэш-памяти блока). К сожалению, однозначно предсказать, какой из блоков кэш-памяти в будущем будет наименее востребован, практически нереально, и приходится привлекать алгоритмы, уступающие оптимальному. Среди возможных алгоритмов замещения наиболее распространенными являются четыре, рассматриваемые в порядке уменьшения их относительной эффективности.

Наиболее эффективным является алгоритм замещения на основе *наиболее давнего использования* (LRU — Least Recently Used), при котором замещается тот блок кэш-памяти, к которому дольше всего не было обращения. Проводившиеся исследования показали, что алгоритм LRU, который «смотрит» назад, работает достаточно хорошо в сравнении с оптимальным алгоритмом, «смотрящим» вперед.

Наиболее известны два способа аппаратурной реализации этого алгоритма.

В первом из них с каждым блоком кэш-памяти ассоциируют счетчик. К содержимому всех счетчиков через определенные интервалы времени добавляется единица. При обращении к блоку ее счетчик обнуляется. Таким образом, наибольшее число будет в счетчике того блока, к которому дольше всего не было обращений, и этот блок — первый кандидат на замещение.

Второй способ реализуется с помощью очереди, куда в порядке заполнения блоков кэш-памяти заносятся ссылки на эти блоки. При каждом обращении к блоку ссылка на него перемещается в конец очереди. В итоге первой в очереди каждый раз оказывается ссылка на блок, к которому дольше всего не было обращений. Именно этот блок прежде всего и замещается.

Другой возможный алгоритм замещения — алгоритм, работающий по принципу «*первый вошел, первый вышел*» (FIFO — First In First Out). Здесь заменяется блок, дольше всего находившийся в кэш-памяти. Алгоритм легко реализуется с помощью рассмотренной ранее очереди, с той лишь разницей, что после обращения к блоку положение соответствующей ссылки в очереди не меняется.

Еще один алгоритм — замена *наименее часто использовавшегося* блока (LFU — Least Frequently Used). Замещается тот блок в кэш-памяти, к которому было меньше всего обращений. Принцип можно воплотить на практике, связав каждый блок кэш-памяти со счетчиком обращений, к содержимому которого после каждого

обращения добавляется единица. Главным претендентом на замещение является блок, счетчик которого содержит наименьшее число.

Простейший алгоритм — *произвольный выбор* блока для замены. Замещаемый блок выбирается случайным образом. Реализовано это может быть, например, с помощью счетчика, содержимое которого увеличивается на единицу с каждым тактовым импульсом, вне зависимости от того, имело место попадание или промах. Значение в счетчике определяет заменяемый блок в полностью ассоциативной кэш-памяти или блок в пределах модуля для частично-ассоциативной кэш-памяти. Данный алгоритм используется крайне редко.

Среди известных в настоящее время систем с кэш-памятью наиболее встречаемым является алгоритм LRU.

Алгоритмы согласования содержимого кэш-памяти и основной памяти

В процессе вычислений ЦП может не только считывать имеющуюся информацию, но и записывать новую, обновляя тем самым содержимое кэш-памяти. С другой стороны, многие устройства ввода/вывода умеют напрямую обмениваться информацией с основной памятью. В обоих вариантах возникает ситуация, когда содержимое блока кэш-памяти и соответствующего блока ОП перестает совпадать. В результате на связанное с основной памятью устройство вывода может быть выдана «устаревшая» информация, поскольку все изменения в ней, сделанные процессором, фиксируются только в кэш-памяти, а процессор будет использовать старое содержимое кэш-памяти вместо новых данных, загруженных в ОП из устройства ввода.

Для разрешения первой из рассмотренных ситуаций (когда процессор выполняет операцию записи) в системах с кэш-памятью предусмотрены методы обновления основной памяти, которые можно разбить на две большие группы: *метод сквозной записи* (write through, store through) и *метод обратной записи* (write back, copy back).

По методу сквозной записи прежде всего обновляется слово, хранящееся в основной памяти. Если в кэш-памяти существует копия этого слова, то она также обновляется. Если же в кэш-памяти отсутствует нужная копия, то либо из основной памяти в кэш-память пересылается блок, содержащий обновленное слово (*сквозная запись с отображением*), либо этого не делается (*сквозная запись без отображения*).

Главное достоинство метода сквозной записи состоит в том, что если блок кэш-памяти освобождается для хранения другого блока, то удаляемый блок можно не возвращать в основную память, поскольку его копия там уже имеется. Метод достаточно прост в реализации. К сожалению, эффект от использования кэш-памяти (сокращение времени доступа) в отношении к операциям записи здесь отсутствует. Определенный выигрыш дает его модификация, известная как *метод буферизированной сквозной записи*. Информация сначала записывается в кэш-память и в специальный буфер, работающий по схеме FIFO. Запись в основную память производится уже из буфера, а процессор, не дожидаясь ее окончания, может сразу же продолжать свою работу. Конечно, соответствующая логика управления должна заботиться о том, чтобы своевременно «опустошать» заполненный буфер. При использовании буферизации процессор полностью освобождается от работы с ОП.

Согласно методу обратной записи, слово заносится только в кэш-память. Если соответствующего блока в кэш-памяти нет, то нужный блок сначала пересылается из ОП, после чего запись все равно выполняется исключительно в кэш-память. При замещении блока его необходимо предварительно переслать в соответствующее место основной памяти. Для метода обратной записи, в отличие от алгоритма сквозной записи, характерно то, что при каждом чтении из основной памяти осуществляются две пересылки между основной и кэш-памятью.

У рассматриваемого метода есть разновидность — *метод флаговой обратной записи*. Когда в каком-то блоке кэш-памяти производится изменение, устанавливается связанный с этим блоком бит изменения (флажок). При замещении блок из кэш-памяти переписывается в ОП только тогда, когда его флажок установлен в 1. Ясно, что эффективность флаговой обратной записи несколько выше. В среднем обратная запись на 10% эффективнее сквозной записи, но для ее реализации требуются и повышенные аппаратные затраты. С другой стороны, практика показывает, что операции записи составляют небольшую долю от общего количества обращений к памяти. Так, в [139] приводится величина 16%. Другие авторы оценивают долю операций записи величинами в диапазоне от 5 до 34%. Таким образом, различие по быстродействию между рассмотренными методами невелико.

Теперь рассмотрим ситуацию, когда в основную память из устройства ввода, минуя процессор, заносится новая информация, и неверной становится копия, хранящаяся в кэш-памяти. Предотвратить подобную несогласованность позволяют два приема. В первом случае система строится так, чтобы ввод любой информации в ОП автоматически сопровождался соответствующими изменениями в кэш-памяти. Для второго подхода «прямой» доступ к основной памяти допускается только через кэш-память.

Смешанная и разделенная кэш-память

Когда в микропроцессорах впервые стали применять внутреннюю кэш-память, ее обычно использовали как для команд, так и для данных. Такую кэш-память принято называть *смешанной*, а соответствующую архитектуру — *Принстонской* (Princeton architecture), по названию университета, где разрабатывались ВМ с единой памятью для команд и данных, то есть соответствующие классической архитектуре фон-Неймана. Сравнительно недавно стало обычным разделять кэш-память на две — отдельно для команд и отдельно для данных. Подобная архитектура получила название *Гарвардской* (Harvard architecture), поскольку именно в Гарвардском университете был создан компьютер «Марк-1» (1950 год), имевший раздельные ЗУ для команд и данных.

Смешанная кэш-память обладает тем преимуществом, что при заданной емкости ей свойственна более высокая вероятность попаданий, так как в ней оптимальный баланс между командами и данными устанавливается автоматически. Так, если в выполняемом фрагменте программы обращения к памяти связаны в основном с выборкой команд, а доля обращений к данным относительно мала, кэш-память имеет тенденцию насыщаться командами, и наоборот.

С другой стороны, при раздельной кэш-памяти выборка команд и данных может производиться одновременно, при этом исключаются возможные конфликты. Последнее обстоятельство существенно в системах, использующих конвейеризацию команд, где процессор извлекает команды с опережением и заполняет ими буфер или конвейер. Раздельная схема обычно характерна для кэш-памяти первого уровня, причем кэш-память команд и кэш-память данных могут быть организованы поразному, например в них могут отличаться методы отображения или замещения информации.

Следует добавить, что в некоторых ВМ, помимо кэш-памяти команд и кэш-памяти данных, может использоваться и адресная кэш-память (в устройствах управления памятью и при преобразовании виртуальных адресов в физические).

Одноуровневая и многоуровневая кэш-память

Логично предположить, что с увеличением емкости кэш-памяти можно ожидать и соответствующего повышения быстродействия ВМ. Современные технологии позволяют разместить на общем с процессором кристалле кэш-память достаточно большой емкости. С другой стороны, попытки увеличения емкости обычно приводят к снижению быстродействия, главным образом из-за усложнения схем управления и дешифрации адреса. По этой причине общую емкость кэш-памяти ВМ увеличивают за счет иерархической организации, при которой кэш-память состоит из нескольких уровней запоминающих устройств, отличающихся емкостью и быстродействием. Каждый последующий уровень имеет большую емкость по сравнению с предыдущим, но и меньшее быстродействие.

Первый уровень (L1) иерархии образует наиболее скоростная кэш-память сравнительно небольшой емкости (не более 128 Кбайт). На кристалле ее располагают по возможности ближе к процессору, чтобы минимизировать длину соединяющей их шины и тем самым способствовать ускорению обмена информацией. Этот уровень в перспективных микропроцессорах обычно строится по разделенной схеме. Так, в многоядерном процессоре Nehalem фирмы Intel каждому ядру (процессору) придается кэш-память команд емкостью 32 Кбайт (4-входовая, модульно-ассоциативная) и кэш-память данных емкостью 32 Кбайт (8-входовая, модульно-ассоциативная). В многоядерном микропроцессоре Phenom фирмы AMD каждое ядро оснащено кэш-памятью команд и кэш-памятью данных емкостью по 64 Кбайт каждая.

Емкость кэш-памяти второго уровня (L2) обычно больше, чем у L1, а быстродействие и стоимость — несколько ниже. В упоминавшихся микропроцессорах каждому ядру придается кэш-память L2, но уже смешанного типа. В Nehalem она составляет 256 Кбайт, а в Phenom — 512 Кбайт. В современных микропроцессорах кэш-память второго уровня также размещают на одном кристалле с процессором, за счет чего сокращается длина связей и повышается быстродействие.

Для новейших многоядерных микропроцессоров типично размещение на том же кристалле еще и кэш-памяти третьего уровня (L3). Обычно этот уровень является общим для всех ядер (или группы ядер). В Nehalem, где количество ядер в зависимости от модели может меняться от 2 до 8, емкость L3 составляет от 2 до 8 Мбайт, а в Phenom — 2 Мбайт.

При доступе к памяти ЦП сначала обращается к кэш-памяти первого уровня. В случае промаха производится обращение к кэш-памяти второго уровня и т. д. (вплоть до основной памяти).

Среднее время доступа T_{av} к одноуровневой кэш-памяти можно оценить как:

$$T_{av} = T_{\mathrm{L1}h} + K_{\mathrm{L1}m} T_{\mathrm{L1}m},$$

где $T_{\mathrm{L1}h}$ — время обращения при попадании; $T_{\mathrm{L1}m}$ — потери на промах; $K_{\mathrm{L1}m}$ — коэффициент промахов. Для двухуровневой кэш-памяти имеем:

$$T_{av} = T_{\mathrm{L1}h} + (K_{\mathrm{L1}m} T_{\mathrm{L2}h}) + (K_{\mathrm{L2}m} T_{\mathrm{L2}m}).$$

Потенциальная экономия за счет применения L2 зависит от вероятности попаданий как в L1, так и в L2. Ряд исследований показывает, что использование кэш-памяти второго уровня существенно улучшает производительность.

При построении иерархии кэш-памяти используется один из двух принципов — инклюзивный или эксклюзивный.

При *инклюзивном* подходе вся информация из верхних уровней иерархии содержится и в последующих нижних уровнях. Это означает, что, например, при замещении блока в L1 удаляемый блок переносится в L2. Так организована кэш-память L3 в микропроцессоре Nehalem.

В *эксклюзивной* схеме блок, удаляемый из кэш-памяти более высокого уровня, не дублируется в памяти следующего уровня. Вместо этого замещаемый и замещающий блоки взаимно меняются местами. Например, место блока из L2, замещающего какой-то блок в L1, занимает блок, удаленный при этом из L1. По такой схеме происходит взаимодействие уровня L3 с уровнем L2 в микропроцессоре Phenom.

Каждый из вариантов имеет свои достоинства. Так, инклюзивная схема потенциально обеспечивает более высокое быстродействие, а эксклюзивная — более эффективное использование емкости второго и последующих уровней кэш-памяти. На практике же сделанные утверждения достаточно условны, и однозначно отдать предпочтение той или иной схеме трудно.

В настоящее время достаточно серьезно обсуждается вопрос введения кэш-памяти четвертого уровня (L4).

Относительно эффективности многоуровневой кэш-памяти можно привести данные одного из многочисленных исследований. Так, если в системе с одноуровневой памятью количество обращений к кэш-памяти составляет 99%, а к основной памяти — 1%, то в двухуровневой памяти получаем: L1 — 99%, L2 — 0,0099% и ОП — 0,00001%. В случае трехуровневой кэш-памяти получаем: L1 — 99%, L2 — 0,0099%, L3 — 0,000099% и ОП — 0,0000001%.

Понятие виртуальной памяти

Для большинства типичных применений ВМ характерна ситуация, когда размещение всей программы в ОП невозможно из-за большого размера программы. Решение этой проблемы базируется на свойстве локальности по обращению, согласно которому в каждый момент времени «внимание» машины концентрируется на определенных сравнительно небольших участках программы. Таким образом, в ОП

достаточно хранить только используемые в данный период фрагменты программ, а остальные их части могут располагаться на внешних ЗУ (ВЗУ). При таком подходе, однако, необходимо принимать во внимание еще одну проблему.

Эта проблема связана с линейностью адресации памяти. Загружаемые в ОП программы или их фрагменты имеют различный размер, но линейную адресацию, то есть должны располагаться в непрерывной области памяти. Для занесения в ОП очередного фрагмента нужно освободить для него непрерывную область памяти, удалив другой фрагмент, но такой, размер которого равен или больше размера загружаемой части программы. Из-за различия в размерах фрагментов, как правило, приходится удалять больший по размеру фрагмент. В итоге свободной остается небольшая по размеру область памяти. При многократном повторе подобных действий в ОП остается множество свободных участков, недостаточных по размеру для загрузки полного непрерывного фрагмента программы. Возникает состояние, известное как *фрагментация памяти*.

Базисом для решения перечисленных проблем служит идея *виртуализации памяти*, под которой понимается такой метод автоматического управления иерархической памятью, при котором программисту кажется, что он имеет дело с единой памятью большой емкости и высокого быстродействия. Эту память называют виртуальной памятью (ВП). По своей сути виртуализация памяти представляет собой способ аппаратной и программной реализации концепции иерархической памяти.

В рамках идеи виртуализации памяти ОП рассматривается как линейное пространство N адресов, называемое *физическим пространством* памяти. Для задач, где требуется более чем N ячеек, предоставляется значительно большее пространство адресов (обычно равное общей емкости всех видов памяти), называемое *виртуальным пространством*, в общем случае не обязательно линейное. Адреса виртуального пространства называют *виртуальными*, а адреса физического пространства — *физическими*. Программа пишется в виртуальных адресах, но поскольку для ее выполнения нужно, чтобы обрабатываемые команды и данные находились в ОП, требуется, чтобы каждому виртуальному адресу соответствовал физический. Таким образом, в процессе вычислений необходимо, прежде всего, переписать из ВЗУ в ОП ту часть информации, на которую указывает виртуальный адрес (отобразить виртуальное пространство на физическое), после чего преобразовать виртуальный адрес в физический (рис. 6.34).

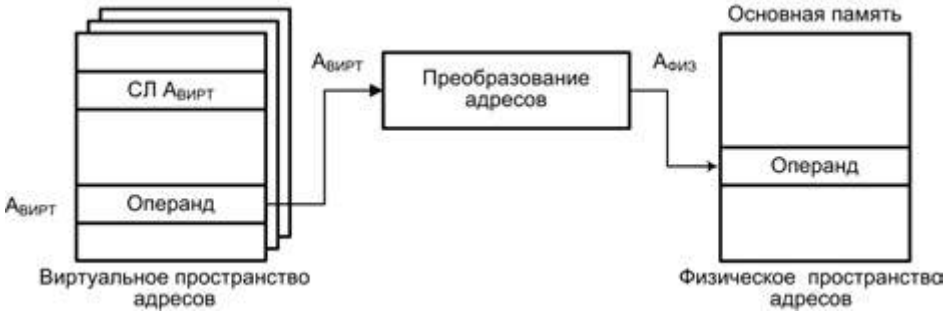


Рис. 6.34. Отображение виртуального адреса на физический

Среди систем виртуальной памяти можно выделить два класса: системы с фиксированным размером блоков (страничная организация) и системы с переменным размером блоков (сегментная организация). Оба варианта обычно совмещают (сегментно-страничная организация).

Страничная организация памяти

Целям преобразования виртуальных адресов в физические служит страничная организация памяти. Ее идея состоит в разбиении программы на части равной величины, называемые *страницами*. Размер страницы обычно выбирают в пределах 4–8 Кбайт, но так, чтобы он был кратен емкости одного сектора магнитного диска. Виртуальное и физическое адресные пространства разбиваются на блоки, размером в страницу. Блок основной памяти, соответствующий странице, часто называют *страничным кадром* или *фреймом* (page frame). Страницам виртуальной и физической памяти присваивают номера. Процесс доступа к данным по их виртуальному адресу иллюстрирует рис. 6.35.

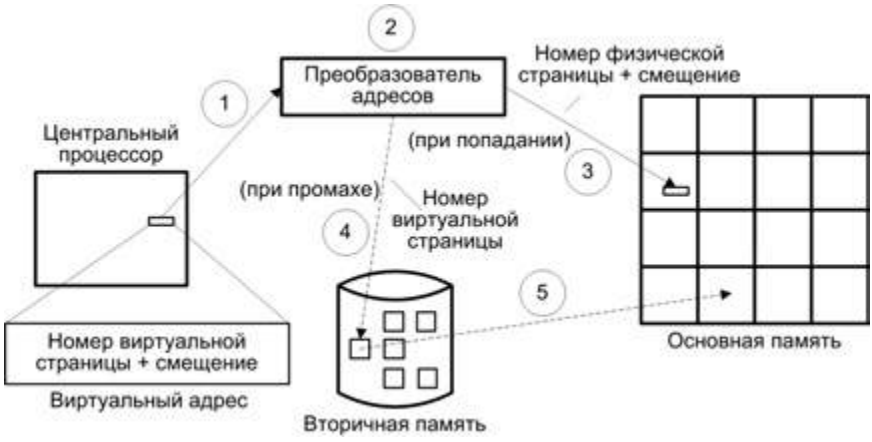


Рис. 6.35. Страничная организация виртуальной памяти

Центральный процессор обращается к ячейке, указав ее виртуальный адрес (1), состоящий из номера виртуальной страницы и смещения относительно ее начала. Этот адрес поступает в систему преобразования адресов (2) с целью получения из него физического адреса ячейки в основной памяти (3). Поскольку смещение в виртуальном и физическом адресе одинаковое, преобразованию подвергается лишь номер страницы. Если преобразователь обнаруживает, что нужная физическая страница отсутствует в основной памяти (произошел промах или страничный сбой), то нужная страница считывается из внешней памяти и заносится в ОП (4, 5).

Преобразователь адресов — это часть операционной системы, транслирующая номер виртуальной страницы в номер физической страницы, расположенной в основной памяти, а также аппаратура, обеспечивающая этот процесс и позволяющая ускорить его. Преобразование осуществляется с помощью так называемой *страничной таблицы*. При отсутствии нужной страницы в ОП преобразователь адресов

вырабатывает признак страничного сбоя, по которому операционная система приостанавливает вычисления, пока нужная страница не будет считана из вторичной памяти и помещена в основную.

Виртуальное пространство полностью описывается двумя таблицами (рис. 6.36): страничной таблицей и картой диска (будем считать, что вторичная память реализована на магнитных дисках). Таблица страниц определяет, какие виртуальные страницы находятся в основной памяти и в каких физических фреймах, а карта диска содержит информацию о секторах диска, где хранятся виртуальные страницы на диске.

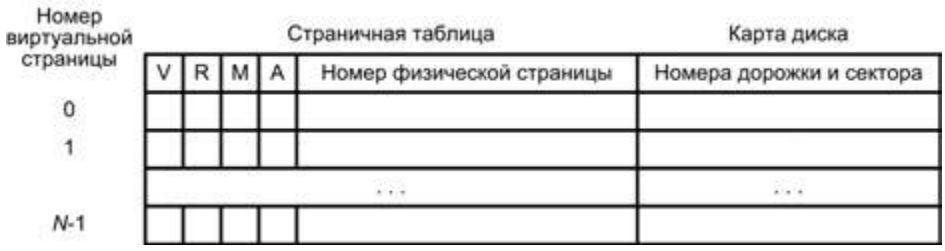


Рис. 6.36. Структура страничной таблицы и карты диска

Число записей в страничной таблице (СТ) в общем случае равно количеству виртуальных страниц. Каждая запись содержит поле номера физической страницы (НФС) и четыре признака: V, R, M и A.

Признак присутствия V устанавливается в единицу, если виртуальная страница в данный момент находится в основной памяти. В этом случае в поле номера физической страницы находится соответствующий номер. Если V = 0, то при попытке обратиться к данной виртуальной странице преобразователь адреса генерирует сигнал страничного сбоя (page fault), и операционная система предпринимает действия по загрузке страницы с диска в ОП, обращаясь для этого к карте диска. В карте указано, на какой дорожке и в каком секторе диска расположена каждая из виртуальных страниц[1]. Загрузка страницы с диска в ОП сопровождается записью в соответствующую строку страничной таблицы (указывается номер физической страницы, куда была загружена виртуальная страница).

В принципе, карта диска может быть совмещена со страничной таблицей путем добавления к последней еще одного поля. Другим вариантом может быть увеличение разрядности поля номера физической страницы и хранение в нем номеров дорожек и секторов для виртуальных страниц, отсутствующих в основной памяти. В этом случае вид хранимой информации будет определять признак V.

Признак использования страницы R устанавливается при обращении к данной странице. Эта информация используется в алгоритме замещения страниц для выбора той из них, которую можно наиболее безболезненно удалить из ОП, чтобы

[1] Множество страниц, присутствующих в данный момент в основной памяти, является подмножеством всех виртуальных страниц, хранящихся на магнитном диске.

освободить место для новой. Проблемы замещения информации в ОП решаются так же, как и для кэш-памяти.

Поскольку в ОП находятся лишь копии страниц, а их оригиналы хранятся на диске, необходимо обеспечить идентичность подлинников и копий. В ходе вычислений содержимое отдельных страниц может изменяться, что фиксируется путем установки в единицу *признака модификации* M. При удалении страницы из ОП проверяется состояние признака M. Если M = 1, то перед удалением страницы из основной памяти ее необходимо переписать на диск, а при M = 0 этого можно не делать.

Признак прав доступа A служит целям защиты информации и определяет, какой вид доступа к странице разрешен: только для чтения, только для записи, для чтения и для записи.

Когда программа загружается в ОП, она может быть направлена в любые свободные в данный момент страничные кадры, независимо от того, расположены ли они подряд или нет. Страничная организация позволяет сократить объем пересылок информации между внешней памятью и ОП, так как страницу не нужно загружать до тех пор, пока она действительно не понадобится.

Способ реализации СТ жизненно важен для эффективности техники виртуальной адресации, поскольку каждое обращение к памяти предполагает обращение к страничной таблице. Наиболее быстрый способ — хранение таблицы в специально выделенных для этого регистрах, но от него приходится отказываться при большом объеме СТ. Остается практически один вариант — выделение страничной таблице области основной памяти, несмотря на то что это приводит к двукратному увеличению времени доступа к информации. Чтобы сократить это время, в состав ВМ включают дополнительное ЗУ, называемое *буфером быстрого преобразования адреса* (TLB — Translation Look-aside Buffer), или *буфером ассоциативной трансляции*, или *буфером опережающей выборки* и представляющее собой кэш-память. При каждом преобразовании номера виртуальной страницы в номер физической страницы результат заносится в TLB: номер физической страницы в память данных, а виртуальной — в память тегов. Таким образом, в TLB попадают результаты нескольких последних операций трансляции адресов. При каждом обращении к ОП преобразователь адресов сначала производит поиск в памяти тегов TLB номера требуемой виртуальной страницы. При попадании адрес соответствующей физической страницы берется из памяти данных TLB. Если в TLB зафиксирован промах, то процедура преобразования адресов производится с помощью страничной таблицы, после чего осуществляется запись новой пары «номер виртуальной страницы — номер физический страницы» в TLB. Структура TLB показана на рис. 6.37.

Буфер преобразования адресов обычно реализуется в виде полностью ассоциативной или множественно-ассоциативной кэш-памяти с высокой степенью ассоциативности и временем доступа, сопоставимым с аналогичным показателем для кэш-памяти первого уровня (L1). Число входов у типовых TLB невелико (64–256). Так, TLB микропроцессора Pentium III имеет 64 входа при размере страницы 4 Кбайт, что позволяет получить быстрый доступ к адресному пространству в 256 Кбайт.

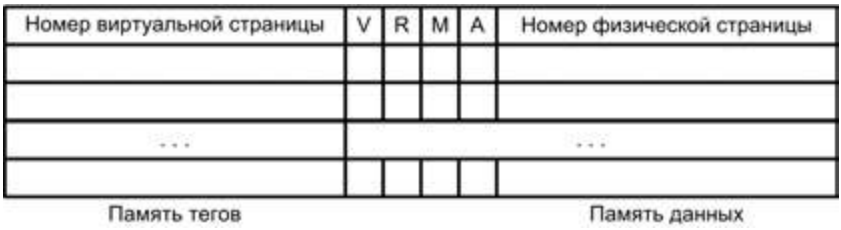


Рис. 6.37. Структура буфера быстрого преобразования адресов

Серьезной проблемой в системе виртуальной памяти является большой объем страничных таблиц, который пропорционален числу виртуальных страниц. Таблица занимает значительную часть ОП, а на поиск уходит много времени, что крайне нежелательно.

Один из способов сокращения длины таблиц основан на введении многоуровневой организации таблиц. В этом варианте информация оформляется в виде нескольких страничных таблиц сравнительно небольшого объема, которые образуют второй уровень. Первый уровень представлен таблицей с каталогом, где указано местоположение каждой из страничных таблиц (адрес начала таблицы в памяти) второго уровня. Сначала в каталоге определяется расположение нужной страничной таблицы и лишь затем производится обращение к нужной таблице. О достигаемом эффекте можно судить из следующего примера. Пусть адресная шина ВМ имеет ширину 32 бита, а размер страницы равен 4 Кбайт. Тогда количество виртуальных страниц, а следовательно, и число входов в единой страничной таблице составит 2^{20}. При двухуровневой организации можно обойтись одной страницей первого уровня на 1024 (2^{10}) входов и 1024 страничными таблицами на такое же число входов.

Другой подход называют *способом обращенных* или *инвертированных страничных таблиц*. Такую таблицу в каком-то смысле можно рассматривать как увеличенный эквивалент TLB, отличающийся тем, что она хранится в ОП и реализуется не аппаратурой, а программными средствами. Число входов в таблицу определяется емкостью ОП и равно числу страниц, которое может быть размещено в основной памяти. Одновременно с этим имеется и традиционная СТ, но хранится она не в ОП, а на диске. Для поиска нужной записи в инвертированной таблице используется хеширование, когда номер записи в таблице вычисляется в соответствии с некой хеш-функцией. Аргументом этой функции служит номер искомой виртуальной страницы. Хеширование позволяет ускорить операцию поиска. Если нужная страница в ОП отсутствует, производится обращение к основной таблице на диске, и после загрузки страницы в ОП корректируется и инвертированная таблица.

Сегментно-страничная организация памяти

При страничной организации предполагается, что виртуальная память — это непрерывный массив со сквозной нумерацией слов, что не всегда можно признать оптимальным. Обычно программа состоит из нескольких частей — кодовой, информационной и стековой. Так как заранее неизвестны длины этих составляющих, то удобно, чтобы при программировании каждая из них имела собственную

нумерацию слов, отсчитываемых с нуля. Для этого организуют систему *сегментированной памяти*, выделяя в виртуальном пространстве независимые линейные пространства переменной длины, называемые *сегментами*. Каждый сегмент представляет собой отдельную логическую единицу информации, содержащую совокупность данных или программный код и расположенную в адресном пространстве пользователя. В каждом сегменте устанавливается своя собственная нумерация слов, начиная с нуля. Виртуальная память также разбивается на сегменты, с независимой адресацией слов внутри сегмента. Каждой составляющей программы выделяется сегмент памяти. Виртуальный адрес определяется номером сегмента и адресом внутри сегмента. Для преобразования виртуального адреса в физический используется специальная *сегментная таблица*.

Недостатком такого подхода является то, что неодинаковый размер сегментов приводит к неэффективному использованию ОП. Так, если ОП заполнена, то при замещении одного из сегментов требуется вытеснить такой, размер которого равен или больше размера нового. При многократном повторе подобных действий в ОП остается множество свободных участков, недостаточных по размеру для загрузки полного сегмента. Решением проблемы служит *сегментно-страничная организация памяти*. В ней размер сегмента выбирается не произвольно, а задается кратным размеру страницы. Сегмент может содержать то или иное, но обязательно целое число страниц, даже если одна из страниц заполнена частично. Возникает определенная иерархия в организации доступа к данным, состоящая из трех ступеней: сегмент → страница → слово. Этой структуре соответствует иерархия таблиц, служащих для перевода виртуальных адресов в физические.

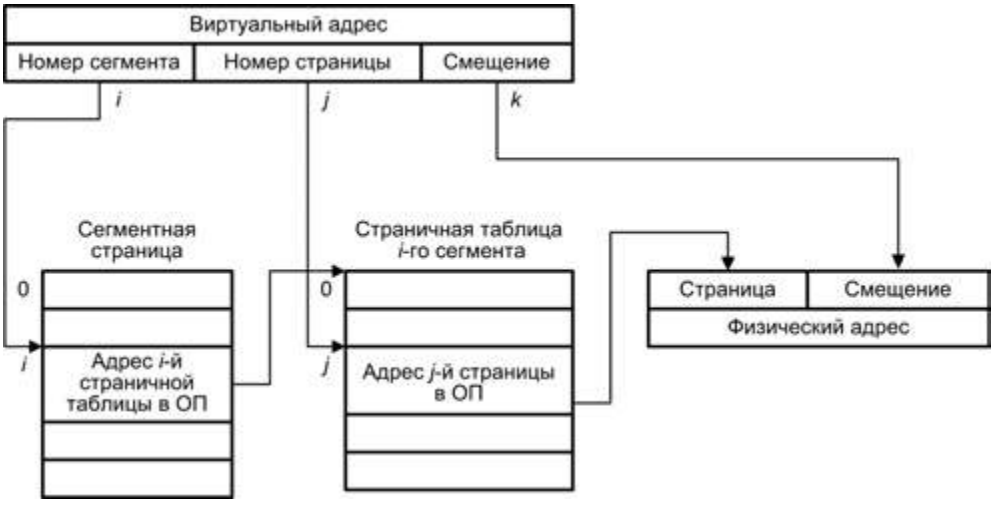


Рис. 6.38. Преобразование адреса при сегментно-страничной организации памяти

В сегментной таблице программы перечисляются все сегменты данной программы с указанием начальных адресов СТ, относящихся к каждому сегменту. Количество страничных таблиц равно числу сегментов и любая из них определяет

расположение каждой из страниц сегмента в памяти, которые могут располагаться не подряд — часть страниц может находиться в ОП, остальные — во внешней памяти. Структуру виртуального адреса и процесс преобразования его в физический адрес иллюстрирует рис. 6.38.

Для получения физического адреса необходим доступ к сегментной и одной из страничных таблиц, поэтому преобразование адреса может занимать много времени.

Организация защиты памяти

Современные вычислительные машины, как правило, работают в многопользовательском и многозадачном режимах, когда в основной памяти одновременно находятся программы, относящиеся как к разным пользователям, так и к различным задачам одного пользователя. Если даже ВМ выполняет только одну программу, то и в этом случае в ОП, помимо этой программы и относящихся к ней данных, всегда присутствуют фрагменты операционной системы. Каждой задаче в основной памяти выделяется свое адресное пространство. Такие пространства, если только это специально не предусмотрено, обычно независимы. В то же время в программах могут содержаться ошибки, приводящие к вторжению в адресное пространство других задач. Следствием этих ошибок может стать искажение информации, принадлежащей другим программам. Следовательно, в ВМ обязательно должны быть предусмотрены меры, предотвращающие несанкционированное воздействие программ одного пользователя на работу программ других пользователей и на операционную систему. Особенно опасны последствия таких ошибок при нарушении адресного пространства операционной системы.

Чтобы воспрепятствовать разрушению одних программ другими, достаточно защитить область памяти данной программы от попыток записи в нее со стороны других программ (защита от записи). В ряде случаев необходимо иметь возможность защиты и от чтения со стороны других программ, например при ограничениях на доступ к системной информации.

Защита от вторжения программ в чужие адресные пространства реализуется различными средствами и способами, но в любом варианте к системе защиты предъявляются два требования: ее реализация не должна заметно снижать производительность ВМ и требовать слишком больших аппаратных затрат.

Задача обычно решается за счет сочетания программных и аппаратных средств, хотя ответственность за охрану адресных пространств от несанкционированного доступа обычно возлагается на операционную систему. Наибольшее распространение в архитектуре ВМ получили следующие классические методы защиты: защита отдельных ячеек, метод граничных регистров и метод ключей защиты.

Защита отдельных ячеек памяти

Этим видом защиты обычно пользуются при отладке новых программ параллельно с функционированием других программ. Реализовать подобный режим можно за счет выделения в каждой ячейке памяти специального «разряда защиты» и связывания его со схемой управления записью в память. Установка этого разряда в 1 блокирует запись в данную ячейку. Подобный режим использовался в вычислительных

машинах предыдущих поколений (для современных ВМ он не типичен). Основной недостаток метода состоит в избыточности кодирования информации из-за малой величины защищаемого объекта (ячейки). На практике целесообразней осуществлять защиту на уровне более крупных единиц — блоков ячеек.

Метод граничных регистров

Данный вид защиты наиболее распространен. Метод предполагает наличие в процессоре двух *граничных регистров*, содержимое которых определяет нижнюю и верхнюю границы области памяти, куда программа имеет право доступа. Заполнение граничных регистров производится операционной системой при загрузке программы. Метод иллюстрирует рис. 6.39.

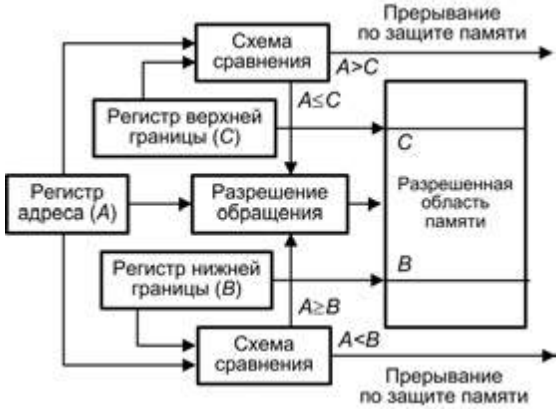


Рис. 6.39. Защита памяти методом граничных регистров

При каждом обращении к памяти проверяется, попадает ли используемый адрес в установленные границы. Такую проверку, например, можно организовать на этапе преобразования виртуального адреса в физический. При нарушении границы доступ к памяти блокируется, и формируется запрос прерывания, вызывающий соответствующую процедуру операционной системы. Нижнюю границу разрешенной области памяти определяет сегментный регистр. Верхняя граница подсчитывается операционной системой в соответствии с размером размещаемого в ОП сегмента.

Возможен и модифицированный вариант рассматриваемого метода, в котором один из регистров, как и ранее, содержит адрес нижней границы защищаемой области памяти, а второй — длину этой области.

В рассмотренной схеме необходимо, чтобы в ВМ поддерживались два режима работы: привилегированный и пользовательский. Запись информации в граничные регистры возможна лишь в привилегированном режиме.

Основной недостаток метода в том, что поддерживается работа лишь с непрерывными областями памяти.

Метод ключей защиты

Метод позволяет организовать защиту несмежных областей памяти. Память условно делится на блоки одинакового размера. Каждому блоку ставится в соответствие некоторый код, называемый *ключом защиты памяти*. Каждой программе, в свою очередь, присваивается *код защиты программы*. Условием доступа программы к конкретному блоку памяти служит совпадение ключей защиты памяти и программы либо равенство одного из этих ключей нулю. Нулевое значение ключа защиты программы разрешает доступ ко всему адресному пространству и используется только программами операционной системы. Распределением ключей защиты программы ведает операционная система.

Ключ защиты программы обычно представлен в виде отдельного поля *слова состояния программы*, хранящегося в специальном регистре. Ключи защиты памяти хранятся в специальной памяти, более быстродействующей, чем основная память. При каждом обращении к ОП специальная комбинационная схема производит сравнение ключей защиты памяти и программы. При совпадении доступ к памяти разрешается. Действия в случае несовпадения ключей зависят от того, какой вид доступа запрещен: при записи, при чтении или в обоих случаях. Для этого в ключе защиты памяти имеется дополнительный разряд режима обращения (РО). Нулевое значение РО определяет необходимость защиты только при записи. Единичное значение РО определяет защиту при любом виде обращения к памяти. Если выяснилось, что данный вид доступа запрещен, то, так же как и в методе граничных регистров, формируется запрос прерывания и вызывается соответствующая процедура операционной системы.

Механизм этого метода защиты поясняет рис. 6.40.

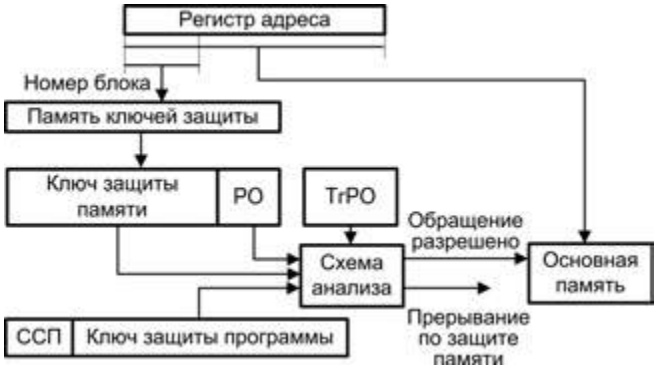


Рис. 6.40. Защита памяти методом ключей защиты

При обращении к памяти старшие разряды адреса используются как адрес нужного ключа в памяти ключей защиты памяти. Схема анализа сравнивает ключ защиты блока памяти и ключ программы, хранящийся в регистре слова состояния программы (ССП). Результаты сравнения анализируются схемой анализа, которая учитывает также режим обращения к ОП, указываемый триггером режима обращения

(ТгРО), и режим защиты РО в ключе защиты памяти. По результатам анализа вырабатывается один из двух сигналов: «Обращение разрешено» или «Прерывание по защите памяти».

Кольца защиты

Защиту адресного пространства операционной системы от несанкционированного вторжения со стороны пользовательских программ обычно организуют за счет аппаратно-реализованного разделения программ по *уровням привилегий*. Процессор постоянно контролирует, имеет ли выполняемая программа достаточный уровень привилегий, чтобы:

- выполнять команду данного типа;
- выполнять команду ввода/вывода на данном внешнем устройстве;
- обращаться к данным других программ;
- вызывать другие программы.

Этот вариант защиты напоминает метод ключей защиты и характерен он для персональных ВМ. Предусматривается, как минимум, два режима работы процессора: системный (режим супервизора — «надзирателя») и пользовательский. Такую структуру принято называть *кольцами защиты* и изображать в виде концентрических окружностей, где пользовательский режим представлен внешним кольцом, а системный — внутренней окружностью (рис. 6.41). В системном режиме программе доступны все ресурсы ВМ, а возможности пользовательского режима существенно ограничены. Переключение из пользовательского режима в системный осуществляется специальной командой. В большинстве современных ВМ число уровней привилегий (колец защиты) увеличено. Так, аппаратура современных процессоров различает 4 уровня привилегий (их обычно нумеруют от 0 до 3). Наиболее привилегированным считается уровень 0. Распределение программ по уровням привилегий зависит от значимости этих программ. Как правило, программы операционной системы (ОС) и программы пользователя распределяют по уровням привилегий следующим образом:

- **уровень 0** — ядро (ОС), включающее программы, без которых работа других программ была бы невозможна;
- **уровень 1** — основная часть программ ОС (утилиты);
- **уровень 2** — драйверы устройств ввода/вывода, системы управления базами данных, системы программирования и т. п.;
- **уровень 3** — прикладные программы пользователя.

С увеличением номера уровня привилегий возрастает и число программ, которые могут выполняться на данном уровне, то есть наибольшее число программ выполняется на уровне 3.

Отметим, что операционные системы не обязательно поддерживают все 4 уровня привилегий и, как правило, ограничиваются лишь двумя уровнями, то есть работают с двумя кольцами защиты: кольцом супервизора, охватывающем программы ОС (аппаратный уровень 0), и кольцом пользователя (аппаратные уровни 1, 2, 3). Некоторые ОС используют три кольца защиты.

Рис. 6.41. Кольца защиты

Внешняя память

Важным звеном в иерархии запоминающих устройств является внешняя (вторичная и третичная) память, реализуемая на базе различных ЗУ. Наиболее распространенные виды таких ЗУ — это твердотельные, магнитные и оптические диски, а также магнитоленточные устройства.

Характеристики ЗУ внешней памяти

Ранее были рассмотрены основные характеристики ЗУ внутренней памяти. Многие из них применимы и к ЗУ внешней памяти, иные — модифицируются для учета их специфики. Обсудим следующий перечень характеристик ЗУ внешней памяти:

- место расположения;
- емкость;
- единица пересылки;
- метод доступа;
- быстродействие;
- физический тип;
- физические особенности;
- стоимость.

Место расположения. Внешние ЗУ обычно выполнены в виде отдельных конструктивов или встраиваемых блоков. В последнем случае они монтируются в специально отведенных для них местах корпуса ВМ. К ядру ВМ внешние ЗУ подключаются аналогично устройствам ввода/вывода.

Емкость ЗУ внешней памяти достаточно велика, поэтому ее характеризуют такими крупными единицами, как мегабайт, гигабайт, терабайт и петабайт.

Важной характеристикой ЗУ является *единица пересылки*. Применительно к внешней памяти, данные часто передаются единицами, превышающими размер слова, и такие единицы называются *блоками*.

При оценке быстродействия необходимо учитывать применяемый в данном типе *ЗУ метод доступа* к данным. В ЗУ внешней памяти используется два основных метода доступа:

- *Последовательный доступ* (позволяет обращаться к ячейкам ЗУ в определенной последовательности). ЗУ с последовательным доступом ориентировано на хранение информации в виде последовательности блоков данных, называемых записями. Для доступа к нужному элементу (слову или байту) необходимо прочитать все предшествующие ему данные. Время доступа зависит от положения требуемой записи в последовательности записей на носителе информации и позиции элемента внутри данной записи. Примером может служить ЗУ на магнитной ленте.
- *Прямой доступ*. Каждая запись имеет уникальный адрес, отражающий ее физическое размещение на носителе информации. Обращение осуществляется по адресу к началу записи, с последующим последовательным доступом к определенной единице информации внутри записи. В результате время доступа к определенной позиции является величиной переменной. Такой режим характерен для магнитных дисков.

Быстродействие ЗУ внешней памяти обычно оценивают с помощью двух дополнительных параметров:

- *Время доступа* ($T_Д$). В ЗУ с подвижным носителем информации — это время, затрачиваемое на установку головки записи/считывания (или носителя) в нужную позицию.
- *Среднее время считывания или записи N битов* T_N. Для ЗУ внешней памяти определяется соотношением $T_N = T_A + \frac{N}{R}$, где T_A — среднее время доступа; R — скорость пересылки в бит/с.

Говоря о *физическом типе* запоминающего устройства, необходимо упомянуть три наиболее распространенных технологии внешних ЗУ — это полупроводниковая память, лежащая в основе твердотельных дисков, память с магнитным носителем информации, используемая в магнитных дисках и лентах, и память с оптическим носителем — оптические диски.

Из *физических особенностей* внешних ЗУ важнейшее значение имеет их энергонезависимость. Магнитная и оптическая память — энергонезависимы. Полупроводниковая память может быть как энергозависимой, так и нет, в зависимости от ее типа. В твердотельных дисках, в частности, используется оба варианта.

Стоимость ЗУ, как и в случае внутренней памяти, принято оценивать отношением общей стоимости ЗУ к его емкости в битах, то есть стоимостью хранения одного бита информации.

Запоминающие устройства на основе магнитных дисков

Информация в ЗУ на магнитных дисках (ЗУМД)[1] хранится на плоских дисках из немагнитного материала, покрытых намагничивающимся материалом. Традици-

[1] Рассматриваются принципы организации накопителей на жестких магнитных дисках, поскольку гибкие магнитные диски в последнее время утратили свою актуальность.

онно подложка выполнялась из алюминия или его сплавов. В последнее время стали применяться подложки из стекла, позволяющие улучшить ряд характеристик ЗУМД.

Данные записываются и считываются с диска с помощью электромагнитной катушки, называемой *головкой считывания/записи*, которая в процессе считывания и записи неподвижна, в то время как диск вращается относительно нее. В результате данные образуют окружности на поверхности диска, называемые *дорожками*. Запись основана на том, что ток, проходящий через катушку, создает магнитное поле. При записи на головку подаются электрические импульсы, намагничивающие участок поверхности под ней. Характер намагниченности поверхности различен в зависимости от направления тока в катушке. Одна полярность соответствует логической единице, а другая — логическому нулю. Считывание базируется на электрическом токе, наводимом в катушке головки, под воздействием перемещающегося относительно нее магнитного поля. Когда под головкой проходит участок поверхности диска, в катушке наводится ток той же полярности, что использовался для записи информации.

Механизмы чтения и записи

В большинстве систем имеются две независимые головки: головка считывания и головка записи.

В ЗУМД, где вектор намагниченности расположен в плоскости диска, головка записи имеет вид прямоугольной катушки с прорезью со стороны магнитного диска и несколькими витками провода с противоположной стороны. Электрический ток в проводе создает магнитное поле в прорези, которое, в свою очередь, намагничивает маленький участок на поверхности диска (битовую ячейку). Изменение направления тока меняет на противоположное намагничивание магнитной поверхности.

В последнее время для увеличения плотности записи используют технологию перпендикулярной записи, отличие которой от описанного выше способа продольной записи состоит в том, что векторы намагниченности битовых ячеек располагаются не в плоскости диска, а перпендикулярно его поверхности. Перпендикулярная запись выполняется не кольцевой головкой, а специальной G-образной, геометрические свойства которой способствуют повышению максимального значения намагничивания в два раза.

Структура головки считывания практически такая же, что и у головки записи, поэтому одна и та же головка может быть использована для обеих операций. Такие совмещенные головки применяются в накопителях на гибких магнитных дисках, а ранее — также в системах с жесткими дисками. Современные системы жестких дисков используют разные механизмы считывания, требующие отдельной головки считывания, расположенной по возможности ближе к головке записи. Головка считывания состоит из магниторезистивного материала, электрическое сопротивление которого зависит от направления намагниченности среды, движущейся под головкой. Изменения сопротивления фиксируются как изменение напряжения, вызванное пропусканием тока через сенсор с изменяющимся сопротивлением.

Характеристики дисковых систем

Положение головок относительно поверхности дисков может быть фиксированным или изменяющимся. В ЗУ с *фиксированными головками* на каждую дорожку приходится по одной головке считывания/записи. Головки смонтированы на жестком рычаге, пересекающем все дорожки диска (рис. 6.42, *а*). В дисковом ЗУ с *подвижными головками* имеется только одна головка, также установленная на рычаге (рис. 6.42, *б*), однако рычаг способен перемещаться в радиальном направлении над поверхностью диска, обеспечивая возможность позиционирования головки на любую дорожку.

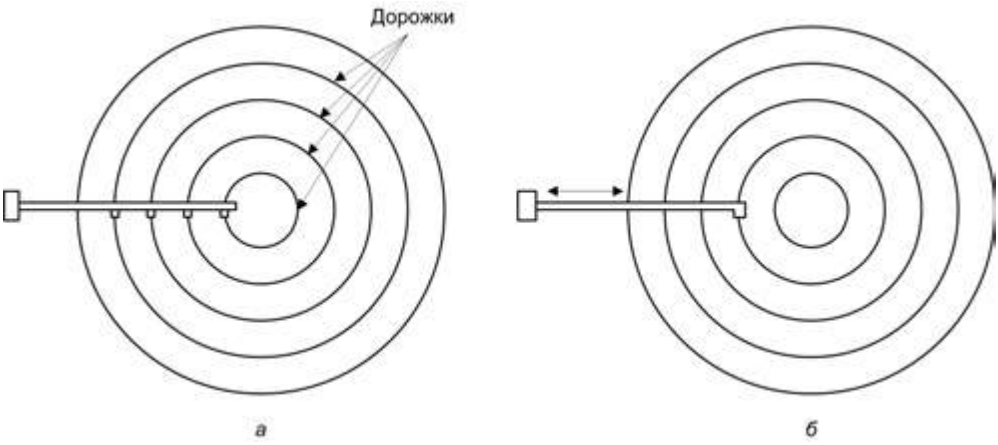


Рис. 6.42. Варианты организации дисков: *а* — с фиксированными головками; *б* — с подвижной головкой

Диски с магнитным носителем устанавливаются в дисковод, состоящий из рычага, шпинделя, вращающего диск, и электронных схем, требуемых для ввода и вывода двоичных данных. Диски могут быть несъемными либо съемными. *Несъемный диск* зафиксирован на дисководе. *Съемный диск* может быть вынут из дисковода и заменен на другой аналогичный диск. Преимущество системы со съемными дисками — возможность хранения неограниченного количества данных при ограниченном числе дисковых устройств. Кроме того, такой диск может быть перенесен с одной ВМ на другую.

На оси может располагаться один (рис. 6.43, *а*) или несколько (рис. 6.43, *б*) дисков. В последнем случае используют термин *дисковый пакет*. В современных накопителях обычно устанавливается несколько дисков, и данные записываются на обеих сторонах каждого из них. В большинстве накопителей есть по меньшей мере два или три диска (что позволяет выполнять запись на четырех или шести сторонах), но существуют также устройства, содержащие до 11 и более дисков. Однотипные (одинаково расположенные) дорожки на всех сторонах дисков объединяются в *цилиндр*. Для каждой стороны диска предусмотрена своя дорожка чтения/записи, но при этом все головки смонтированы на общем стержне, или стойке. Поэтому головки не могут перемещаться независимо друг от друга и двигаются только синхронно.

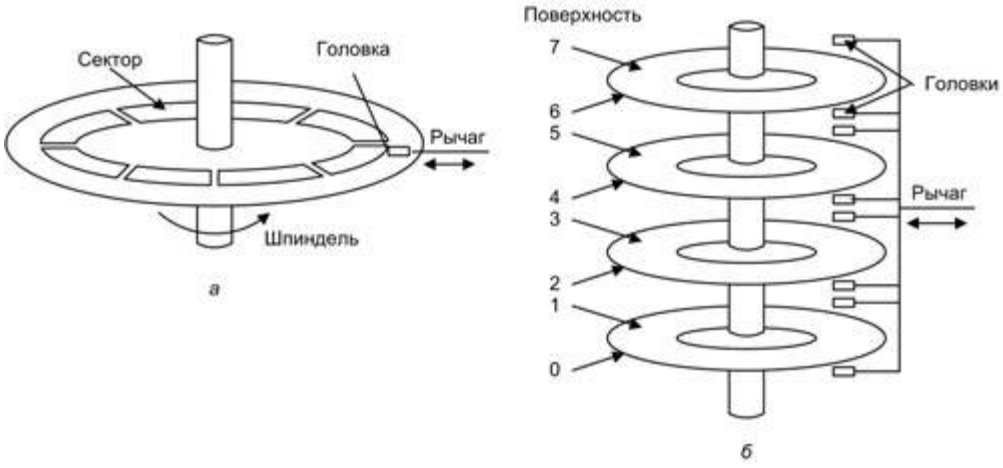


Рис. 6.43. Дисковое ЗУ: *а* — с одной пластиной; *б* — с дисковым пакетом

Жесткие диски вращаются достаточно быстро. Частота их вращения даже в большинстве первых моделей составляла 3600 об/мин (то есть в 10 раз больше, чем в накопителе на гибких дисках). В настоящее время частота вращения жестких дисков возросла до 6400, 7200, 10 000 об/мин и даже 15 000 об/мин.

В зависимости от применяемой головки считывания/записи можно выделить три типа дисковых ЗУ. В первом варианте головка устанавливается на фиксированной дистанции от поверхности так, чтобы между ними оставался воздушный промежуток.

Второй вариант — это когда в процессе чтения и записи головка и диск находятся в физическом контакте. Такой способ используется, например, в накопителях на гибких магнитных дисках (дискетах).

Для правильной записи и считывания головка должна генерировать и воспринимать магнитное поле определенной интенсивности, поэтому чем уже головка, тем ближе должна она размещаться к поверхности диска (более узкая головка означает и более узкие дорожки, а значит, и большую емкость диска). Однако приближение головки к диску означает и больший риск ошибки за счет загрязнения и дефектов. В процессе решения этой проблемы был создан диск типа *винчестер*. Головки винчестера и диски помещены в герметичный корпус, защищающий их от загрязнения. Кроме того, головки имеют очень малый вес и аэродинамическую форму. Они спроектированы для работы значительно ближе к поверхности диска, чем головки в обычных жестких дисках, тем самым повышается плотность хранения данных. При остановленном диске головка прилегает к его поверхности. Давления, возникающего при вращении диска, достаточно для подъема головки над поверхностью. В результате создается неконтактная система с узкими головками считывания/записи, обеспечивающая более плотное прилегание головки к поверхности диска, чем в обычных жестких дисках.

Организация данных и форматирование

Данные на диске организованы в виде набора концентрических окружностей, называемых *дорожками* (рис. 6.44). На поверхности диска располагаются тысячи дорожек. Каждая из них имеет ту же ширину, что и головка. Соседние дорожки разделены промежутками. Это предотвращает ошибки из-за смещения головки или из-за интерференции магнитных полей. Как правило, для упрощения электроники принимается, что на всех дорожках может храниться одинаковое количество информации. Таким образом, плотность записи увеличивается от внешних дорожек к внутренним.

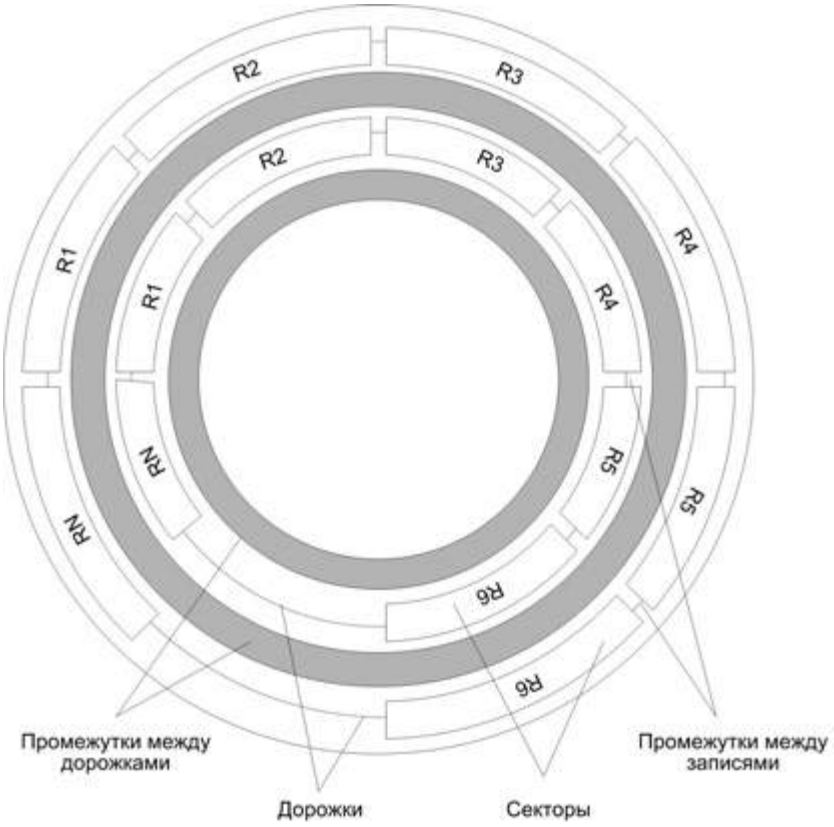


Рис. 6.44. Порядок размещения информации на магнитном диске

Дорожка записи на диске слишком велика, чтобы использовать ее в качестве единицы хранения информации. Во многих накопителях ее емкость превышает 100 тыс. байтов, и отводить такой блок для хранения небольшого файла крайне расточительно. Поэтому дорожки на диске разбивают на нумерованные отрезки, называемые секторами. Обмен информацией с ЗУМД осуществляется *секторами*. Количество секторов может быть разным в зависимости от плотности дорожек и типа накопителя. Например, дорожка жесткого диска может содержать от 380 до 700 секторов.

Секторы, создаваемые с помощью стандартных программ форматирования, имеют емкость 512 байтов, но не исключено, что в будущем эта величина изменится. Соседние секторы на дорожке разделены между собой межсекторными промежутками. Нумерация секторов на дорожке начинается с единицы, в отличие от головок и цилиндров, отсчет которых ведется с нуля.

Бит, находящийся вблизи центра вращающегося диска, проходит под головкой считывания/записи медленней, чем бит, расположенный на периферии диска. По этой причине нужен какой-либо механизм для компенсации этого различия в скоростях, с тем чтобы головка могла читать все биты с одинаковой скоростью. Это может быть обеспечено путем увеличения расстояния между битами информации, записываемой в сегментах диска. Информация затем может сканироваться с той же скоростью путем вращения диска с фиксированной скоростью, известной как постоянная угловая скорость (ПУС). На рис. 6.45, *а* показана организация диска, использующего ПУС. Диск поделен на несколько секторов и на ряд концентрических дорожек. Преимущество использования ПУС в том, что к индивидуальным блокам данных можно прямо адресоваться, указав дорожку и сектор. Перемещение головки от текущего положения к определенному адресу требует лишь короткого движения головки к нужной дорожке и короткого ожидания, когда под ней окажется нужный сектор. Недостаток систем с ПУС в том, что количество данных, которое может храниться на длинных внешних дорожках, то же самое, что и на коротких внутренних дорожках.

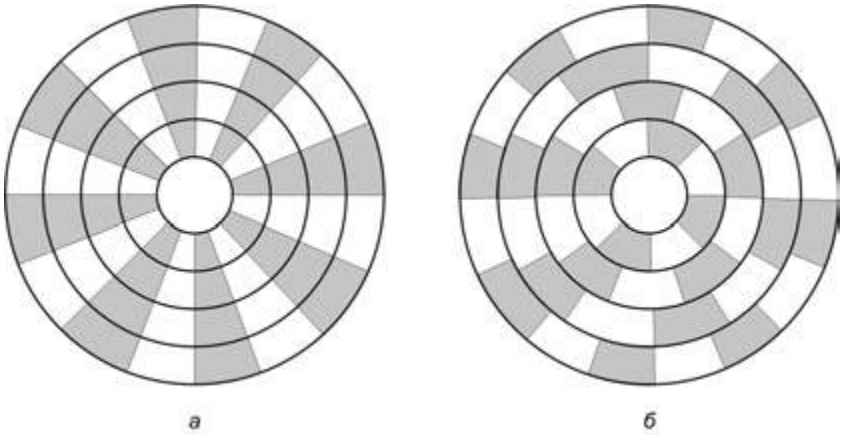


Рис. 6.45. Сравнение методов организации диска: *а* — с постоянной угловой скоростью; *б* — с многозонной записью

Ввиду того, что плотность информации возрастает по мере приближения к внутренним дорожкам, емкость диска ограничивается максимальной плотностью записи на самой внутренней дорожке. Для увеличения плотности в современных дисковых системах используется техника, известная как *многозонная запись*, где поверхность разбивается на несколько зон (обычно 16). Внутри зоны число битов на дорожку постоянно. Зоны, более удаленные от центра, содержат больше битов (больше

секторов), чем зоны, расположенные ближе к центру вращения. Это позволяет достичь большей емкости за счет более сложной аппаратной части. По мере движения головки диска от одной зоны к другой длина индивидуальных битов меняется, вызывая изменения во временных соотношениях чтения и записи. На рис. 6.45, *б* показана сущность многозонной записи: на рисунке каждая зона имеет ширину только одной дорожки.

При такой организации ЗУМД должны быть заданы: точка отсчета секторов и способ определения начала и конца каждого сектора. Все это обеспечивается с помощью *форматирования*, в ходе которого на диск заносится служебная информация, недоступная пользователю и используемая только аппаратурой дискового ЗУ.

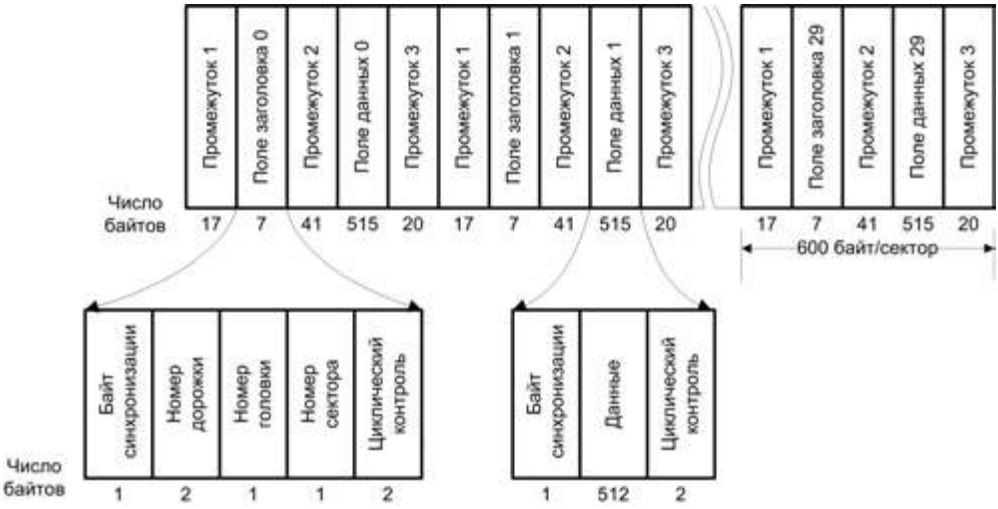


Рис. 6.46. Формат дорожки диска типа «Винчестер» (Seagate ST506)

Пример разметки МД показан на рис. 6.46. Здесь каждая дорожка включает в себя 30 секторов по 600 байтов в каждом. Сектор хранит 512 байтов данных и управляющую информацию, нужную для контроллера диска. Поле заголовка содержит информацию, служащую для идентификации сектора. Байт синхронизации представляет собой характерную двоичную комбинацию, позволяющую определить начало поля. Номер дорожки определяет дорожку на поверхности. Если в накопителе используются несколько дисков, то номер головки определяет нужную поверхность. Поле заголовка и поле данных содержат также код циклического контроля, позволяющий обнаружить ошибки. Обычно этот код формируется последовательным сложением по модулю 2 всех байтов, хранящихся в поле.

Массивы магнитных дисков с избыточностью

Несмотря на технологические успехи в области внешних ЗУ, построенных на самых различных физических принципах, подсистема памяти на базе магнитных дисков по-прежнему остается основой внешней памяти любой ВМ. По этой причине дисковой подсистеме предъявляются самые высокие требования в плане про-

изводительности и отказоустойчивости, и уже с самого начала использования подсистем на базе ЗУМД не прекращаются попытки улучшить их характеристики. Одно из наиболее интересных и универсальных усовершенствований было предложено в 1987 году учеными университета Беркли (Калифорния) [129]. Проект известен под аббревиатурой RAID (Redundant Array of Independent (or Inexpensive) Disks) — *массив независимых* (*или недорогих*) *дисков с избыточностью*. В основе концепции RAID лежит переход от одного физического магнитного диска (МД) большой емкости к массиву независимо и параллельно работающих физических дисковых ЗУ, рассматриваемых операционной системой как одно большое логическое дисковое запоминающее устройство. Такой подход позволяет повысить производительность дисковой памяти за счет возможности параллельного обслуживания запросов на считывание и запись, при условии, что данные находятся на разных дисках. Повышенная надежность достигается тем, что в массиве дисков хранится избыточная информация, позволяющая обнаружить и исправить возможные ошибки. На период, когда концепция RAID была впервые предложена, определенный выигрыш достигался и в плане стоимости. В настоящее время, с развитием технологии производства МД, утверждение об экономичности массивов RAID становится проблематичным, что, однако, вполне компенсируется их повышенными быстродействием и отказоустойчивостью.

В настоящее время производители RAID-систем, объединившиеся в ассоциацию RAB (RAID Advisory Board), договорились о единой классификации RAID, включающей в себя восемь уровней. Известны также еще несколько схем RAID, не включенных в эту классификацию, поскольку по сути они представляют собой различные комбинации стандартных уровней. Хотя ни одна из схем массива МД не может быть признана идеальной для всех случаев, каждая из них позволяет существенно улучшить какой-то из показателей (производительность, отказоустойчивость) либо добиться наиболее подходящего сочетания этих показателей. Для всех уровней RAID характерны три общих свойства:

- RAID представляет собой набор физических дисковых ЗУ, управляемых операционной системой и рассматриваемых как один логический диск;
- данные распределены по физическим дискам массива;
- избыточное дисковое пространство используется для хранения дополнительной информации, гарантирующей восстановление данных в случае отказа диска.

Повышение производительности дисковой подсистемы

Повышение производительности дисковой подсистемы в RAID достигается с помощью приема, называемого *расслоением* или *расщеплением* (striping). В его основе лежит разбиение данных и дискового пространства на сегменты, так называемые *полосы* (strip — узкая полоса). Полосы распределяются по различным дискам массива, в соответствии с определенной системой. Это позволяет производить параллельное считывание или запись сразу нескольких полос, если они расположены на разных дисках. В идеальном случае производительность дисковой подсистемы может быть увеличена в число раз, равное количеству дисков в массиве. Размер полосы выбирается исходя из особенностей каждого уровня RAID и может быть равен биту, байту, размеру физического сектора МД (обычно 512 байтов) или размеру дорожки.

Чаще всего логически последовательные полосы распределяются по последовательности дисков массива. Так, в *n*-дисковом массиве *n* первых логических полос физически расположены как первые полосы на каждом из *n* дисков, следующие *n* полос — как вторые полосы на каждом физическом диске и т. д. Набор логически последовательных полос, одинаково расположенных на каждом ЗУ массива, называют *лентой* (stripe — широкая полоса).

Повышение отказоустойчивости дисковой подсистемы

Одной из целей концепции RAID была возможность обнаружения и коррекции ошибок, возникающих при отказах дисков или в результате сбоев. Достигается это за счет избыточного дискового пространства для хранения дополнительной информации, позволяющей восстановить искаженные или утерянные данные. В RAID предусмотрены три возможности:

- дублирование (зеркалирование);
- код Хэмминга;
- биты паритета.

Первая возможность заключается в дублировании всех данных, при условии, что экземпляры одних и тех же данных расположены на разных дисках массива. Это позволяет при отказе одного из дисков воспользоваться соответствующей информацией, хранящейся на исправных ЗУМД. В принципе, распределение информации по дискам массива может быть произвольным, но для сокращения издержек, связанных с поиском копии, обычно применяется разбиение массива на пары ЗУМД, где в каждой паре дисков информация идентична и одинаково расположена. Для управления парой дисков может использоваться общий или раздельные контроллеры. Избыточность дискового массива здесь составляет 100%.

Вторая возможность основана на вычислении кода Хэмминга для каждой группы полос, одинаково расположенных на всех дисках массива (для каждой ленты). Корректирующие биты хранятся на специально выделенных для этой цели дополнительных дисках (по одному диску на каждый бит). Так, для массива из десяти МД требуются четыре таких дополнительных диска, и избыточность в данном случае близка к 30%.

В третьем случае вместо кода Хэмминга для каждого набора полос, расположенных в идентичной позиции на всех дисках массива, вычисляется контрольная полоса, состоящая из битов паритета. В ней значение отдельного бита формируется как сумма по модулю два для одноименных битов во всех контролируемых полосах. Для хранения полос паритета требуется только один дополнительный диск. В случае отказа какого-либо из дисков массива производится обращение к диску паритета, и данные восстанавливаются по битам паритета и данным от остальных дисков массива. Избыточность при таком способе в среднем близка к 20%.

RAID 0: дисковый массив без отказоустойчивости

Схема RAID 0 обеспечивает максимально возможную производительность дисковой подсистемы. При распределении информации по дискам массива используется техника расщепления. В качестве полос могут выступать физические блоки, секторы или какие-то иные информационные единицы, размер которых не менее

размера физического сектора МД. Полосы распределены по последовательным дискам массива по циклической схеме (рис. 6.47). Благодаря технике расщепления запись или чтение нескольких логически последовательных полос (вплоть до n) может производиться параллельно, поскольку они располагаются на разных дисках. Это и обеспечивает существенное снижение общего времени ввода/вывода.

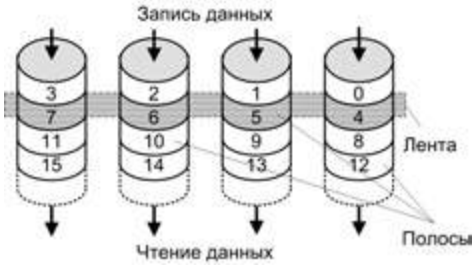


Рис. 6.47. Схема RAID 0

В схеме отсутствует избыточность (не предусматривается хранение дополнительной информации для контроля и исправления ошибок), то есть пользователю предоставляется весь объем дискового массива. Однако выход из строя одного из физических дисков массива приводит к невосстановимой потере хранимых данных. По этой причине схема RAID 0 применима лишь там, где производительность и емкость дисковой системы намного важнее частичной потери данных, например при хранении фото- и видеоизображений. Из-за отсутствия в RAID 0 средств по защите данных рекомендуется использовать в массиве высоконадежные диски и хранить дубликаты файлов на другом, более надежном носителе информации, например на оптических дисках или магнитной ленте.

RAID 1: дисковый массив с дублированием или зеркалированием

Схему RAID 1 обычно используют при работе с файлами, особенно критичными к времени восстановления при отказах диска. В RAID 1 каждому «основному» диску массива придается так называемый «зеркальный» диск, содержащий идентичную копию данных (рис. 6.48). Восстановление при отказе осуществляется очень просто: когда одно дисковое ЗУ отказывает, данные могут быть просто взяты со второго диска пары, то есть информация доступна практически немедленно.

Рис. 6.48. Схема RAID 1

Основной и «зеркальный» диски в паре могут управляться либо общим контроллером («зеркалирование»), либо независимыми контроллерами («дублирование»). В варианте зеркалирования информация сначала заносится на основной диск, а затем — на «зеркальный». В случае дублирования возможна одновременная запись на оба диска, что более эффективно.

Запрос на чтение обслуживается тем диском пары, которому в данный момент требуется меньшее время поиска и меньшая задержка вращения.

Принципиальный изъян RAID 1 — высокая стоимость: требуется вдвое больше физического дискового пространства. По этой причине использование RAID 1 обычно ограничивают хранением загрузочных разделов, системного программного обеспечения и данных, а также других особенно критичных файлов: RAID 1 обеспечивает резервное копирование всех данных, так что в случае отказа диска критическая информация доступна практически немедленно.

RAID 2: дисковый массив с использованием кода Хэмминга

В схеме RAID 2 используется техника параллельного доступа, где в выполнении каждого запроса на ввод/вывод одновременно участвуют все диски.

Единицей хранимых данных служит слово. Количество информационных дисков равно разрядности слова и при записи каждый отдельный бит слова записывается на свой диск. Таким образом, каждое слово данных представлено лентой из идентично размещенных однобитовых полос. Шпиндели всех дисков синхронизированы так, что головки каждого диска в каждый момент времени находятся в одинаковых позициях.

Для каждой ленты, представляющей слово, вычисляется корректирующий код (обычно это код Хэмминга, способный корректировать одиночные и обнаруживать двойные ошибки). Биты этого кода хранятся на дополнительных дисках, выделенных для корректирующего кода, причем каждый бит на своем диске (рис. 6.49).

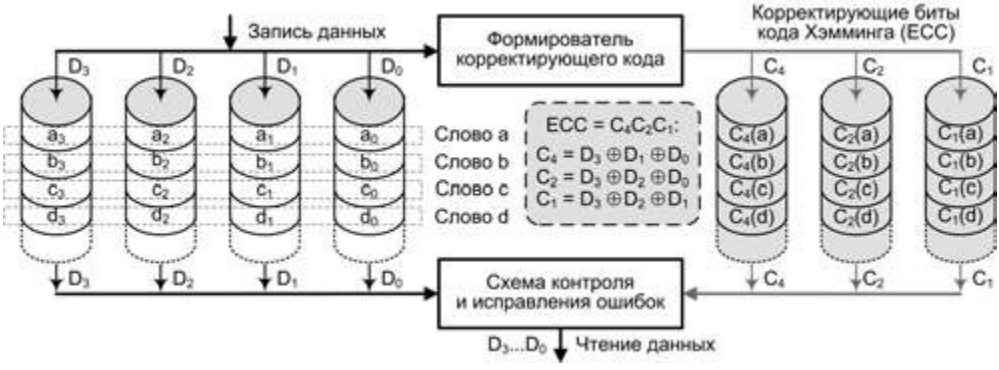


Рис. 6.49. Схема RAID 2

При записи слова вычисляется корректирующий код, который одновременно с записью данных на информационные диски заносится на диски, отведенные под корректирующий код. При чтении производится доступ ко всем дискам массива, включая

дополнительные. Считанные данные вместе с корректирующим кодом подаются на схему контроля и исправления ошибок контроллера дискового массива, где происходит повторное вычисление корректирующего кода и его сравнение с хранившимся на избыточных дисках. Если присутствует одиночная ошибка, контроллер способен ее мгновенно распознать и исправить, так что время считывания не увеличивается.

В реальности доступ к информации на дисках производится не пословно, а по блокам, размером не менее одного сектора диска. Это означает, что при изменении хотя бы одного бита должен быть перезаписан весь блок. Кроме того, предлагаемый в этой схеме метод коррекции с помощью кода Хэмминга уже встроен в современные дисковые ЗУ. В таких условиях использование нескольких избыточных дисков представляется неэффективным, и схема RAID 2 в настоящее время не применяется.

RAID 3: дисковый массив с параллельной передачей данных и четностью

RAID 3 организован сходно с RAID 2, но вместо кода Хэмминга вычисляется простой бит паритета (четности), то есть требуется только один дополнительный диск, вне зависимости от того, насколько велик массив дисков (рис. 6.50). В RAID 3 используется параллельный доступ к данным, разбитым на полосы длиной в байт. Все диски массива синхронизированы.

Рис. 6.50. Схема RAID 3

В случае отказа дискового ЗУ[1] восстановление данных производится достаточно просто. Рассмотрим массив из пяти дисковых ЗУ, где диски $D_3 - D_0$ содержат данные, а диск P — биты четности. Для i-го бита имеем:

$$P_i = D_{3i} \oplus D_{2i} \oplus D_{1i} \oplus D_{0i}.$$

Предположим, что отказал диск D_1. Если мы добавим к левой и правой частям предшествующего выражении $P_i \oplus D_{1i}$, то получим

$$D_{1i} = P_i \oplus D_{3i} \oplus D_{2i} \oplus D_{0i}.$$

[1] Как правило, RAID-контроллеры способны получить данные об ошибке с помощью механизмов отслеживания случайных сбоев

Таким образом, содержание каждой полосы данных на диске D_1 может быть восстановлено из содержания соответствующих полос на оставшихся дисках массива. Этот принцип справедлив для всех уровней RAID от 3 до 6.

Так как данные разбиты на очень маленькие полосы, RAID 3 позволяет достигать очень высоких скоростей передачи данных. Каждый запрос на ввод/вывод приводит к параллельной передаче данных со всех дисков. Схема RAID 3 предусматривает использование как минимум трех дисков и применяется обычно в системах, работающих с относительно большими последовательными записями, например файлами изображений.

RAID 4: массив независимых дисков с общим диском паритета

По своей идее и технике формирования избыточной информации RAID 4 идентичен RAID 3, только размер полос в RAID 4 значительно больше (обычно один-два физических блока на диске). Главное отличие состоит в том, что в RAID 4 используется техника независимого доступа, когда каждое ЗУ массива в состоянии функционировать независимо, так что отдельные запросы на ввод/вывод могут удовлетворяться параллельно (рис. 6.51). Благодаря этому массивы с независимым доступом наиболее подходят для приложений, характеризующихся высокой интенсивностью запросов, и наименее удобны для приложений с высокой интенсивностью пересылок данных.

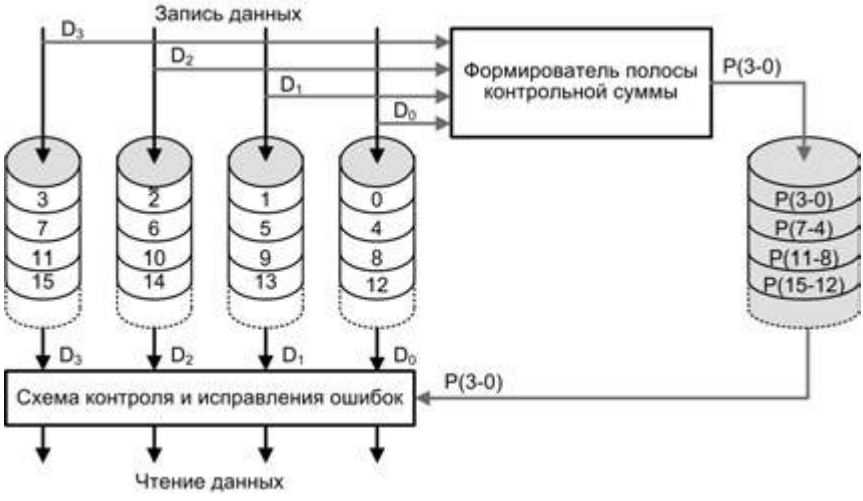


Рис. 6.51. Схема RAID 4

Для RAID 4 характерны издержки, обусловленные независимостью дисков. Если в RAID 3 запись производилась одновременно для всех полос одной ленты, в RAID 4 осуществляется запись полос в разные ленты. Это различие особенно ощущается при записи данных малого размера.

Каждый раз для выполнения записи программному обеспечению дискового массива необходимо обновить не только данные пользователя, но и соответствующие

биты паритета. Как и в случае RAID 3, рассмотрим массив из пяти дисковых ЗУ. Первоначально для каждого бита i мы имеем следующее соотношение:

$$P_i = D_{3i} \oplus D_{2i} \oplus D_{1i} \oplus D_{0i}.$$

Положим, что производится запись, охватывающая только полосу на диске D_1. После обновления, для потенциально измененных битов, обозначаемых с помощью апострофа, получаем:

$$\begin{aligned} P_i' &= D_{3i} \oplus D_{2i} \oplus D_{1i}' \oplus D_{0i} = \\ &= D_{3i} \oplus D_{2i} \oplus D_{1i}' \oplus D_{0i} \oplus D_{1i} \oplus D_{1i} = \\ &= P_i \oplus D_{1i} \oplus D_{1i}'. \end{aligned}$$

Для вычисления новой полосы паритетов необходимо сначала прочитать старую полосу пользователя и старую полосу паритета. Затем заменить эти две полосы новой полосой данных и новой вычисленной полосой паритета. Таким образом, запись каждой полосы связана с двумя операциями чтения и двумя операциями записи. Поэтому существенный выигрыш в производительности RAID 4 дает только при записи большого объема информации, охватывающего полосы на всех дисках.

Массивы RAID 4 наиболее подходят для приложений, требующих поддержки высокого темпа поступления запросов ввода/вывода, и уступают RAID 3 там, где приоритетен большой объем пересылок данных.

RAID 5: массив независимых дисков с распределенным паритетом

RAID 5 — самая распространенная схема. Различие с RAID 4 заключается в том, что RAID 5 не содержит отдельного диска для хранения информации паритета. Блоки данных и контрольная информация циклически записываются на все диски массива (рис. 6.52).

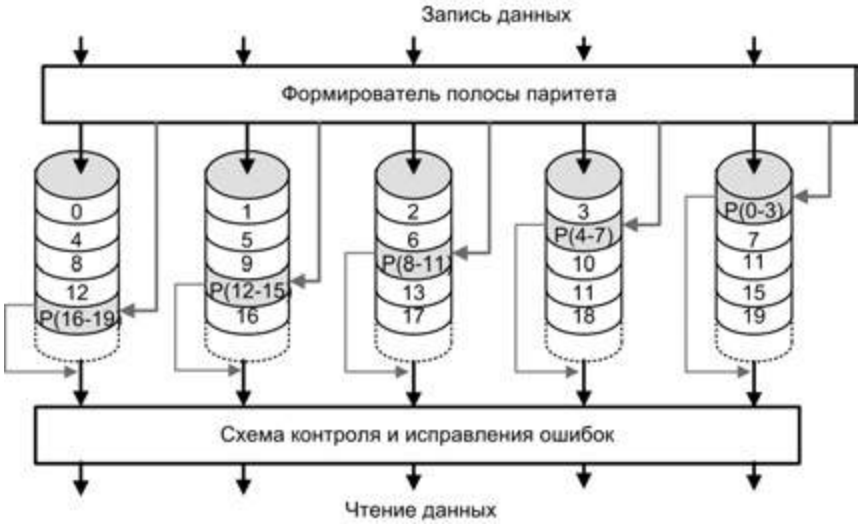


Рис. 6.52. Схема RAID 5

В N-дисковом массиве полоса паритета вычисляется для полос $N - 1$ дисков, расположенных в одной ленте, и хранится в той же ленте, но на диске, который не учитывался при вычислении паритета. При переходе от одной ленты к другой эта схема циклически повторяется. Такая модификация дает возможность параллельного выполнения $\frac{N}{2}$ записей. В целом схема показывает хорошую производительность при записи и при чтении, причем как небольших, так и больших массивов информации.

RAID 6: массив независимых дисков с двумя независимыми распределенными схемами паритета

RAID 6 очень похож на RAID 5. Данные и контрольная информация также разбиваются на полосы размером в блок и распределяются по всем дискам массива. Доступ к полосам независимый и асинхронный. Отличие состоит в том, что на каждом диске хранится не одна, а две полосы паритета, и хранятся они в отдельных блоках на разных дисках. Первая из полос паритета, как и в RAID 5, содержит контрольную информацию для полос, расположенных на горизонтальном срезе массива (за исключением диска, где эта полоса паритета хранится). В дополнение, формируется и записывается вторая полоса паритета, контролирующая все полосы какого-то одного диска массива (вертикальный срез массива), но только не того, где хранится данная полоса паритета. Сказанное иллюстрируется рис. 6.53.

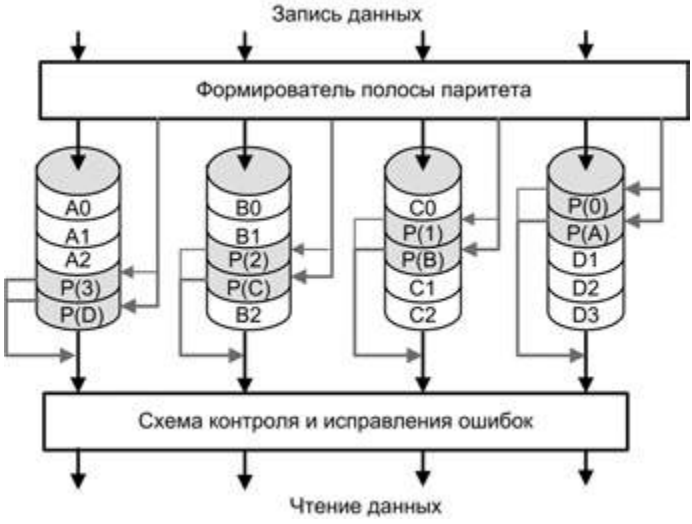


Рис. 6.53. Схема RAID 6

Такая схема массива позволяет восстановить информацию при отказе сразу двух дисков. С другой стороны, увеличивается время на вычисление и запись паритетной информации и требуется дополнительное дисковое пространство. Кроме того, реализация данной схемы связана с усложнением контроллера дискового массива. В силу этих причин схема среди выпускаемых RAID-систем встречается крайне редко.

RAID 7: дисковый массив, оптимизированный для повышения производительности

Схема RAID 7, запатентованная Storage Computer Corporation, объединяет массив асинхронно работающих дисков и кэш-память, управляемых встроенной в контроллер массива операционной системой реального времени (рис. 6.54). Данные разбиты на полосы, размером в блок, и распределены по дискам массива. Полосы паритета хранятся на специально выделенных для этой цели дисках.

Рис. 6.54. Схема RAID 7

Схема некритична к виду решаемых задач и при работе с большими файлами не уступает по производительности RAID 3. Вместе с тем, RAID 7 может так же эффективно, как и RAID 5, производить одновременно несколько операций чтения и записи для небольших объемов данных. Все это обеспечивается использованием кэш-памяти и собственной операционной системой.

Составные массивы RAID

Из-за того, что каждая из основных схем RAID имеет свои достоинства и недостатки, в последнее время все чаще используются так называемые составные массивы, представляющие собой комбинацию основных схем. Обычно в них сочетаются быстрый уровень RAID 0 с надежными уровнями RAID 1, RAID 3 или RAID 5. В названии составных массивов номер уровня содержит две цифры. Наиболее распространенными в настоящее время являются составные массивы:

RAID 01 (RAID 0 + RAID 1); RAID 10 (RAID 1 + RAID 0);
RAID 03 (RAID 0 + RAID 3); RAID 30 (RAID 3 + RAID 0);
RAID 05 (RAID 0 + RAID 5); RAID 50 (RAID 5 + RAID 0);
RAID 15 (RAID 1 + RAID 5); RAID 51 (RAID 5 + RAID 1).

Особенности реализации RAID-систем

Массивы RAID могут быть реализованы программно, аппаратно или как комбинация программных и аппаратных средств.

При программной реализации используются обычные дисковые контроллеры и стандартные команды ввода/вывода. Работа дисковых ЗУ в соответствии с алгоритмами различных уровней RAID обеспечивается программами операционной системы ВМ. Программный режим RAID предусмотрен, например, начиная с Windows NT. Это дает возможность программного изменения уровня RAID, в зависимости от особенностей решаемой задачи. Хотя программный способ является наиболее дешевым, он не позволяет добиться высокого уровня производительности, характерного для аппаратурной реализации RAID.

Аппаратурная реализация RAID предполагает возложение всех или большей части функций по управлению массивом дисковых ЗУ на соответствующее оборудование, при этом возможны два подхода. Первый из них заключается в замене стандартных контроллеров дисковых ЗУ на специализированные, устанавливаемые на место стандартных. Базовая ВМ общается с контроллерами на уровне обычных команд ввода/вывода, а режим RAID обеспечивают контроллеры. При втором способе аппаратной реализации RAID-система выполняется как автономное устройство, объединяющее в одном корпусе массив дисков и контроллер. Контроллер содержит микропроцессор и работает под управлением собственной операционной системы, полностью реализующей различные RAID-режимы. Такая подсистема подключается к шине базовой ВМ или к ее каналу ввода/вывода как обычное дисковое ЗУ.

При аппаратной реализации RAID-систем обычно предусматривается возможность замены неисправных дисков без потери информации и без остановки работы. Кроме того, многие из таких систем позволяют разбивать отдельные диски на разделы, причем разные разделы дисков могут объединяться в соответствии с различными уровнями RAID.

Запоминающие устройства на основе твердотельных дисков

Последние достижения в области полупроводниковых микросхем памяти сделали такие микросхемы привлекательными в плане создания на их базе альтернативы жестким магнитным дискам. Подобные «заменители» магнитных дисков получили название «твердотельных дисков» (SSD — Solid State Disk). Запоминающую среду в SSD образуют микросхемы энергозависимой (DRAM SSD) либо энергонезависимой (FLASH SSD) памяти. Твердотельные диски полностью взаимозаменяемы со стандартными накопителями на жестких магнитных дисках как по конструктивному исполнению, так и по интерфейсу.

DRAM SSD и FLASH SSD

Твердотельные диски на базе энергозависимой динамической памяти (DRAM SSD) характеризуются сверхбыстрыми чтением, записью и поиском информации. В то же время DRAM — это энергозависимая память, поэтому DRAM SSD, как правило, оснащены аккумуляторами для сохранения данных при потере питания, а более дорогие модели — системами резервного и/или оперативного копирования. Система питания утяжеляет накопитель и делает его менее надежным. В настоящее время DRAM SSD используются для ускорения работы крупных систем управления базами данных и мощных графических станций.

Накопители, построенные на основе энергонезависимой NAND флэш-памяти (FLASH SSD) появились относительно недавно. Пока стоимость FLASH SSD (3–10 $/ГБ) несколько выше, чем у магнитных дисков, однако с каждым годом она уменьшается на 20–30%.

Будучи сопоставимыми по скорости чтения и записи (порядка 70 Мбит/с), FLASH SSD-устройства значительно быстрее при поиске информации. Так, время доступа у них составляет около 0,5 мс против 11 мс у жестких магнитных дисков. По этой причине FLASH SSD предпочтительнее при работе с приложениями, для которых характерно частое чтение и запись небольших, случайно расположенных блоков данных.

FLASH SSD характеризуются относительно небольшими размерами и низким энергопотреблением. Они практически полностью завоевали рынок ускорителей баз данных среднего уровня и начинают теснить традиционные диски в мобильных приложениях.

Преимущества и недостатки по сравнению с жесткими дисками

По сравнению с жесткими магнитными дисками SSD обладает рядом преимуществ, большинство из которых являются следствием того, что SSD не содержит движущихся частей. Три основных преимущества:

- меньшая потребляемая мощность;
- более быстрый доступ к данным;
- более высокая надежность.

Первое из них связано с отсутствием основного потребителя энергии — двигателя, вращающего диск. В среднем потребляемая мощность сокращается на порядок. Более быстрый доступ связан с исключением затрат времени на перемещение головок считывания/записи. Наконец, отсутствие механически перемещаемых элементов, склонных к поломкам или способных стать причиной повреждений других устройств ВМ, положительно сказывается на надежности устройства в целом.

К упомянутым преимуществам SSD можно добавить:

- полное отсутствие шума от движущихся частей и охлаждающих вентиляторов;
- высокую механическую стойкость;
- широкий диапазон рабочих температур;
- практически устойчивое время считывания файлов вне зависимости от их расположения или фрагментации;
- малый размер и вес;
- совместимость со стандартными интерфейсами, используемыми для подключения дисковых ЗУ — SATA-I, SATA-2.

Основными недостатками (на момент написания раздела) можно считать:

- высокую стоимость (от 8 $/Гбайт);
- относительно малую емкость (в продаже доступны Flash SSD до 250 Гбайт);
- более высокую чувствительность к внезапной потере питания, магнитным и электрическим полям;
- ограниченное количество циклов перезаписи (несколько миллионов раз).

Дисковая кэш-память

Концепция кэш-памяти применима и к дисковым ЗУ. Принцип кэширования дисков во многом схож с принципом кэширования основной памяти, хотя способы доступа к диску и ОП существенно разнятся. Если время обращения к любой ячейке ОП одинаково, то для диска оно зависит от целого ряда факторов. Во-первых, нужно затратить некоторое время для установки головки считывания/записи на нужную дорожку. Во-вторых, поскольку при движении головка вибрирует, необходимо подождать, чтобы она успокоилась. В-третьих, искомый сектор может оказаться под головкой также лишь спустя некоторое время.

Дисковая кэш-память представляет собой память с произвольным доступом, «размещенную» между дисками и ОП. Емкость такой памяти обычно достаточно велика — от 8 Мбайт и более. Пересылка информации между дисками и основной памятью организуется контроллером дисковой кэш-памяти. Изготавливается дисковая кэш-память на базе таких же полупроводниковых запоминающих устройств, что и основная память, поэтому в ряде случаев с ней обращаются как с дополнительной основной памятью.

В дисковой кэш-памяти хранятся блоки информации, которые с большой вероятностью будут востребованы в ближайшем будущем. Принцип локальности, обеспечивающий эффективность обычной кэш-памяти, справедлив и для дисковой, приводя к сокращению времени ввода/вывода данных от величин 20–30 мс до значений порядка 2–5 мс, в зависимости объема передаваемой информации.

В качестве единицы пересылки может выступать сектор, несколько секторов, а также одна или несколько дорожек диска. Кроме того, иногда применяется пересылка информации, начиная с выбранного сектора на дорожке и до ее конца. В случае пересылки секторов кэш-память заполняется не только требуемым сектором, но секторами, непосредственно следующими за ним, так как известно, что в большинстве случаев взаимосвязанные данные хранятся в соседних секторах. Этот метод известен также как опережающее чтение (read ahead).

Для дисковой кэш-памяти наиболее характерно полностью ассоциативное отображение, замещение информации по алгоритму LRU и согласование информации методом сквозной записи.

Специфика дисковой кэш-памяти состоит в том, что далеко не всю информацию, перемещаемую между дисками и основной памятью, выгодно помещать в дисковый кэш. В ряде случаев определенные данные и команды целесообразно пересылать напрямую между ОП и диском. По этой причине в системах с дисковой кэш-памятью предусматривают специальный динамический механизм, позволяющий переключать тракт пересылки информации: через кэш или минуя его.

Одна из привлекательных сторон дисковой кэш-памяти в том, что связанные с ней преимущества могут быть получены без изменений в имеющемся аппаратном и программном обеспечении. Многие серийно выпускаемые дисковые кэши интегрированы в состав дисковых ЗУ. Дисковая кэш-память применяется и в персональных ВМ.

Запоминающие устройства на основе оптических дисков

Запоминающие устройства на базе оптических дисков (ЗУОД) — это сравнительно «молодое» направление в области внешних запоминающих устройств. Накопители на оптических дисках появились в 80-х годах XX века и первоначально предназначались для звуковой звукозаписи, однако быстро стали популярными в качестве устройств внешней памяти. Технология ЗУОД с самого момента появления постоянно совершенствовалась, в результате чего к настоящему времени распространение получили несколько типов таких устройств: CD (Compact Disks — компакт-диски), DVD (Digital Versatile Disks — цифровые универсальные диски) и пока еще в меньшей степени — BD (Blue-ray Disks — диски с голубым лучом). Каждый следующий тип в этом списке можно рассматривать как результат эволюции предшествующих типов, поскольку, несмотря на отличия, в основе всех упомянутых типов лежат некоторые общие принципы.

Общие принципы построения ЗУОД

В отличие от жестких магнитных дисков, где магнитные диски и привод представляют собой единый блок, ЗУОД включает два компонента: съемный оптический диск и привод, обеспечивающий вращение диска. На приводе также размещаются записывающий и считывающий лазеры и соответствующие электронные схемы.

Носителем информации в ЗУОД служат круглые пластмассовые диски, с отражающим слоем[1] с диаметром 12 или 8 см. Как чтение, так и запись (если диск предусматривает такую возможность) информации осуществляется лазерным лучом, направляемым на поверхность вращающегося диска[2].

В исходном состоянии вся поверхность диска обладает «идеальными» отражающими свойствами. В процессе записи информации определенные участки отражающего слоя диска, расположенные вдоль траектории перемещения луча записывающего лазера, нагреваются. В зависимости от типа отражающего слоя в облученном месте возникают углубления, либо материал отражающего слоя переходит в иное состояние. В обоих случаях изменяются показатели отражения участка отражающего слоя. Для обозначения углублений или участков с измененным состоянием применяют термин «*пит*». Промежутки между питами, для которых сохраняется исходное «идеальное» отражение, называют *лендами*. Ленды отражают бóльшую часть падающего на них света лазерного излучателя, а питы в силу своей удаленности от точки фокуса не отражают практически ничего.

Траектория перемещения луча лазера по поверхности вращающегося диска имеет вид единой спиральной дорожки, начинающейся от центра диска и идущей к его внешней границе. Питы формируются вдоль этой дорожки.

---

[1] В рекламных целях иногда выпускаются так называемые Shape CD не круглой формы, а с очертанием внешнего контура в форме разнообразных объектов, таких как силуэты машины, сердечки, овалы и т. п. Обычно такие диски не рекомендуют применять в компьютерных приводах, поскольку при высоких скоростях вращения диск может лопнуть, следствием чего будет полный выходу привода из строя.

[2] Для записи обычно используется более мощный лазер

При чтении отраженный луч считывающего лазера фиксируется приемником, при этом в зависимости от того, является ли облучаемый участок питом или лендом, характеристики отраженного луча (интенсивность или фаза) будут различными. Условно можно считать, что луч, отраженный от ленда, воспринимается приемником, а отраженный от пита в приемник не попадает. Такое различие можно трактовать как 1 и 0.

На самом деле питы и ленды не являются битами 0 или 1 в привычном понимании, а представляют сразу несколько битов информации. Связано это с тем, что для надежности хранения информацию перед записью подвергают помехоустойчивому кодированию. При таком кодировании каждый байт информации с помощью специальной таблицы заменяется 14-разрядным словом. К этому слову добавляются три так называемых соединительных бита. В полученном 17-разрядном коде между двумя единицами никогда не может быть менее двух и более десяти нулей. Именно такие последовательности и представлены питами и лендами, причем питы и ленды чередуются в моменты, соответствующие началу бита, равного 1. С позиций представления информации питы и ленды равнозначны и характеризуют лишь временные интервалы между двумя единицами в последовательности битов. Сенсор тестирует поверхность через равные временные интервалы. Так как временные интервалы отсчитываются в количестве периодов тактовой частоты, то протяженность каждого пита или ленда лежит в пределах от 3 (интервал 3T) до 11 (интервал 11T) периодов тактовой частоты (учитывается и бит, равный 1). Переход от пита к ленду (или наоборот) соответствует логической 1, а логический 0 представляется отсутствием переходов в данном месте (рис. 6.55).

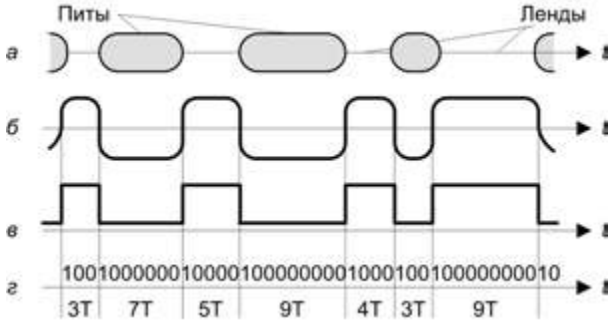


Рис. 6.55. Представление информации на оптическом диске: *а* — питы и ленды (вид сверху); *б* — информационный сигнал с фотоприемника; *в* — преобразованный информационный сигнал; *г* — интервалы

Информационный сигнал с приемника (рис. 6.55, *б*) преобразуется в последовательность прямоугольных импульсов, длительность которых кратна периоду следования битов последовательного кода (рис. 6.55, *в*). Каждые 14 последовательных периодов декодируются в один байт информации, соединительные разряды просто отбрасываются.

Таким образом, информация на оптическом диске представлена цепочкой питов и лендов, расположенной вдоль спиралевидной дорожки. Одним из кардинальных

отличий между разными типами ЗУОД служит длина волны используемых лазеров. Чем она меньше, тем меньше размер сфокусированного луча лазера, а значит, размер питов и лендов, ширина дорожки и расстояние между витками спирали. Как следствие, возрастает количество информации, которую можно разместить на диске. На рис. 6.56 приведены увеличенные фотографии поверхности дисков типа CD и DVD.

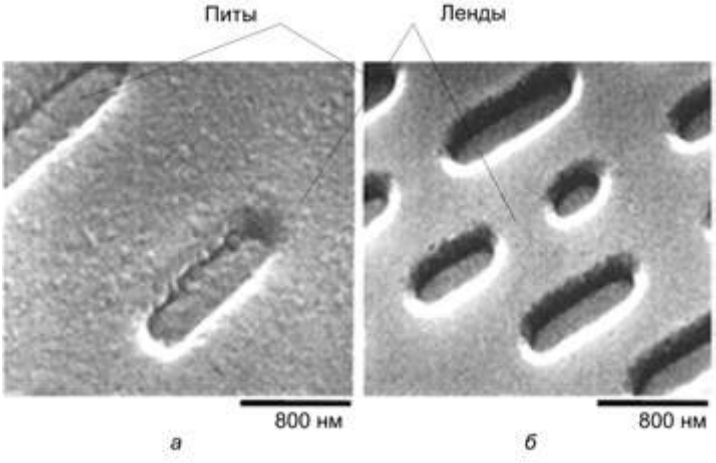


Рис. 6.56. Размещение информации на поверхности дисков типа: *а* — CD; *б* — DVD

Способы записи информации на оптические диски

В рамках каждого из упоминавшихся типов ЗУОД существуют оптические диски, различающиеся способом занесения на них информации и возможностями ее смены. Обычно выделяют три группы дисков, которые можно отличить по аббревиатуре, отделяемой от названия типа ЗУОД символом «минус» или «плюс», например CD-ROM, DVD-R или DVD+RW[1].

Первую группу образуют *прессованные* диски, информация в которые заносится в процессе изготовления. В их названии используется добавка ROM по аналогии с масочными постоянными запоминающими устройствами. Сначала изготавливается так называемый мастер-диск, запись информации на который производится с помощью сильно сфокусированного луча лазера высокой интенсивности или механически. На основе мастер-диска создается матрица для штамповки копий, которые и являются оптическими дисками. Копия представляет собой диск из пластмассы, например поликарбоната. Цифровая информация в виде углублений на подложке переносится на диск с матрицы путем штамповки. Поверхность копий с углублениями покрывается алюминием или золотом, то есть материалом с высо-

[1] Конкуренция между производителями оптических дисков привела к двум устоявшимся стандартам и, как следствие, к двум форматам — формату «плюс» и формату «минус». Большинство приводов способно работать с дисками любого из этих форматов.

кой отражающей способностью. Далее углубления на копии защищаются от пыли и повреждений путем покрытия поверхности диска прозрачным лаком. Рабочая поверхность таких дисков обычно белого цвета.

Вторая группа оптических дисков — это диски *с однократной записью*. Они обозначаются добавкой R (от Recordable — записываемые), например CD–R (CD+R) или DVD–R (DVD+R). Технология дисков с однократной записью и многократным считыванием была разработана для мелкосерийного производства оптических дисков. Такие диски предполагают однократную запись информации лучом относительно мощного лазера с возможностью последующего многократного считывания. Диск также выполнен из поликарбоната, но в исходном состоянии на его поверхности уже сформирована направляющая спиральная канавка, вдоль которой в дальнейшем перемещается луч лазера. Эта канавка заполнена органическим красителем. В процессе записи на отдельных участках мощность лазера увеличивается относительно уровня считывании примерно на порядок. Энергия лазерного луча поглощается органическим красителем и преобразуется в тепло. Иногда этот процесс называется «прожигание». В результате нагрева краситель обугливается и в нем появляются микроскопические газовые пузырьки. В процессе выделения газов увеличивается объем красителя и деформируется отражающий слой. Краситель нагревается до температуры, превышающей температуру плавления поликарбоната, вследствие чего и сама основа в данной точке плавится и деформируется, а отражающая способность на облученном участке ухудшается. С позиций отражения такой участок эквивалентен питу. Роль лендов выполняет не вся поверхность диска, а лишь недеформированные участки канавки. В зависимости от применяемого красителя диски имеют цвета зеленоватых или синих оттенков. Данный тип носителя привлекателен для архивного хранения документов и файлов.

Третью группу образуют *перезаписываемые* диски. Их обозначают добавкой RW (Rewritable), например CD–RW (CD+RW) или DVD–RW (DVD+RW). Информация в оптических дисках данного типа может быть многократно перезаписана, как это имеет место с магнитными дисками. В дисках используется эффект *фазового перехода*, для чего используется материал, который может быть в одном из двух фазовых состояний, причем отражающая способность его в этих состояниях существенно отличается. Одно из этих состояний — аморфное, когда молекулы материала имеют случайную ориентацию и материал отражает свет плохо. Второе состояние — кристаллическое, характеризующееся высокой отражающей способностью. Под воздействием луча лазера активный слой оптического диска может менять свое состояние с кристаллического на аморфное, и наоборот.

До записи поверхность диска находится в кристаллическом состоянии. Для перевода участка активного слоя в аморфное состояние (записи пита) он облучается коротким лазерным импульсом высокой мощности. Импульс нагревает участок до температуры T, превышающей температуру плавления $T_{плав}$ ($T > T_{плав}$), и расплавляет материал активного слоя в этом месте. Затем следует охлаждение ниже температуры кристаллизации $T_{крист}$, при этом центры кристаллизации не образуются, и вещество остается в аморфном состоянии. Для стирания информации надо вернуть вещество в кристаллическое состояние. Это достигается опять же с помощью

лазера, но работающего в другом режиме. Аморфное вещество нагревают до температуры, меньшей, чем $T_{плав}$, но большей, чем $T_{крист}$ ($T_{крист} < T < T_{плав}$). Нагрев (его называют обжигом) продолжается в течение времени, достаточного для восстановления кристаллического состояния вещества.

Главный недостаток оптических дисков с фазовым переходом заключается в том, что материал со временем теряет желательные свойства. Современные материалы могут быть использованы для 500 000–1 000 000 циклов стирания.

Возможность перезаписи позволяет использовать диски типа RW в качестве вторичной памяти как дополнение к магнитным дискам.

Оптические диски типа CD

Компакт-диск (CD — compact disk) — это односторонний диск, способный хранить двоичную информацию. Выпускаются CD различной емкости. В типовом варианте расстояние между витками спиралевидной дорожки составляет 1,6 микрон, что позволяет обеспечить 20 344 витка, при этом длина спиралевидной дорожки равна 5,27 км. Диск вращается с постоянной линейной скоростью 1,2 м/с, то есть для «прохождения» спирали требуется 4391 с или 73,2 мин. Так как данные считываются с диска со скоростью 176,4 Кбайт/с, емкость CD равна 774,57 Мбайт.

Рис. 6.57. Формат блока CD-ROM

Данные на CD-ROM организованы как последовательность блоков. Типичный формат блока показан на рис. 6.57. Блок включает в себя следующие поля:

- *Синхронизация*. Это поле идентифицирует начало блока и состоит из байта со всеми нулями, десяти байтов, содержащих только единичные разряды, и вновь байта из всех нулей.
- *Идентификатор*. Заголовок, содержащий адрес блока и байт режима. Режим 0 определяет пустое поле данных; режим 1 отвечает за использование кода, корректирующего ошибки, и наличие 2048 байтов данных; режим 2 определяет наличие 2336 байтов данных и отсутствие корректирующего кода.
- *Данные*. Данные пользователя.
- *Корректирующий код (КК)*. Поле предназначено для хранения дополнительных данных пользователя в режиме 2, а в режиме 1 содержит 288 байтов кода с исправлением ошибок.

Рисунок 6.58 иллюстрирует организацию информации на CD-ROM.

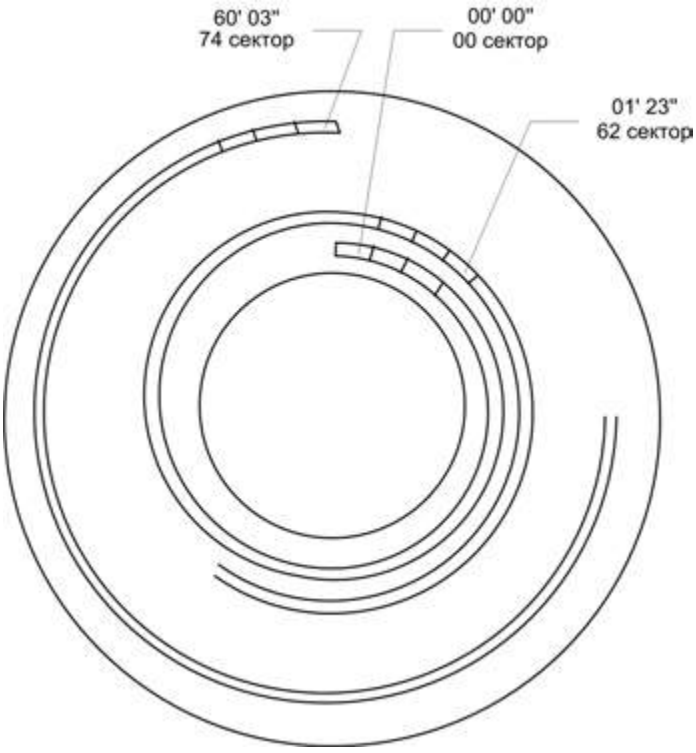


Рис. 6.58. Организация диска с постоянной линейной скоростью

Ввиду вращения диска с постоянной линейной скоростью (ПЛС), а не с постоянной угловой скоростью (как это имеет место в случае магнитных дисков) произвольный доступ к информации становится более сложным. Позиционирование по указанному адресу включает перемещение головки к общей области, подбирая скорость вращения, и чтение адреса, а затем небольшую регулировку для нахождения и доступа к нужному сектору.

Для обеспечения более быстрого доступа в современных ЗУОД на базе CD поддерживается метод постоянной угловой скорости при относительном снижении емкости (до 682 Мбайт).

Оптические диски типа DVD

Последние годы характеризуются повсеместным переходом от CD к так называемым DVD-дискам (Digital Versatile Disk — цифровой универсальный диск). Официально DVD был анонсирован в сентябре 1995 года. По сравнению с CD в DVD существенно увеличена емкость диска, что достигается благодаря трем моментам:

1. Биты на DVD упакованы более плотно. Расстояние между витками спирали на CD равно 1,6 мкм, и минимальное расстояние между питами составляет 0,834 мкм. В DVD-технологии используется лазер с меньшей длиной волны (650 нм против 780 нм для стандартных CD), что позволяет сократить расстояние между

витками до 0,74 мкм, а между питами — до 0,4 мкм. Результатом этих двух улучшений становится почти семикратное увеличение емкости, то есть вместо 682 Мбайт до 4,7 Гбайт.

2. В DVD реализован второй слой питов и лендов поверх первого слоя. В двухслойном DVD поверх отражающего слоя расположен полуотражающий слой, и путем изменения фокусировки накопители DVD могут читать каждый слой по отдельности. Эта техника почти удваивает емкость диска, увеличивая ее до величины порядка 8,5 Гбайт. Более низкие отражающие свойства второго слоя ограничивают емкость диска, поэтому полного удвоения не происходит. Вместимость можно определить на глаз – нужно посмотреть, сколько рабочих (отражающих) сторон у диска, и обратить внимание на их цвет: двухслойные стороны обычно имеют золотой цвет, а однослойные — серебряный, как у компакт-диска.
3. DVD-ROM могут быть двусторонними, в то время как на CD данные записываются только на одной стороне. При использовании двустороннего DVD его нужно переворачивать.

На рис. 6.59 показана структура диска типа CD и типа DVD.

Рис. 6.59. Структура оптического диска: *а* — CD емкостью 682 Мбайт; *б* — двустороннего двухслойного DVD емкостью 17 Гбайт

Оптические диски типа BD

Blue-ray (голубой луч) диски или сокращенно BD представляют собой очередное поколение оптических дисков. В технологии Blue-ray для чтения и записи используется сине-фиолетовый лазер с длиной волны 405 нм. Напомним, что обычные DVD и CD используют красный и инфракрасный лазеры с длиной волны 650 нм и 780 нм соответственно. Такое уменьшение позволило сузить дорожку вдвое (до 0,32 микрон). В сочетании с другими изменениями (высококачественной системой фокусировки луча с двумя линзами и уменьшением толщины защитного слоя на носителе) это позволило записывать информацию в меньшие точки на диске, а значит, при сохранении стандартных размеров диска хранить на нем больше информации, а также увеличить скорость считывания до 430–435 Мбит/с.

В настоящее время выпускаются BD-диски с диаметром 12 и 8 см, причем оба могут быть однослойными и двухслойными. Емкость 12-сантиметровых дисков

может достигать 27 Гбайт (в однослойном варианте) или 54 Гбайт (в двухслойном варианте). Аналогичные показатели для 8-сантиметровых дисков — 7,8 Гбайт и 15,6 Гбайт соответственно. В разработке находятся диски вместимостью 100 Гбайт и 200 Гбайт с использованием соответственно четырех и шести слоев.

Перспективные типы оптических дисков

Среди перспективных типов оптических дисков, прежде всего, следует упомянуть ЗУОД типа HVD. HVD (Holographic Versatile Disk — голографический универсальный диск) — новая технология хранения данных на оптических носителях, которая значительно увеличит объем информации, записываемой на один диск. Она основывается на явлении, называемом коллинеарная голография. В работе используются два лазера — красного и сине-зеленого цветов. Сине-зеленый лазер считывает информацию, закодированную в голографическом слое, а красный лазер — читает информацию для правильного позиционирования механизма из обычного алюминиевого слоя в самой глубине диска. На CD и DVD эта информация записывается вместе с основными данными. В HVD же есть особый пограничный слой, отражающий сине-зеленый лазер, но пропускающий красный. Эта особенность позволяет разделять два потока информации (служебную и сами данные), что было недостижимо в более ранних вариантах подобных устройств. Новые носители могут хранить до 3,9 Тбайт данных, что примерно в 160 раз превышает емкость обычного BD-диска. Планируемая скорость передачи данных — 1 Гбит/с.

В 2009 году австралийские ученые представили новый тип ЗУОД, в котором оптический диск способен хранить до 1,6 терабайт информации. Информация записывается в несколько уровней, и если плотность записи на DVD равна 51 мегабайт на квадратный сантиметр, то здесь плотность составляет 1,1 терабита на кубический сантиметр. В основе технологии лежит эффект оптической поляризации светового излучения. Данные записываются в несколько слоев, отделенных друг от друга тончайшей прослойкой на основе молекул золота. У каждого слоя своя способность к светоотражению, каждый слой способен воспринимать только свет определенной длины волны. Когда луч лазера заданной длины достигает искомого слоя, то происходит эффект поляризации и слой начинает предоставлять данные, записанные на нем. Это позволяет хранить множество битов на одном и том же месте. Сейчас созданы считывающие системы, способные работать с 6–9 слоями. В перспективе предполагается довести количество слоев до 10. Появление подобных ЗУОД на рынке ожидается к 2015–2017 году, поскольку необходимо разрешить несколько проблем, в частности проблему разогрева дисков, так как лазерные лучи, считывающие данные, сравнительно мощны. Кроме того, предстоит доработать считывание таким образом, чтобы слои ни в коем случае не читались одновременно.

Запоминающие устройства на основе магнитных лент

ЗУ на базе магнитных лент (ЗУМЛ) в иерархии запоминающих устройств занимают нижнюю позицию и используются в основном в качестве устройств резервного копирования информации.

Общие принципы построения ЗУМЛ

В ленточных системах используется та же техника чтения и записи, что и в дисковых системах. Носителем служит тонкая полистироловая лента, покрытая намагничивающимся материалом. Покрытие может состоять из частиц технически чистого металла и связующего материала либо специально напыленной металлической пленки. В различных типах ЗУМЛ лента может находиться в кассетах или картриджах. Кассета представляет собой корпус с двумя бобинами, на которые намотана магнитная лента. В картридже имеется лишь одна бобина с магнитной лентой, а приемная бобина находится в накопителе, куда устанавливается картридж. В момент установки картриджа свободный конец ленты из картриджа захватывается специальным устройством накопителя и наматывается на приемную бобину.

В зависимости от типа ЗУМЛ ширина ленты варьируется от 0,38 см (0,15 дюйма) до 1, 27 см (0,5 дюйма). Толщина ленты порядка 0,025 мм.

Технологии записи

При записи информации на ЗУМЛ применяется один из двух подходов: линейная запись или наклонная запись.

Линейный метод записи. В первых ЗУМЛ, реализующих этот метод, использовалась единственная дорожка, протяженностью во всю длину ленты. Данные представляются как последовательность битов, расположенных вдоль дорожки, аналогично тому, как это имеет место в случае дисков. Биты информации сгруппированы в байты, содержащие 9 битов информации (8 информационных и один контрольный). Так же, как и в дисках, данные читаются и записываются в виде смежных блоков, называемых *физическими записями*. Каждая запись отделялась от соседней *межблочным промежутком*, дающим возможность позиционирования головки считывания/записи на начало любого блока. Конец каждого блока помечается специальным маркером «конец блока», а конец ленты — маркером «конец ленты». Как и диски, лента форматируется для обеспечения доступа к конкретной физической записи.

Позже число дорожек стали увеличивать, а сами дорожки стали логически соединять в виде серпантина. Это позволило выполнять операции чтения/записи при двустороннем движении; увеличилась «логическая» длина ленты, которая составляла сумму длин всех дорожек. Такая техника записи представляет собой подвид линейной записи и называется *серпантинной записью.* Первое множество битов записывается вдоль полной длины ленты. При достижении конца ленты головки позиционируются на новую дорожку, и запись опять продолжается вдоль всей длины ленты, но уже при движении ленты в противоположном направлении. Процесс продолжается, пока не будет заполнена вся лента (рис. 6.60, *а*). Для увеличения скорости головка способна читать или писать несколько смежных дорожек одновременно (обычно от 2 до 8). Данные также записываются последовательно вдоль индивидуальных дорожек, но последовательные блоки распределяются по смежным дорожкам, как это показано на рис. 6.60, *б*.

Рис. 6.60. Типовая организация хранения данных на магнитной ленте при линейном методе записи: *а* — серпантинное чтение и запись; *б* — расположение блоков в системе, где одновременно читаются 4 дорожки

Наклонный-строчный метод записи. В ЗУМЛ с наклонно-строчной записью несколько головок считывания/записи размещают на вращающемся барабане, установленном под углом к вертикальной оси (рис. 6.61). Последний охватывается лентой, прилегающей к нему рабочим слоем. Количество головок может быть различным; чаще всего применяют две головки, сдвинутые относительно друг друга на 180°. Ось барабана наклонена к плоскости движения ленты, благодаря чему и получается наклонное расположение дорожек. Лента транспортируется медленно, а головки вращаются с большой скоростью. Движение ленты при записи и чтении может производиться только в одном направлении. Сочетание быстрого вращения барабана и медленного перемещения ленты обеспечивает высокую плотность и скорость записи.

Рис. 6.61. Наклонно-строчный принцип записи: *а* — принцип записи; *б* — охват лентой барабана

Устройства резервного копирования

В процессе эволюции вычислительной техники характер задач, возлагавшихся на ЗУМЛ, неоднократно менялись. К настоящему моменту такие ЗУ используются главным образом в качестве устройств резервного копирования (УРК) информации. Такие устройства обеспечивают создание копий данных, хранящихся на магнитных дисках, с целью их восстановления в случае потери или искажения. Исходя из этого, рассмотрим лишь те типы ЗУМЛ, которые используются именно для этой цели. В основе всех типов систем резервного копирования на базе магнитных лент всегда лежит *стример*.

Стримером называют накопитель, который записывает информацию на установленный в этот накопитель картридж или кассету.

Следующим типом УРК можно считать *стекер,* представляющий собой стример, которому придаются несколько картриджей, собранных в специальный лоток. Картриджи подаются в накопитель с помощью роботизированного механизма, причем последовательность подачи картриджей строго определена.

Разновидностью стекера можно считать *автозагрузчик.* Отличие состоит в том, что картриджи, размещенные в специальном магазине, могут подаваться в стример в произвольном порядке.

Значительно большие возможности для резервного копирования предоставляет *ленточная библиотека.* Она содержит несколько стримеров, так называемые отсеки, в которых хранятся картриджи, и механизм смены картриджей в стримерах. Возможны две схемы смены картриджей в стримерах: в первой схеме любой картридж может быть загружен в любой стример; во второй — каждый стример работает лишь с определенными картриджами. Ленточную библиотеку отличает высокая скорость передачи данных, огромная емкость и большая надежность.

В рамках рассматриваемой группы УРК имеется также система, по структуре аналогичная RAID и полностью повторяющая спецификации RAID. Такой структурой являются массивы стримеров с избыточностью (RAIT — Redundant Array of Independent Tape). Задачи, решаемые с помощью RAIT, те же, что и в случае RAID — обеспечение высокой скорости и надежности. Физически RAIT-массив представляет собой корпус, где расположены несколько одновременно работающих стримеров. К сожалению, RAIT-массивы имеют малую емкость и не позволяют проводить автоматическую смену (ротацию) картриджей.

Контрольные вопросы

1. Какие операции определяет понятие «обращение к ЗУ»?
2. Какие единицы измерения используются для указания емкости запоминающих устройств?
3. В чем отличие между временем выборки и циклом обращения к запоминающему устройству?
4. Чем вызвана необходимость построения системы памяти по иерархическому принципу?
5. Что включает в себя понятие «локальность по обращению»?
6. Благодаря чему среднее время доступа в иерархической системе памяти определяется более быстродействующими видами ЗУ?
7. Что в иерархической системе памяти определяют термины «промах» и «попадание»?
8. На какие вопросы необходимо ответить, чтобы охарактеризовать определенный уровень иерархической памяти?
9. Какие виды запоминающих устройств может содержать основная память?
10. Охарактеризуйте возможные варианты построения блочной памяти.

11. Какие возможности по сокращению времени доступа к информации предоставляет блочная организация памяти?
12. Чем обусловлена эффективность расслоения памяти?
13. Какая топология запоминающих элементов лежит в основе организации полупроводниковых ЗУ?
14. Какое минимальное количество линий должен содержать столбец ИМС памяти?
15. Поясните назначение управляющих сигналов в микросхеме памяти.
16. Чем отличаются страничный, быстрый страничный и пакетный режимы доступа к памяти?
17. Чем обусловлена необходимость регенерации содержимого динамических ОЗУ?
18. Охарактеризуйте основные сферы применения статических и динамических ОЗУ.
19. Какое влияние на асинхронный режим работы памяти оказывает синхронный характер работы контроллера памяти?
20. В чем состоит особенность подхода, применяемого в микросхемах ОЗУ типа RDRAM и SLDRAM?
21. Какой вид ПЗУ обладает наиболее высокой скоростью перепрограммирования?
22. Какими методами обеспечивается энергонезависимость ОЗУ?
23. В чем состоит различие между режимами стандартной и запаздывающей записи в статических ОЗУ?
24. В чем проявляется специфика ОЗУ, предназначенных для видеосистем?
25. Каким образом в многопортовых ОЗУ разрешаются конфликты при одновременном доступе к памяти?
26. Какую функцию выполняет система семафоров в многопортовой памяти?
27. Для каких целей предназначена память типа FIFO?
28. Какая идея лежит в основе систем обнаружения и коррекции ошибок?
29. Какие ошибки может обнаруживать схема контроля по паритету?
30. От чего зависят возможности выявления и коррекции ошибок с использованием кода Хэмминга?
31. Поясните назначение и принцип формирования кода синдрома в системе коррекции ошибок.
32. Чем объясняется тенденция размещения стека в области старших адресов основной памяти?
33. Какая информация хранится в указателе стека?
34. Поясните назначение маски в ассоциативном запоминающем ЗУ.
35. Как реализуется запись новой информации в ассоциативное ЗУ?
36. Какие виды поиска можно осуществлять в ассоциативном ЗУ?
37. Поясните назначение и логику работы кэш-памяти.
38. Какие проблемы порождает включение в иерархию ЗУ кэш-памяти?
39. Чем обусловлено разнообразие методов отображения основной памяти на кэш-память?

40. Какому требованию должен отвечать «идеальный» алгоритм замещения содержимого кэш-памяти?
41. Какими методами обеспечивается согласованность содержимого основной и кэш-памяти?
42. Чем обусловлено введение дополнительных уровней кэш-памяти?
43. Какие факторы влияют на выбор емкости кэш-памяти и размера блока?
44. Как соотносятся характеристики обычной и дисковой кэш-памяти?
45. Какими средствами обеспечивается виртуализация памяти?
46. Существует ли ограничение на размер виртуального пространства?
47. Что определяет объем страничной таблицы?
48. Какими приемами достигают сокращения объема страничных таблиц?
49. Какие алгоритмы замещения используются при загрузке в основную память новой виртуальной страницы?
50. Поясните назначение буфера быстрого преобразования адреса (TLB).
51. Чем мотивируется разбиение виртуальных секторов на страницы?
52. Какая часть виртуального адреса остается неизменной при его преобразовании в физический адрес?
53. Чем обусловлена необходимость защиты памяти?

Глава 7

Организация шин

Совокупность трактов, объединяющих между собой основные устройства ВМ, образует *систему соединений* вычислительной машины. Система соединений должна обеспечивать обмен информацией между:

- центральным процессором и памятью;
- центральным процессором и модулями ввода/вывода;
- памятью и модулями ввода/вывода.

По мере развития вычислительной техники система соединений устройств ВМ также претерпевала изменения (рис. 7.1). На начальной стадии преобладали непосредственные связи между взаимодействующими устройствами ВМ. С появлением миниЭВМ и особенно первых микроЭВМ все более популярной становится схема с одной общей шиной. Последовавший за этим быстрый рост производительности практически всех устройств ВМ привел к неспособности единственной шины справиться с возросшим трафиком, и ей на смену приходят структуры на базе нескольких шин.

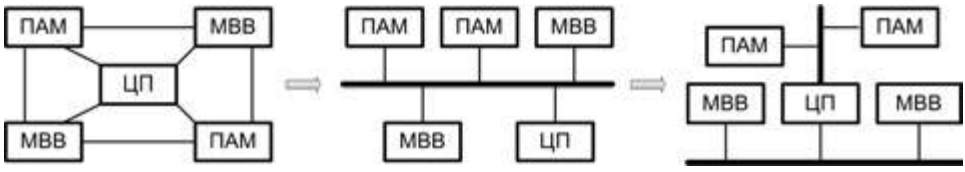


Рис. 7.1. Эволюция систем соединений (ЦП — центральный процессор, ПАМ — модуль основной памяти, МВВ — модуль ввода/вывода)

Взаимосвязь частей вычислительной машины и ее «общение» с внешним миром обеспечиваются системой шин. Большинство машин содержат несколько различных шин, каждая из которых оптимизирована под определенный вид коммуникаций. Часть шин скрыта внутри интегральных микросхем или доступна только в пределах печатной платы. Некоторые шины имеют доступные извне точки, с тем чтобы к ним легко можно было подключить дополнительные устройства, причем

большинство таких шин не просто доступны, но и отвечают определенным стандартам, что позволяет подсоединять к шине устройства различных производителей.

Чтобы охарактеризовать конкретную шину, нужно описать (рис. 7.2):

- совокупность сигнальных линий;
- физические, механические и электрические характеристики шины;
- используемые сигналы арбитража, состояния, управления и синхронизации;
- правила взаимодействия подключенных к шине устройств (протокол шины).

Рис. 7.2. Параметры, характеризующие шину

Шину образует набор линий связи. Линия связи — это физическая среда, обеспечивающая передачу сигналов, чаще всего представляющих двоичные цифры 1 и 0. По линии может пересылаться развернутая во времени последовательность таких сигналов. При совместном использовании несколько линий могут обеспечить одновременную (параллельную) передачу двоичных чисел. Физически линиями связи могут быть провода, полоски проводящего материала на монтажной плате, оптоволокно, радио и инфракрасные каналы.

Операции на шине называют *транзакциями*. Основные виды транзакций — *транзакции чтения* и *транзакции записи*. Если в обмене участвует устройство ввода/вывода, можно говорить о *транзакциях ввода* и *вывода*, по сути эквивалентных транзакциям чтения и записи соответственно. Шинная транзакция включает в себя две составляющих: посылку адреса и прием (или посылку) данных.

Когда два устройства обмениваются информацией по шине, одно из них должно инициировать обмен и управлять им. Такого рода устройства будем называть *ведущими* (bus master). В компьютерной терминологии «ведущий» — это любое устройство, способное взять на себя владение шиной и управлять пересылкой данных. Ведущий не обязательно использует данные сам. Он, например, может захватить управление шиной в интересах другого устройства. Устройства, не обладающие возможностями инициирования транзакции, носят название *ведомых* (bus slave). В принципе, к шине может быть подключено несколько потенциальных ведущих, но в любой момент времени активным может быть только один из них: если несколько устройств передают информацию одновременно, их сигналы перекрываются и искажаются. Для предотвращения одновременной активности нескольких ведущих в любой шине предусматривается процедура допуска к управлению шиной только одного из претендентов (арбитраж). В то же время некоторые шины

допускают широковещательный режим записи, когда информация одного ведущего передается сразу нескольким ведомым (здесь арбитраж не требуется). Сигнал, направленный одним устройством, доступен всем остальным устройствам, подключенным к шине.

Английский эквивалент термина «шина» — «bus» — восходит к латинскому слову *omnibus*, означающему «для всего». Этим стремятся подчеркнуть, что шина ведет себя как магистраль, способная обеспечить всевозможные виды трафика.

В данной главе рассматриваются только общие вопросы, касающиеся организации, функционирования и применения шин, без ориентации на конкретные реализации.

Типы шин

Важным параметром, определяющим характеристики шины, может служить ее целевое назначение. По этому параметру можно выделить:

- шины «процессор-память»;
- шины ввода/вывода;
- системные шины.

Шины «процессор-память»

Шины «процессор-память» обеспечивают непосредственную связь центрального процессора (ЦП) вычислительной машины с основной памятью (ОП). Часто эта задача возлагается на системную шину (см. ниже), однако в плане эффективности значительно выгоднее, если обмен между ЦП и ОП ведется по независимой шине. В современных микропроцессорах шину, связывающую ЦП и ОП, называют *передней* или *первичной шиной* и обозначают аббревиатурой FSB (Front-Side Bus). Интенсивный обмен информацией между процессором и основной памятью требует, чтобы полоса пропускания шины, то есть количество информации, проходящей по шине в единицу времени, была наибольшей. К группе шин «процессор-память» можно отнести также шину, связывающую процессор с кэш-памятью второго уровня (L2), известную как *тыльная* или *вторичная шина* — BSB (Back-Side Bus). BSB позволяет вести обмен с большей скоростью, чем FSB, что отвечает возможностям более скоростной кэш-памяти. Архитектура с распределением функций связи ЦП с памятью между шинами FSB и BSB известна как архитектура DIB (Dual Independent Bus — двойная независимая шина). Наличие двух шин позволяет одновременно обращаться к ОП и L2, тем самым увеличивая общую производительность ВМ. Впервые архитектура DIB была использована в микропроцессоре Pentium II. Перечисленные варианты связи процессора с памятью иллюстрирует рис. 7.3.

Поскольку в фон-неймановских машинах именно обмен между процессором и памятью во многом определяет быстродействие ВМ, разработчики уделяют связи ЦП с памятью особое внимание. Для обеспечения максимальной пропускной способности шины «процессор-память» всегда проектируются с учетом особенностей организации системы памяти, а длина шины делается по возможности минимальной.

Рис. 7.3. Шины «процессор-память»

Шина ввода/вывода

Шина ввода/вывода служит для соединения процессора (памяти) с устройствами ввода/вывода (УВВ). Учитывая разнообразие таких устройств, шины ввода/вывода унифицируются и стандартизируются. Связи с большинством УВВ (но не с видеосистемами) не требуют от шины высокой пропускной способности. При проектировании шин ввода/вывода в учет берутся стоимость конструктива и соединительных разъемов. Такие шины содержат меньше линий по сравнению с вариантом «процессор-память», но длина линий может быть весьма большой.

Системная шина

С целью снижения стоимости некоторые ВМ имеют общую шину для памяти и устройств ввода/вывода. Такая шина часто называется системной. *Системная шина* служит для физического и логического объединения всех устройств ВМ. Поскольку основные устройства машины, как правило, размещаются на общей монтажной плате, системную шину часто называют объединительной шиной (backplane bus), хотя эти термины нельзя считать строго эквивалентными.

Системная шина обычно состоит из нескольких сотен линий, которые можно подразделить на три функциональных группы (рис. 7.4): шину данных, шину адреса и шину управления. К последней обычно относят также линии для подачи питающего напряжения на подключаемые к системной шине устройства.

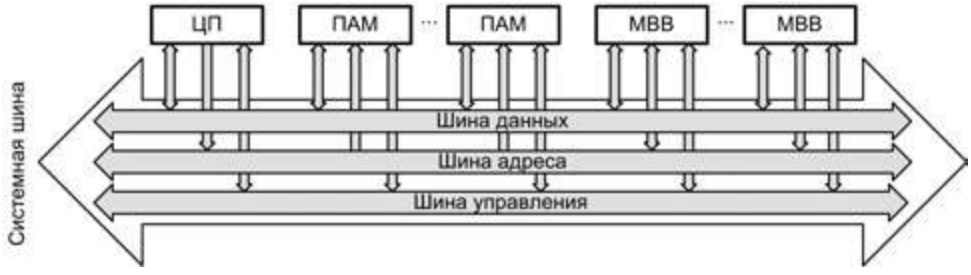


Рис. 7.4. Структура системной шины

Любая транзакция на системной шине начинается с выставления ведущим устройством адресной информации. Адрес позволяет выбрать ведомое устройство и уста-

новить соединение между ним и ведущим. Для передачи адреса используется часть сигнальных линий шины, совокупность которых часто называют *шиной адреса* (ША).

На ША могут выдаваться адреса ячеек памяти, номера регистров ЦП, адреса портов ввода/вывода и т. п. Многообразие видов адресов предполагает наличие дополнительной информации, уточняющей вид адреса, используемый в данной транзакции. Такая информация может косвенно содержаться в самом адресе, но чаще передается по специальным управляющим линиям.

Разнообразной может быть и структура адреса. Так, в адресе старшие биты могут указывать на один из модулей основной памяти, в то время как младшие биты определяют ячейку внутри этого модуля.

В некоторых шинах предусмотрены адреса специального вида, обеспечивающие одновременный выбор определенной группы ведомых либо всех ведомых сразу (*broadcast*). Эта возможность обычно практикуется в транзакциях записи (от ведущего к ведомым), однако существует также специальный вид транзакции чтения (одновременно от нескольких ведомых общему ведущему). Английское название такой транзакции чтения *broadcall* можно перевести как «*широковещательный опрос*». Информация, возвращаемая ведущему, представляет собой результат побитового логического сложения данных, поступивших от всех адресуемых ведомых.

Число сигнальных линий, выделенных для передачи адреса (*ширина шины адреса*), определяет максимально возможный размер адресного пространства. Это одна из базовых характеристик шины, поскольку от нее зависит потенциальная емкость адресуемой памяти и число обслуживаемых портов ввода/вывода. В современных микропроцессорах шина адреса состоит из 36 сигнальных линий, при этом адресное пространство составляет 64 Гбайт. В перспективных разработках предусматривается расширение ША до 40 линий.

Совокупность линий, служащих для пересылки данных между модулями системы, называют *шиной данных* (ШД). Важнейшие характеристики ШД — ширина и пропускная способность.

Ширина шины данных определяется количеством битов информации, которое может быть передано по шине за одну транзакцию (*цикл шины*). Цикл шины следует отличать от периода тактовых импульсов — одна транзакция на шине может занимать несколько тактовых периодов. В середине 1970-х годов типовая ширина шины данных составляла 8 битов. В наше время это обычно 32, 64 или 128 битов. В любом случае ширину ШД выбирают кратной целому числу байтов, причем это число, как правило, представляет собой целую степень числа 2.

Элемент данных, задействующий всю ширину ШД, принято называть *словом*, хотя в архитектуре некоторых ВМ понятие «слово» трактуется по-другому, то есть слово может иметь разрядность, не совпадающую с шириной ШД.

В большинстве шин используются адреса, позволяющие указать отдельный байт слова. Это свойство оказывается полезным, когда желательно изменить в памяти лишь часть полного слова.

При передаче по ШД части слова пересылка обычно производится по тем же сигнальным линиям, что и в случае пересылки полного слова, однако в ряде шин «урезанное» слово передается по младшим линиям ШД. Последний вариант может оказаться более удобным при последующем расширении шины данных, поскольку в этом случае сохраняется преемственность со «старой» шиной.

Ширина шины данных существенно влияет на производительность ВМ. Так, если ширина шины данных вдвое меньше длины команды, ЦП в течение каждого цикла команды вынужден осуществлять доступ к памяти дважды.

Если адрес и данные в системной шине передаются по независимым (выделенным) сигнальным линиям, то ширина ША и ШД обычно выбирается независимо. Наиболее частые комбинации: 16–8, 16–16, 20–8, 20–16, 24–32, 32–32 и 36–64.

В некоторых ВМ адрес и данные пересылаются по одним и тем же линиям, но в разных тактах цикла шины. Этот прием называется *временным мультиплексированием*. В этом случае линии адреса и данных объединены в единую *мультиплексируемую шину адреса/данных*. Такая шина функционирует в режиме разделения времени, поскольку цикл шины разбит на временной интервал для передачи адреса и временной интервал для передачи данных. Структура мультиплексируемой шины адреса/данных показана на рис. 7.5.

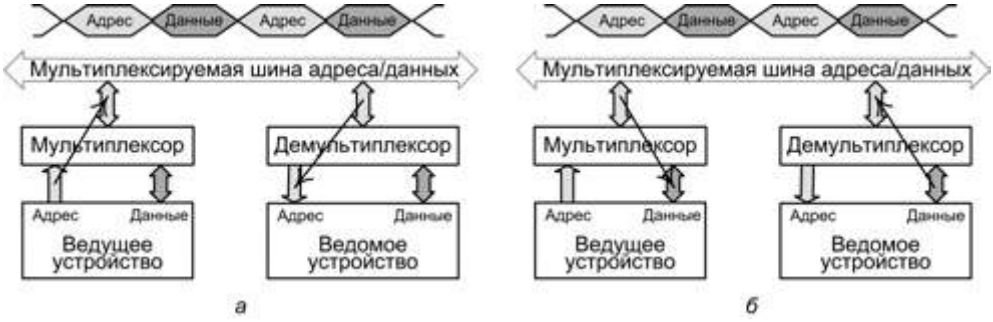


Рис. 7.5. Мультиплексирование адреса и данных: *а* — передача адреса; *б* — передача данных

Мультиплексирование адресов и данных предполагает наличие мультиплексора на одном конце тракта пересылки информации и демультиплексора на его другом конце. Мультиплексоры и демультиплексоры играют роль коммутирующих устройств. Они обеспечивают перенаправление информации с одного из своих входов на общий выход (мультиплексор) либо с общего входа на один из выходов (демультиплексор).

Мультиплексирование позволяет сократить общее число линий, но требует усложнения логики связи с шиной. Кроме того, оно ведет к потенциальному снижению производительности, поскольку исключает возможность параллельной передачи адресов и данных, что можно было бы использовать в транзакциях записи, одновременно выставляя на ША адрес, а на ШД — записываемое слово.

Примером применения мультиплексируемой шины адреса/данных может служить шина Futurebus+.

Помимо трактов пересылки адреса и данных, неотъемлемым атрибутом любой шины являются линии, по которым передается управляющая информация, а также информация о состоянии участвующих в транзакции устройств. Совокупность таких линий принято называть *шиной управления* (ШУ). Сигнальные линии, входящие в ШУ, можно условно разделить на несколько групп.

Первую группу образуют линии, по которым пересылаются *сигналы управления транзакциями*, то есть сигналы, определяющие:

- тип выполняемой транзакции (чтение или запись);
- количество байтов, передаваемых по шине данных, и если пересылается часть слова, то номера байтов;
- какой тип адреса выдан на шину адреса;
- какой протокол передачи должен быть применен.

На перечисленные цели обычно отводится от двух до восьми сигнальных линий.

Ко второй группе относят линии передачи *информации состояния* (*статуса*). В эту группу входят от одной до четырех линий, по которым ведомое устройство может информировать ведущего о своем состоянии или передать код возникшей ошибки.

Третья группа — *линии арбитража*. Вопросы арбитража рассматриваются несколько позже. Пока отметим лишь, что арбитраж необходим для выбора одного из нескольких ведущих, одновременно претендующих на доступ к шине. Число линий арбитража в разных шинах варьируется от 3 до 11.

Четвертую группу образуют *линии прерывания*. По этим линиям передаются запросы на обслуживание, посылаемые от ведомых устройств к ведущему. Под собственно запросы обычно отводятся одна или две линии, однако при одновременном возникновении запросов от нескольких ведомых возникает проблема арбитража, для чего могут понадобиться дополнительные линии, если только с этой целью не используются линии третьей группы.

Пятая группа — линии для организации *последовательных локальных сетей*. Наличие от 1 до 4 таких линий стало общепринятой практикой в современных шинах. Обусловлено это тем, что последовательная передача данных протекает значительно медленнее, чем параллельная, и сети значительно выгоднее строить без загрузки быстрых линий основных шин адреса и данных. Кроме того, шины этой группы могут быть использованы как полноценный, хотя и медленный, избыточный тракт для замены ША и ШД в случае их отказа. Иногда шины пятой группы назначаются для реализации специальных функций, таких, например, как обработка прерываний или сортировка приоритетов задач.

В некоторых ШУ имеется **шестая группа** сигнальных линий — от 4 до 5 *линий позиционного кода*, подсоединяемых к специальным выводам разъема. С помощью перемычек на этих выводах можно задать уникальный позиционный код разъема на материнской плате или вставленной в этот разъем дочерней платы. Такой код может быть использован для индивидуальной инициализации каждой отдельной платы при включении или перезапуске системы.

Наконец, в каждой шине обязательно присутствуют линии **седьмой группы**, которая по сути является одной из важнейших. Это группа линий *тактирования и синхронизации*. При проектировании шины таким линиям уделяется особое внимание. В состав группы, в зависимости от протокола шины (синхронный или асинхронный), входят от двух до шести линий.

В довершение необходимо упомянуть линии для подвода питающего напряжения и линии заземления.

Функционирование системной шины можно описать следующим образом. Если один из модулей хочет передать данные в другой, он должен выполнить два действия: получить в свое распоряжение шину и передать по ней данные. Если какой-то модуль хочет получить данные от другого модуля, он должен получить доступ к шине и с помощью соответствующих линий управления и адреса передать в другой модуль запрос. Далее он должен ожидать, пока модуль, получивший запрос, пошлет данные.

Физически системная шина представляет собой совокупность параллельных электрических проводников. Этими проводниками служат металлические полоски на печатной плате. Шина подводится ко всем модулям, и каждый из них подсоединяется ко всем или некоторым ее линиям. Если ВМ конструктивно выполнена на нескольких платах, то все линии шины выводятся на разъемы, которые затем объединяются проводниками на общем шасси.

Иерархия шин

Если к шине подключено большое число устройств, ее пропускная способность падает, поскольку слишком частая передача прав управления шиной от одного устройства к другому приводит к ощутимым задержкам. По этой причине во многих ВМ предпочтение отдается использованию нескольких шин, образующих определенную иерархию. Сначала рассмотрим ВМ с одной шиной.

Вычислительная машина с одной шиной

В системах соединений с одной шиной имеется одна системная шина, обеспечивающая обмен информацией между всеми устройствами ВМ (рис. 7.6, *а*).

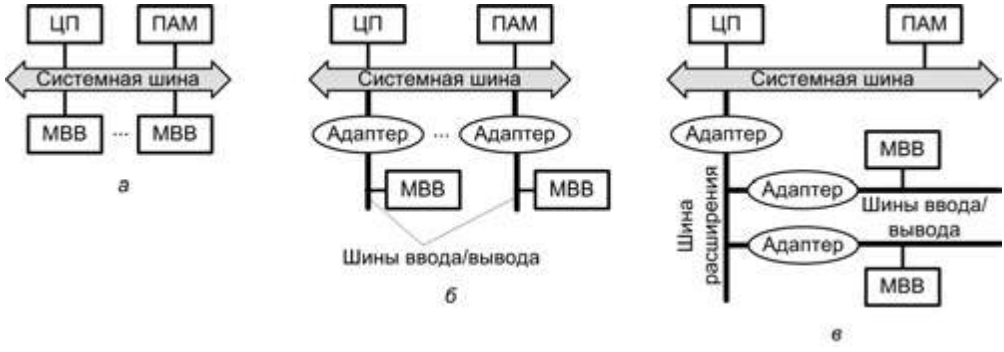


Рис. 7.6. Система соединений: *а* — с одной шиной; *б* — с двумя видами шин; *в* — с тремя видами шин

Для такого подхода характерны простота и низкая стоимость. Однако одношинная организация не в состоянии обеспечить высокие интенсивность и скорость транзакций, в силу чего становится «узким местом» ВМ.

Вычислительная машина с двумя видами шин

Хотя контроллеры модулей ввода/вывода (МВВ) могут быть подсоединены непосредственно к системной шине, больший эффект достигается применением одной или нескольких шин ввода/вывода (рис. 7.6, *б*). МВВ подключаются к шинам ввода/вывода, которые берут на себя основной трафик, не связанный с выходом на процессор или память. *Адаптеры шин* обеспечивают буферизацию данных при их пересылке между системной шиной и контроллерами МВВ. Это позволяет ВМ поддерживать работу множества модулей ввода/вывода и одновременно «развязать» обмен информацией по тракту процессор-память и обмен информацией с МВВ.

Подобная схема существенно снижает нагрузку на скоростную шину «процессор-память» и способствует повышению общей производительности ВМ.

Вычислительная машина с тремя видами шин

Для подключения быстродействующих периферийных устройств в систему шин может быть добавлена высокоскоростная шина расширения (рис. 7.6, *в*).

Шины ввода/вывода подключаются к шине расширения, а уже с нее через адаптер к системной шине, что еще более снижает нагрузку на последнюю. Такую организацию шин называют *архитектурой с «пристройкой»* (mezzanine architecture).

Арбитраж шин

В реальных системах на роль ведущего вправе одновременно претендовать сразу несколько из подключенных к шине устройств, однако управлять шиной в каждый момент времени может только одно из них. Чтобы исключить конфликты, шина должна предусматривать определенные механизмы арбитража запросов и правила предоставления шины одному из запросивших устройств. Решение обычно принимается на основе приоритетов претендентов: шина предоставляется устройству с наивысшим приоритетом среди запросивших.

Алгоритмы арбитража

Каждому потенциальному ведущему присваивается определенный уровень приоритета, который может оставаться неизменным (*статический* или *фиксированный приоритет*) либо изменяться по какому-либо алгоритму (*динамический приоритет*). Кроме того, существуют алгоритмы, в которых понятие уровня приоритета не имеет смысла, например, если все устройства имеют одинаковый приоритет.

В системах со статическим приоритетом предполагается, что каждому ведущему изначально присваивается отличный от других уровень приоритета, который в процессе работы остается неизменным. Основной недостаток систем со статическими приоритетами в том, что устройства, имеющие высокий приоритет, в состоянии полностью блокировать доступ к шине устройств с меньшим уровнем приоритета.

Алгоритмы арбитража на основе динамических приоритетов

Системы с динамическими приоритетами дают шанс каждому из запросивших устройств рано или поздно получить право на управление шиной, то есть в таких системах реализуется принцип равнодоступности. Наибольшее распространение получили следующие алгоритмы динамического изменения приоритетов:

- простая циклическая смена приоритетов;
- циклическая смена приоритетов с учетом последнего запроса;
- смена приоритетов по случайному закону;
- алгоритм наиболее давнего использования.

В алгоритме *простой циклической смены приоритетов* каждое устройство имеет уникальный уровень приоритета в диапазоне от минимального до максимального. После каждого цикла арбитража приоритеты всех устройств циклически понижаются на один уровень, при этом устройство, имевшее низший уровень приоритета, получает наивысший приоритет (рис. 7.7).

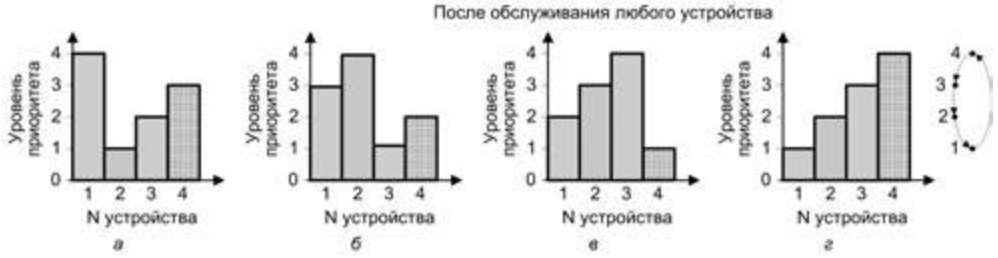


Рис. 7.7. Пример арбитража по алгоритму простой циклической смены приоритетов: *а* — исходные уровни приоритета; *б, в, г* — уровни приоритетов после очередного цикла арбитража

В алгоритме *циклической смены приоритетов с учетом последнего запроса* (RDC – Rotating Daisy Chain) функции арбитража распределены между всеми устройствами, цепи арбитража которых соединены в кольцо. После обработки очередного запроса обслуженному устройству назначается низший уровень приоритета, и оно становится арбитром для следующего цикла шины. Остальным устройствам назначаются новые приоритеты, в зависимости от их удаленности в кольце от устройства, выполняющего в данный момент роль арбитра: следующее устройство в кольце получает наивысший приоритет, а остальным устройствам приоритеты назначаются в убывающем порядке, согласно их следованию в кольце (рис. 7.8). Хотя оба алгоритма циклической смены приоритетов дают шанс каждому ведущему устройству получить шину в свое распоряжение, большее распространение получил второй алгоритм.

При *смене приоритетов по случайному закону* после очередного цикла арбитража с помощью генератора псевдослучайных чисел каждому ведущему присваивается случайное, но отличное от других значение уровня приоритета.

В *алгоритме наиболее давнего использования* (LRU, Least Recently Used) после каждого цикла арбитража наивысший приоритет присваивается ведущему, дольше других не использовавшему шину.

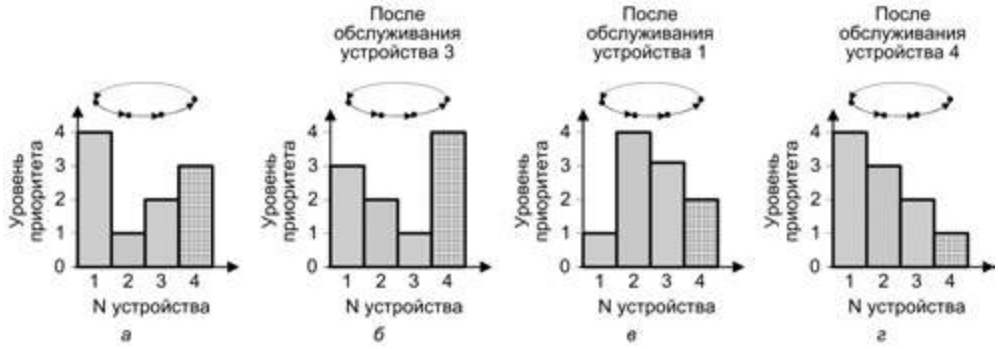


Рис. 7.8. Пример арбитража по алгоритму циклической смены приоритетов с учетом последнего запроса: *а* — исходные уровни приоритета; *б*, *в*, *г* — уровни приоритетов после очередного цикла арбитража с учетом запросившего устройства

Алгоритм арбитража на основе фиксированного кванта времени

Согласно данному алгоритму все устройства имеют одинаковый приоритет. Каждому устройству на циклической основе предоставляется квант времени, в течение которого устройство получает право на управление шиной. Если устройство в этот момент не нуждается в шине, выделенный квант времени остается неиспользованным. Такой метод наиболее подходит для шин с синхронным протоколом.

Алгоритм арбитража на основе очереди

В алгоритме очереди (FIFO — First In — First Out) запросы обслуживаются в порядке очереди, образовавшейся к моменту начала цикла арбитража. Сначала обслуживается первый запрос в очереди, то есть запрос, поступивший раньше остальных. Аппаратурная реализация алгоритма связана с определенными сложностями, поэтому используется он редко.

Схемы арбитража

Арбитраж запросов на управление шиной может быть организован по централизованной или децентрализованной схеме. Выбор конкретной схемы зависит от требований к производительности и стоимостных ограничений.

Централизованный арбитраж

При *централизованном арбитраже* в системе имеется специальное устройство — *центральный арбитр*, — ответственное за предоставление доступа к шине только одному из запросивших ведущих. Это устройство, называемое иногда *центральным контроллером шины*, может быть самостоятельным модулем или частью ЦП. Наличие на шине только одного арбитра означает, что в централизованной схеме имеется единственная точка отказа. В зависимости от того, каким образом

ведущие устройства подключены к центральному арбитру, возможные схемы централизованного арбитража можно подразделить на параллельные и последовательные.

В параллельном варианте центральный арбитр связан с каждым потенциальным ведущим индивидуальными двухпроводными трактами. Поскольку запросы к центральному арбитру могут поступать независимо и параллельно, данный вид арбитража называют *централизованным параллельным арбитражем* или *централизованным арбитражем независимых запросов*.

Идею централизованного параллельного арбитража на примере восьми ведущих устройств иллюстрирует рис. 7.9.

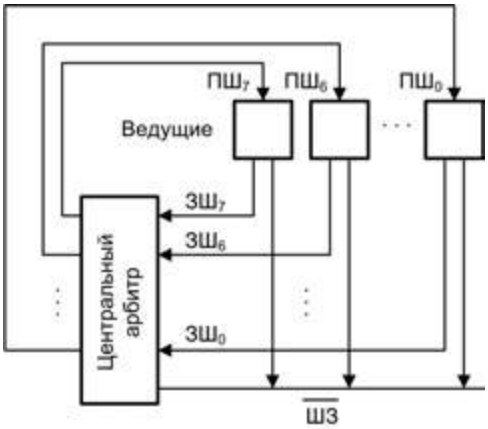


Рис. 7.9. Централизованный параллельный арбитраж

Здесь и далее под «текущим ведущим» будем понимать ведущее устройство, управляющее шиной в момент поступления нового запроса. Устройство, выставившее запрос на управление шиной, будем называть «запросившим ведущим». Сигналы запроса шины (ЗШ) поступают на вход центрального арбитра по индивидуальным линиям. Ведущему с номером i, который был выбран арбитром, также по индивидуальной линии возвращается сигнал предоставления шины ($ПШ_i$), но реально занять шину новый ведущий сможет лишь после того, как текущий ведущий (пусть он имеет номер j) снимет сигнал занятия шины (ШЗ). Текущий ведущий должен сохранять сигналы ШЗ и $ЗШ_j$ активными в течение всего времени, пока он использует шину. Получив запрос от ведущего, приоритет которого выше, чем у текущего ведущего, арбитр снимает сигнал $ПШ_j$ на входе текущего ведущего и выдает сигнал предоставления шины $ПШ_i$ запросившему ведущему. В свою очередь, текущий ведущий, обнаружив, что центральный арбитр убрал с его входа сигнал $ПШ_j$, снимает свои сигналы ШЗ и $ЗШ_j$, после чего запросивший ведущий может перенять управление шиной. Если в момент пропадания сигнала ПШ на шине происходит передача информации, текущий ведущий сначала завершает передачу и лишь после этого снимает свои сигналы. Логика выбора одного из запрашивающих ведущих обычно реализуется аппаратными средствами.

При большом числе источников запроса центральный арбитр может строиться по схеме двухуровневого параллельного арбитража. Все возможные запросы разбиваются на группы, и каждая группа анализируется своим арбитром первого уровня, который выбирает запрос, имеющий в данной группе наивысший приоритет. Арбитр второго уровня отдает предпочтение тому арбитру первого уровня, у которого выбранный запрос имеет наиболее высокий приоритет. При необходимости могут вводиться дополнительные уровни арбитража.

Схема централизованного параллельного арбитража очень гибка и обладает высоким быстродействием, но в ней затруднено подключение дополнительных устройств. Обычно максимальное число ведущих при параллельном арбитраже не превышает восьми.

Второй вид централизованного арбитража известен как *централизованный последовательный арбитраж*. В последовательных схемах для выделения запроса с наивысшим приоритетом используется один из сигналов, поочередно проходящий через цепочку ведущих, чем и объясняется другое название — *цепочечный* или *гирляндный арбитраж*. В дальнейшем будем полагать, что уровни приоритета ведущих устройств в цепочке понижаются слева направо.

В зависимости от того, какой из сигналов используется для целей арбитража, различают три основных типа схем цепочечного арбитража: с цепочкой для сигнала предоставления шины (ПШ), с цепочкой для сигнала запроса шины (ЗШ) и с цепочкой для дополнительного сигнала разрешения (РШ). Наиболее распространена схема *цепочки для сигнала* ПШ (рис. 7.10).

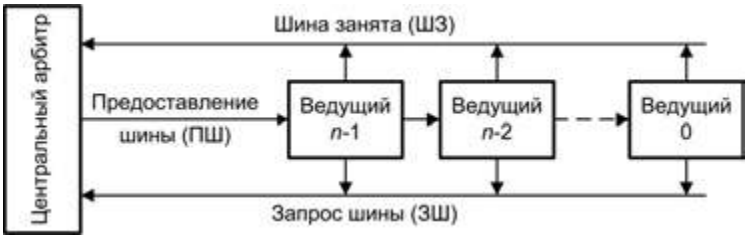


Рис. 7.10. Централизованный последовательный арбитраж с цепочкой для сигнала предоставления шины

Запросы от ведущих объединяются на линии запроса шины по схеме «ИЛИ». Аналогично организована и линия, сигнализирующая о том, что шина в данный момент занята одним из ведущих. Когда один или несколько ведущих выставляют запросы, эти запросы транслируются на вход центрального арбитра. Получив сигнал ЗШ, арбитр анализирует состояние линии занятия шины, и если шина свободна, формирует сигнал предоставления шины. Сигнал ПШ последовательно переходит по цепочке от одного ведущего к другому. Если устройство, на которое поступил сигнал ПШ, не запрашивало шину, оно просто пропускает сигнал дальше по цепочке. Когда ПШ достигнет самого левого из запросивших ведущих, последний блокирует дальнейшее распространение сигнала по цепочке и берет на себя управление шиной.

Еще раз отметим, что очередной ведущий не может приступить к управлению шиной до момента ее освобождения. Центральный арбитр не должен формировать сигнал ПШ вплоть до этого момента.

Цепочечная реализация предполагает статическое распределение приоритетов. Наивысший приоритет имеет ближайшее к арбитру ведущее устройство. Далее приоритеты ведущих в цепочке последовательно понижаются.

Основное достоинство цепочечного арбитража — простота реализации и малое количество используемых линий. Последовательные схемы арбитража позволяют легко наращивать число устройств, подключаемых к шине. Схема, однако, обладает и существенными недостатками. Прежде всего, последовательное прохождение сигнала по цепочке замедляет арбитраж, причем время арбитража растет пропорционально длине цепочки. Кроме того, статическое распределение приоритетов может привести к полному блокированию устройств с низким уровнем приоритета, расположенных ближе к концу цепочки.

Децентрализованный арбитраж

При *децентрализованном* или *распределенном арбитраже* единый арбитр отсутствует. Вместо этого каждый ведущий содержит блок управления доступом к шине (контроллер шины), и при совместном использовании шины такие блоки взаимодействуют друг с другом, разделяя между собой ответственность за доступ к шине.

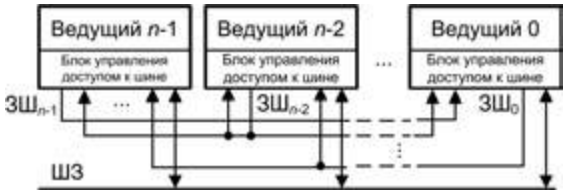


Рис. 7.11. Иллюстрация децентрализованного арбитража

Идею децентрализованного арбитража иллюстрирует рис. 7.11. Каждый ведущий имеет уникальный уровень приоритета и обладает собственным контроллером шины, способным формировать сигналы предоставления и занятия шины. Сигналы запроса от любого ведущего поступают на входы всех остальных ведущих. Логика арбитража реализуется в контроллере шины каждого ведущего.

Для большинства шин все-таки более характерна другая организация децентрализованного арбитража. Такие схемы предполагают наличие в составе шины группы арбитражных линий, организованных по схеме «ИЛИ». Это позволяет любому ведущему видеть сигналы, выставленные остальными устройствами. Каждому ведущему присваивается уникальный номер, совпадающий с кодом уровня приоритета данного ведущего. Запрашивающие шину устройства выдают на арбитражные линии свой номер. Каждый из запросивших ведущих, обнаружив на арбитражных линиях номер устройства с более высоким приоритетом, снимает с этих линий младшие биты своего номера. В конце концов, на арбитражных линиях остается только номер устройства, обладающего наиболее высоким приоритетом. Победителем в

процедуре арбитража становится ведущий, опознавший на арбитражных линиях свой номер. Подобная схема известна также как *распределенный арбитраж с самостоятельным выбором*, поскольку ведущий сам определяет, стал ли он победителем в арбитраже, то есть выбирает себя самостоятельно. Идея подобного арбитража была предложена М. Таубом (Matthew Taub) в 1975 году.

Вариации рассмотренной схемы широко используются в таких шинах, как Futurebus, NuBus, MultiBus II, Fastbus.

В целом схемы децентрализованного арбитража потенциально более надежны, поскольку отказ контроллера шины в одном из ведущих не нарушает работу с шиной других, подключенных к ней устройств. Тем не менее должны быть предусмотрены средства для обнаружения неисправных контроллеров, например, на основе таймаута. Основной недостаток децентрализованных схем — в относительной сложности логики арбитража, которая должна быть реализована в аппаратуре каждого ведущего.

В некоторых ВМ применяют комбинированные последовательно-параллельные схемы арбитража, в какой-то мере сочетающие достоинства обоих методов. Здесь все ведущие разбиваются на группы. Арбитраж внутри группы ведется по последовательной схеме, а между группами — по параллельной.

Ограничение времени управления шиной

Вне зависимости от принятой модели арбитража должна быть также продумана стратегия ограничения времени контроля над шиной. Одним из вариантов может быть разрешение ведущему занимать шину в течение одного цикла шины, с предоставлением ему возможности конкуренции за шину в последующих циклах. Другим вариантом является принудительный захват контроля над шиной устройством с более высоким уровнем приоритета, при сохранении восприимчивости текущего ведущего к запросам на освобождение шины от устройств с меньшим уровнем приоритета.

Протокол шины

Хотя все биты адреса ведомого выдаются ведущим устройством одновременно (параллельно), это совсем не гарантирует их одновременного поступления к получателю. Отдельные биты адреса могут преодолевать более длинный путь, другие — предварительно должны пройти через аппаратуру преобразования адресов. Кроме того, отличия есть и в физических характеристиках отдельных сигнальных линий. Рассмотренная ситуация называется *перекосом сигналов*. Прежде чем реагировать на поступивший адрес, все ведомые должны знать, с какого момента его можно считать достоверным.

Метод, выбираемый проектировщиками шин для информирования о достоверности адреса, данных, управляющей информации и информации состояния, называется *протоколом шины*. Используется два основных класса протоколов — синхронный и асинхронный. В синхронном протоколе все сигналы «привязаны» к

импульсам единого генератора тактовых импульсов (ГТИ). В асинхронном протоколе для каждой группы линий шины формируется свой сигнал подтверждения достоверности. Хотя в каждом из протоколов можно найти как синхронные, так и асинхронные аспекты, различия все же весьма существенны.

Синхронный протокол

В синхронных шинах имеется центральный генератор тактовых импульсов (ГТИ), к импульсам которого «привязаны» все события на шине. Тактовые импульсы (ТИ) распространяются по специальной сигнальной линии и представляют собой регулярную последовательность чередующихся единиц и нулей. Один период такой последовательности называется *тактовым периодом шины*. Именно он определяет минимальный квант времени на шине (временной слот). Все подключенные к шине устройства могут считывать состояние тактовой линии, и все события на шине отсчитываются от начала тактового периода. Изменение управляющих сигналов на шине обычно совпадает с передним или задним фронтом тактового импульса, иными словами, момент смены состояния на синхронной шине известен заранее и определяется тактовыми импульсами.

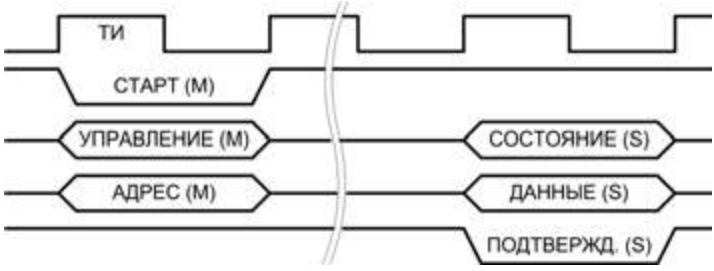


Рис. 7.12. Чтение на синхронной шине

На рис. 7.12 показана транзакция чтения с использованием простого синхронного протокола шины (буквой «М» обозначены сигналы ведущего, а буквой «S» — ведомого). Моменты изменения сигналов на шине определяет нарастающий фронт тактового импульса. Задний фронт ТИ служит для указания момента, когда сигналы можно считать достоверными. Стартовый сигнал отмечает присутствие на линиях шины адресной или управляющей информации. Когда ведомое устройство распознает свой адрес и находит затребованные данные, оно помещает эти данные и информацию о состоянии на шину и сигнализирует об их присутствии на шине сигналом подтверждения.

Операция записи выглядит сходно. Отличие состоит в том, что данные выдаются ведущим в тактовом периоде, следующем за тактовым периодом выставления адреса, и остаются на шине до отправки ведомым сигнала подтверждения и информации состояния.

Отметим, что синхронные протоколы требуют меньше сигнальных линий, проще для понимания, реализации и тестирования. С другой стороны, они менее гибки, по-

скольку привязаны к конкретной максимальной тактовой частоте и, следовательно, к конкретному уровню технологии.

По синхронному протоколу обычно работают шины «процессор-память».

Асинхронный протокол

Синхронная передача быстра, но в ряде ситуаций не подходит для использования. В частности, в синхронном протоколе ведущий не знает, корректно ли ответил ведомый, — возможно, ведомое устройство было не в состоянии удовлетворить запрос на нужные данные. Более того, ведущий должен работать со скоростью самого медленного из участвующих в пересылке данных ведомых. Обе проблемы успешно решаются в асинхронном протоколе шины.

В асинхронном протоколе начало очередного события на шине определяется не тактовым импульсом, а предшествующим событием и следует непосредственно за этим событием. Помещение ведущим устройством на шину любой информации сопровождается соответствующим синхронизирующим сигналом — *стробом*. В свою очередь, ведомое устройство для поддержания асинхронного протокола может вырабатывать свои синхронизирующие сигналы, называемые *квитирующими сигналами* (handshakes) или *подтверждениями сообщения* (acknowledges).

На рис. 7.13 показана асинхронная операция чтения.

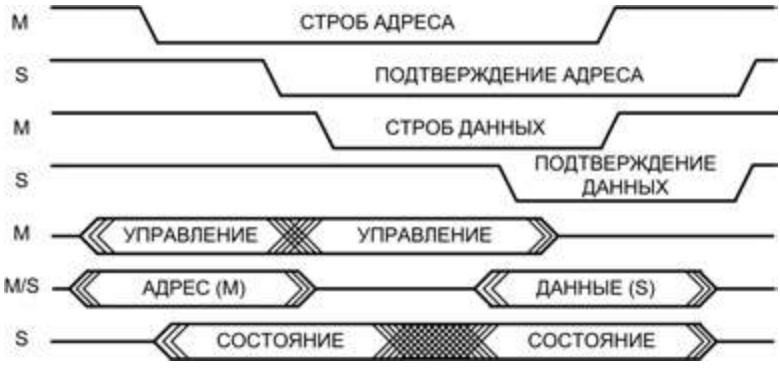


Рис. 7.13. Чтение на асинхронной шине

Сначала ведущее устройство выставляет на шину адрес и управляющие сигналы и выжидает время перекоса сигналов, после чего выдает строб адреса, подтверждающий достоверность информации. Ведомые следят за адресной шиной, чтобы определить, должны ли они реагировать. Ведомый, распознавший на адресной шине свой адрес, отвечает информацией состояния, сопровождаемой сигналом подтверждения адреса. Когда ведущий обнаруживает подтверждение адреса, он знает, что соединение установлено, и готов к анализу информации состояния. Присутствие адреса на ША далее не требуется, поскольку у ведомого уже имеется копия полного адреса либо нужной его части.

Далее ведущий меняет управляющую информацию, выжидает время перекоса и выдает строб данных. Если это происходит в транзакции записи, то ведущий

одновременно с управляющей информацией выставляет на шину записываемые данные. В рассматриваемом случае управляющая информация извещает ведомое устройство, что это чтение. Когда ведомый подготовит затребованные данные, он выдает их на шину совместно с новой информацией о состоянии и формирует сигнал подтверждения данных. Обнаружив сигнал подтверждения данных, ведущее устройство читает данные с шины и снимает строб данных, чтобы показать, что действия с данными завершены. В нашем примере ведущий снимает также и строб адреса. В более сложных вариантах транзакций строб адреса может оставаться на шине для поддержания соединения в течение нескольких циклов данных. При исчезновении строба данных ведомый снимает с шины данные и информацию состояния, а также сигнал подтверждения данных, переводя шину в свободное состояние.

Как видно из приведенного описания, в цикле асинхронной шины для подтверждения успешности транзакции используется двунаправленный обмен сигналами управления. Такая процедура носит название *квитирования установления связи* или *рукопожатия* (handshake). В рассмотренном варианте процедуры ни один шаг в передаче данных не может начаться, пока не завершен предыдущий шаг. Такое квитирование известно как *квитирование с полной взаимоблокировкой* (fully interlocked handshake).

Как и в синхронных протоколах, в любой асинхронной транзакции присутствуют элементы чтения и записи: по отношению к управляющей информации выполняется операция записи, а к информации состояния — чтения. Различие проявляется в идеологии координации этих операций, установления их зависимости друг от друга.

Скорость асинхронной пересылки данных диктуется ведомым, поскольку ведущему для продолжения транзакции приходится ждать отклика. Асинхронные протоколы по своей сути являются самосинхронизирующимися, поэтому шину могут совместно использовать устройства с различным быстродействием, построенные на базе как старых, так и новых технологий. Шина автоматически адаптируется к требованиям устройств, обменивающихся информацией в данный момент. Таким образом, с развитием технологий к шине могут быть подсоединены более быстрые устройства, и пользователь сразу ощутит все их преимущества. В отличие от синхронных систем, для ускорения системы с асинхронной шиной не требуется замена на шине старых медленных устройств на быстрые новые. Платой за перечисленные преимущества асинхронного протокола служит некоторое усложнение аппаратуры.

Иногда транзакция на шине не может быть завершена стандартным образом, например, если ведущий из-за программных ошибок обращается к несуществующей ячейке памяти. В этом случае ведомое устройство не отвечает соответствующим подтверждающим сигналом. Чтобы предотвратить бесконечное ожидание в шинах, используется тайм-аут, то есть задается время, спустя которое при отсутствии отклика транзакция принудительно прекращается.

Шины ввода/вывода обычно реализуются как асинхронные.

Методы повышения эффективности шин

Существует несколько приемов, позволяющих повысить производительность шин. К ним, прежде всего, следует отнести пакетный режим, конвейеризацию и расщепление транзакций.

Пакетный режим пересылки информации

Эффективность как выделенных, так и мультиплексируемых шин может быть улучшена, если они функционируют в *блочном* или *пакетном режиме* (burst mode), когда одиночный адресный цикл сопровождается множественными, но не чередующимися циклами чтения или записи данных. Это означает, что пакет данных передается без указания текущего адреса внутри пакета.

При записи в память последовательные элементы блока данных заносятся в соседние ячейки. Так как в пакетном режиме передается адрес только первой ячейки, все последующие адреса генерируются уже в самой памяти путем последовательного увеличения этого начального адреса. Скорость передачи данных в пакетном режиме увеличивается естественным образом за счет уменьшения числа передаваемых адресов. Внутри пакета очередные данные могут передаваться в каждом тактовом периоде шины, длина пакета может достигать 1024 байта. Наиболее частый вариант — пакеты, состоящие из четырех байтов.

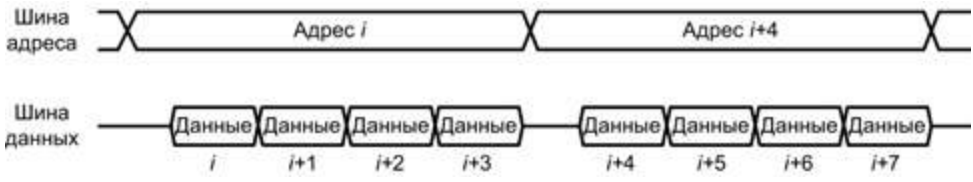


Рис. 7.14. Пакетный режим передачи данных

Концепцию адресации в пакетном режиме на примере пакета, состоящего из четырех элементов данных, иллюстрирует рис. 7.14. По шине адреса передается только адрес ячейки i, а для ячеек $i + 1$, $i + 2$ и $i + 3$ адреса не указываются.

В асинхронных системах пакетный режим позволяет достичь дополнительного эффекта, благодаря сокращению времени, затрачиваемого на прохождение слова от отправителя к приемнику и процедуру подтверждения, а также на внутренние задержки в ведущем и ведомом устройствах и линиях шины. Примером реализации пакетного режима передачи может служить шина Futurebus+.

Конвейеризация транзакций

Одним из способов повышения скорости передачи данных по шине является *конвейеризация транзакций*. Очередной элемент данных может быть отправлен устройством A до того, как устройство B завершит считывание предыдущего элемента. Идею конвейеризации транзакций иллюстрирует рис. 7.15. На рисунке $t_{зд}$ — задержка между моментом выставления данных устройством A и моментом

их появления на шине; $t_{рс}$ — задержка распространения сигнала от устройства A до устройства B; $t_{ст}$ — время, в течение которого данные на входе устройства B должны стабилизироваться, с тем чтобы их можно было однозначно распознать; $t_{уд}$ — интервал, в течение которого информация должна оставаться на шине данных после того, как они были зафиксированы устройством B.

Рис. 7.15. Конвейеризация транзакций чтения

Данные на шине должны оставаться стабильными в течение времени $t_{ст} + t_{уд}$. Только после этого возможна смена элемента данных. Максимальная скорость передачи при конвейеризации определяется выражением $\frac{1}{t_{ст}+t_{уд}}$.

Протокол с расщеплением транзакций

Для увеличения эффективной полосы пропускания шины во многих современных шинах используется протокол *с расщеплением транзакций* (split transaction), известный также как *протокол соединения/разъединения* (connect/disconnect) или *протокол с коммутацией пакетов* (packet-switched). Этот протокол обычно обеспечивает преимущество на транзакциях чтения.

В классическом варианте любая транзакция на шине неразрывна, то есть новая транзакция может начаться только после завершения предыдущей, причем в течение всего периода транзакции шина остается занятой. Протокол с расщеплением транзакций допускает совмещение во времени сразу нескольких транзакций.

В шине с расщеплением транзакций линии адреса и данных обязаны быть независимыми. Каждая транзакция чтения разделяется на две части: адресную транзакцию и транзакцию данных. Считывание данных из памяти начинается с адресной транзакции: выставления ведущим на адресную шину адреса ячейки. С приходом адреса память приступает к относительно длительному процессу поиска и извлечения затребованных данных. По завершении чтения память становится ведущим устройством, запрашивает доступ к шине и направляет считанные данные по шине данных. Фактически, от момента поступления запроса до момента формирования отклика шина остается незанятой и может быть востребована для выполнения других транзакций. В этом и состоит главная идея протокола расщепления транзакций.

Таким образом, на шине с расщеплением транзакции имеют место поток запросов и поток откликов. Часто в системах с расщеплением транзакций контроллер памяти проектируется так, чтобы обеспечить буферизацию множественных запросов.

Случай, когда затребованные данные возвращаются в той же последовательности, в которой поступали запросы, в сущности представляет собой рассмотренную выше конвейеризацию. Шина с расщеплением транзакций зачастую может обеспечивать вариант, при котором ответы на запросы поступают в произвольной последовательности (рис. 7.16). Чтобы не спутать, какому из запросов соответствует информация на шине данных, ее необходимо снабдить признаком (тегом).

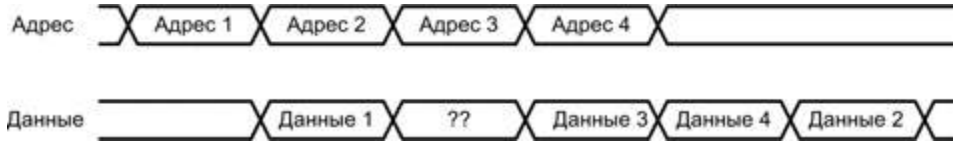


Рис. 7.16. Расщепление транзакций

Хотя протокол с расщеплением транзакций и позволяет более эффективно использовать полосу пропускания шины по сравнению с протоколами, удерживающими шину в течение всей транзакции, он обычно вносит дополнительную задержку из-за необходимости получать два подтверждения — при запросе и при отклике. Кроме того, реализация протокола связана с дополнительными затратами, так как требует, чтобы транзакции были тегированы (помечены) и отслеживались каждым устройством.

Для любой шины с расщеплением транзакций существует предельное значение числа одновременно обслуживаемых запросов.

Ускорение транзакций

Для сокращения времени транзакций, помимо расщепления транзакций, проектировщики обычно прибегают к следующим приемам, связанным с арбитражем шины:

- арбитражу с перекрытием;
- арбитражу с удержанием шины.

Арбитраж с перекрытием (overlapped arbitration) заключается в том, что одновременно с выполнением текущей транзакции производится арбитраж следующей транзакции.

При *арбитраже с удержанием шины* (bus parking) ведущий может удерживать шину и выполнять множество транзакций, пока отсутствуют запросы от других потенциальных ведущих.

В современных шинах обычно сочетаются все вышеперечисленные способы ускорения транзакций.

Увеличение полосы пропускания шины

Совершенствование транзакций — не единственный способ расширения полосы пропускания шины. Среди других вариантов основными, пожалуй, можно считать:

- отказ от мультиплексирования шин адреса и данных;
- увеличение ширины шины данных;
- повышение тактовой частоты шины.

Замена мультиплексируемой шины адреса/данных и переход к выделенным шинам адреса и данных делает возможной одновременную пересылку как адреса, так и данных, то есть позволяют реализовать более эффективные варианты транзакций. Такое решение, однако, является более дорогостоящим из-за необходимости иметь большее число сигнальных линий.

Полоса пропускания шины по своему определению непосредственно зависит от количества параллельно пересылаемой информации — практически прямо пропорциональна ширине шины данных. Несмотря на то что данный способ требует увеличения числа сигнальных линий, многие разработчики ВМ используют в своих машинах достаточно широкие шины данных, например 128 битов.

И наконец, еще один очевидный способ увеличения полосы пропускания, которым широко пользуются проектировщики, — это наращивание тактовой частоты.

Стандартизация шин

Стандартизация шин позволяет разработчикам различных устройств вычислительных машин работать независимо, а пользователям — самостоятельно сформировать нужную конфигурацию ВМ. Часто стандарты разрабатываются специализированными организациями. Такими общепризнанными авторитетами в области стандартизации являются IEEE (Institute of Electrical and Electronics Engineers) — Институт инженеров по электротехнике и электронике) и ANSI (American National Standards Institute) — Национальный институт стандартизации США. Многие стандарты становятся итогом кооперации усилий производителей оборудования для вычислительных машин. Иногда, в силу популярности конкретных машин, реализованные в них решения становятся стандартами де-факто, однако успех таких стандартов во многом определяется их принятием и утверждением в IEEE и ANSI.

Рассматривая вопросы стандартизации шин, следует отдельно говорить о шинах, связывающих основные устройства ВМ (центральный процессор, память, модули ввода/вывода), — шинах «большого» интерфейса, и шинах, служащих для присоединения к МВВ периферийных устройств — шинах «малого» интерфейса. В процессе развития вычислительной техники совершенствовались шины обоих видов, что выражалось в появлении все новых и новых стандартов. Общая тенденция в развитии шин «большого» и «малого» интерфейса — переход от параллельных синхронных шин к последовательным асинхронным шинам. По этой причине ниже основное внимание будет уделено стандартам последовательных шин, наиболее актуальных на момент написания данного учебника.

Шины «большого» интерфейса

В меньшей степени стандартизация касается шин «большого» интерфейса, выполняющих роль системных шин. Большей частью сфера применения каждого отдельного

стандарта ограничена вычислительными машинами какого-то одного производителя. До недавнего времени в качестве стандартных шин «большого» интерфейса превалировали параллельные шины, среди которых наиболее широкое распространение получили шины: VME (Motorola), Multibus II (Intel), ISA (IBM), EISA (IBM) и ряд других. Основные характеристики некоторых распространенных шин, как стандартных, так и претендующих на роль таковых, приведены в табл. 7.1–7.3.

Таблица 7.1. Стандартные системные шины общего применения

Характеристика	**VME**	**Futurebus**	**Multibus II**
Разработчик	Motorola, Philips, Mostek	IEEE	Intel
Ширина шины	128	96	96
Мультиплексирование адреса/данных	Нет	Да	Да
Разрядность адреса, бит	16/24/32/64	32	
Разрядность данных, бит	8/16/32/64	16/32/64/128	32
Вид пересылки	Одиночная или групповая	Одиночная или групповая	Одиночная или групповая
Количество ведущих	Несколько	Несколько	Несколько
Арбитраж	Централизованный	Централизованный или децентрализованный	Децентрализованный
Расщепление транзакций	Нет	Возможно	Возможно
Протокол	Асинхронный	Асинхронный	Синхронный
Тактовая частота, МГц	Нет данных	Нет данных	10
Полоса пропускания при одиночной пересылке, Мбайт/с	25	37	20
Полоса пропускания при групповой пересылке, Мбайт/с	28	95	40/80
Максимальное количество устройств	21	20	21
Максимальная длина шины, м	0,5	0,5	0,5
Стандарт	IEEE 1014	IEEE 896.1	ANSI/IEEE 1296

Таблица 7.2. Системные шины высокопроизводительных серверов

Характеристика	**Summit**	**Challenge**	**XDBus**
Разработчик	HP	SGI	Sun
Мультиплексирование адреса/данных	Нет данных	Нет данных	Да
Разрядность адреса, бит	48	40	Нет данных
Разрядность данных, бит	128/512	256/1024	144/512

Характеристика	Summit	Challenge	XDBus
Вид пересылки	Одиночная или групповая	Одиночная или групповая	Одиночная или групповая
Количество ведущих	Несколько	Несколько	Несколько
Арбитраж	Централизованный	Централизованный	Централизованный
Расщепление транзакций	Есть	Есть	Есть
Протокол	Синхронный	Синхронный	Синхронный
Тактовая частота, МГц	60	48	66
Полоса пропускания при одиночной пересылке, Мбайт/с	60	48	66
Полоса пропускания при групповой пересылке, Мбайт/с	960	1200	1056
Максимальная длина шины, м	0,3	0,3	0,4
Стандарт	Нет	Нет	Нет

Таблица 7.3. Системные шины персональных вычислительных машин

Характеристика	NuBus	ISA 8/16	EISA	FSB Pentium 4
Разработчик	Texas Instruments	IBM	AST, Compaq, Epson, HP, NEC, Olivetti, Tandy, Wyse, Zenith	Intel
Ширина шины	96	62/98	98/100	Нет данных
Мультиплексирование	Да	Нет	Нет	Нет
Разрядность адреса, бит	32	20/24	24/32	36
Разрядность данных, бит	32	8/16	16/32	64/128
Вид пересылки	Одиночная или групповая	Одиночная или групповая	Одиночная или групповая	Одиночная или групповая
Арбитраж	Централизованный	Нет данных	Централизованный	Нет данных
Расщепление транзакций	Нет	Нет данных	Возможно	Да
Количество ведущих	Несколько (ограничено)	Один	Один	Нет данных
Протокол	Синхронный	Синхронный	Синхронный	Синхронный
Тактовая частота, МГц	10	4,77/8,33	8,33	400 (баз.100); 533 баз.133); 800 (ожидается)
Полоса пропускания при одиночной пересылке, Мбайт/с	40	33	33	1060 (133); 3200 (400); 4200 (533)

В последнее время все более популярными становятся последовательные шины «большого» интерфейса. Хотя в плане организации обмена информацией параллельные шины проще, для них, помимо большого числа проводников, характерны и другие проблемы, такие, например, как необходимость осуществлять синхронизацию сигналов, передаваемых по каждому проводнику параллельной шины за один такт. Последовательные шины позволяют объединять множество устройств, используя лишь несколько пар проводов, и по своей организации существенно отличаются от параллельных. В последовательных шинах отсутствуют выделенные линии данных, адреса и управления — все функции приходится выполнять, используя одну или две пары сигнальных проводов. Шинный протокол в последовательных шинах основан на пересылках пакетов — определенным образом организованных цепочек битов. Соединение устройств в последовательных шинах обычно реализовано по системе «точка-точка», что позволяет избавиться от процедуры арбитража.

Для последовательных шин характерны два основных типа передач данных — асинхронный и изохронный.

Асинхронный режим используется при передаче сообщений между двумя устройствами. Одно из устройств посылает запрос, на который второе устройство отвечает положительным (ACK) или отрицательным (NACK) подтверждением. Отрицательное подтверждение формируется, если обнаружена ошибка. В этом случае передачи будут повторяться несколько раз до достижения успеха (получения положительного подтверждения) или до фиксации ошибки.

Изохронные передачи используются для потоковых передач данных в реальном масштабе времени и ведутся широковещательно. Для таких передач резервируется определенная полоса пропускания шины, позволяющая обеспечить допустимую величину задержек, однако доставка сообщения не гарантируется (при обнаружении ошибки повторная передача не производится). Изохронные передачи используются для передачи аудио- и видеотрафика.

Среди последовательных шин «большого» интерфейса рассмотрим наиболее популярные — шины PCI Express, HyperTransport и QPI.

Шина PCI Express

Стандарт PCI Express, часто обозначаемый как PCI-E, был опубликован в 2002 году[1] [8, 60]. Разработку шины, которая сначала называлась Arapahoe, инициировала фирма Intel. Развитием стандарта занимается организация PCI Special Interest Group. В январе 2007 года она представила спецификацию PCI Express 2.0, а следующим этапом должна стать версия PCI Express 3.0 с пропускной способностью вдвое большей, чем у PCI Express 2.0.

В отличие от параллельных системных шин, где все устройства подключаются к параллельной двунаправленной шине, системная шина PCI Express — это совокупность самостоятельных последовательных двунаправленных каналов передачи

[1] В 2007 году опубликован стандарт PCI Express 2.0, а в 2010 году ожидается появление стандарта PCI Express 3.0 с пропускной способностью 8 Гбит/с.

данных, объединенных по топологии типа «звезда». В терминологии, принятой для PCI-E, канал называют PCI Express Link или просто *link*. Все каналы одной стороной подключены к коммутатору (switch), расположенному в центре звезды. К противоположному концу канала подключается одно из устройств, участвующих в обмене. Таким образом, между каждым устройством и коммутатором реализуется соединение типа «точка-точка». Благодаря такой схеме любое устройство посредством коммутатора может быть соединено с любым другим, что и требуется от системной шины. В упрощенном варианте организация системной шины PCI-E показана на рис. 7.17.

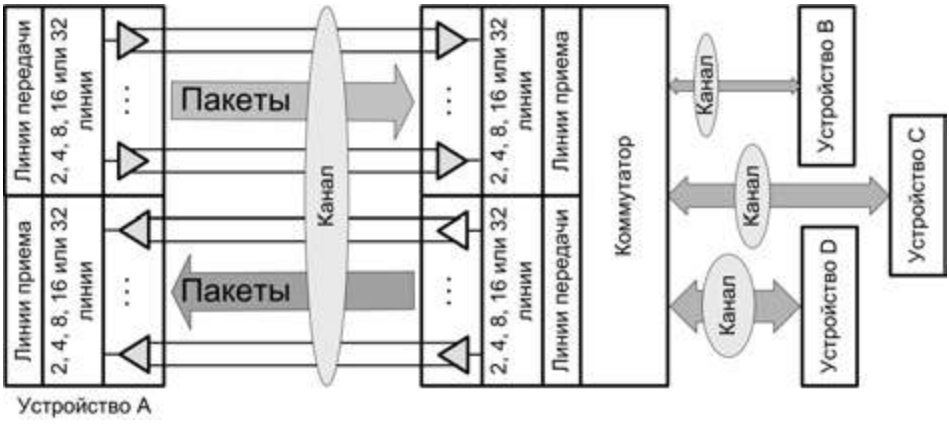


Рис. 7.17. Организация системной шины на базе стандарта PCI-E

Канал может состоять из одной, либо нескольких (2, 4, 8, 16 или 32) двунаправленных линий передачи сигналов, которые в терминологии PCI-E называются *lanes* (в дальнейшем вместо lane будем использовать термин «линия»). Каждая линия реализуется в виде двух пар проводов (по одной паре для передачи и для приема). Применение нескольких линий позволяет масштабировать пропускную способность канала, которая для одной линии составляет 2,5 Гбит/с. Канал из 32 линий обеспечивает пропускную способность 16 Гбайт/с (в PCI Express 2.0 — 32 Гбит/с).

Информация в PCI Express передается в пакетах, которые могут содержать данные, адреса, служебные запросы и т. д., причем все виды пакетов передаются по тем же линиям, что и пакеты с данными. Обмен информацией ведется одновременно (но не синхронно) по всем доступным линиям. Так, для канала с четырьмя линиями первый пакет на передачу может быть отправлен по первой линии, второй — по второй, третий — по третьей, четвертый — по четвертой, а вот пятый — снова по первой линии и т. д. Каждая линия работает независимо от других, за счет этого и достигаются столь большие скорости передачи данных. Используется протокол с расщеплением транзакций.

В каналах для помехозащищенности может использоваться избыточное кодирование 8B/10B, где каждый байт представляется десятью битами. Контрольная информация передается по тем же линиям, что и данные.

Шина PCI Express управляется контроллером, называемым *главным мостом* (Host Bridge). Вся управляющая информация, включая запросы прерывания, передается по тем же линиям, что и данные.

Шина HyperTransport

Не менее прогрессивной можно считать и другую технологию последовательной передачи данных — HyperTransport (HT), предложенную корпорацией AMD и продвигаемую консорциумом HyperTransport Technology. Шина HT состоит из двунаправленных линий передачи данных и является масштабируемой — возможно использование 2, 4, 8, 16 или 32 линий на каждое направление, причем разрешается, например, использовать 2 линии на прием и 32 на передачу. Упрощенная структура шины (без некоторых дополнительных сигналов) показана на рис. 7.18.

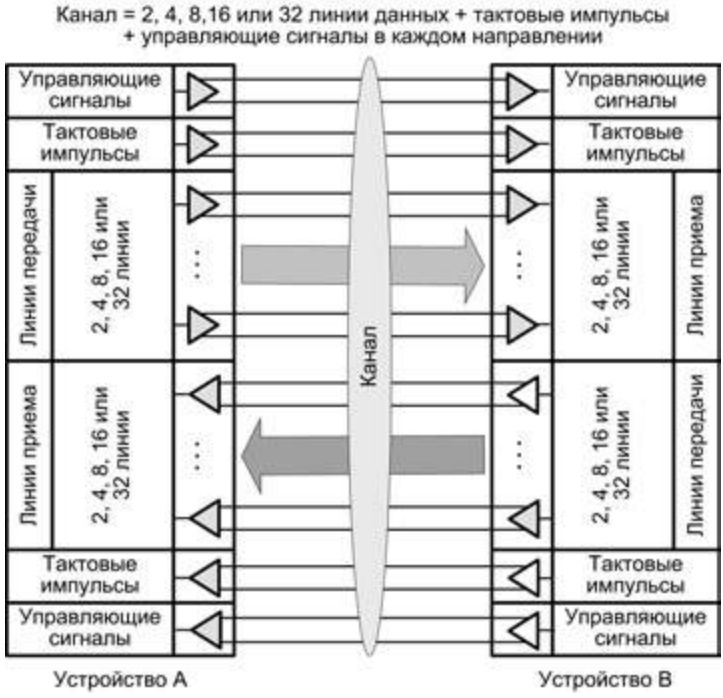


Рис. 7.18. Упрощенная структура шины Hypertransport

Данные, передаваемые по линии, упаковываются в пакеты стандартного вида. Каждый пакет состоит из 32-разрядных слов, первое из которых — управляющее. Если пакет содержит адрес, то последние 8 битов управляющего слова сцеплены со следующим 32-битным словом, тем самым образуя 40-битный адрес. Возможна и 64-разрядная адресация (это указывается в управляющем слове). Данные всегда передаются 32-битными словами, вне зависимости от реальной длины данных.

В линиях используется технология DDR (Double Data Rate), в которой данные посылаются как по переднему, так и по заднему фронтам сигнала синхронизации.

Частота сигнала синхронизации настраивается автоматически. 32-битная шина в двунаправленном режиме способна обеспечить пропускную способность до 41,6 Гбайт/с, являясь, таким образом, самой быстрой шиной среди себе подобных.

По скорости работы и удобству разводки HT перекрывает любые мыслимые потребности, однако возможности используемых протоколов не позволяют реализовывать сложную маршрутизацию, обеспечиваемую PCI Express.

Hyper Transport на данный момент почти идеален в качестве системной шины. PCI Express лучше подходит для оконечных внешних устройств. В общем, каждая из шин по-своему достаточно хороша, и однозначно отдать преимущество какой-то одной из них весьма трудно.

Шина QPI

Шину QPI (QuickPath Interconnect), впервые примененную в многоядерных микропроцессорах Nehalem фирмы Intel, можно рассматривать как приблизительный аналог шины Hyper Transport.

Шина также базируется на технологии «точка-точка» и представляет собой два 20-битных подканала, ориентированных на передачу данных в прямом и обратном направлении. 16 битов предназначаются для передачи данных, оставшиеся четыре носят вспомогательный характер и используются для нужд протокола и для коррекции ошибок. Помимо этого, каждый подканал содержит линию для передачи тактирующих импульсов (ТИ). Все линии двухпроводные, таким образом, канал состоит из 84 проводов. Структура шины QPI показана на рис. 7.19.

Рис. 7.19. Структура шины QPI

Пропускная способность шины составляет 12,8 Гбайт/с в каждую сторону или 25,6 Гбайт/с суммарно, что приблизительно равно пропускной способности шины HyperTransport v3.0.

Шины «малого» интерфейса

Среди наиболее распространенных шин «малого» интерфейса следует, прежде всего, отметить шины USB, FireWire, Bluetooth, IrDA.

Шина USB

Шина USB (Universal Serial Bus — универсальная последовательная шина) ориентирована на среднескоростные и низкоскоростные периферийные устройства (ПУ), подключаемые к ВМ. Для подключения ПУ к шине USB используется четырехпроводный кабель, при этом два провода используются для приема и передачи данных, а два оставшихся — для питания ПУ. Благодаря встроенным линиям питания USB позволяет подключать периферийные устройства без собственного источника питания. Работой всей системы USB управляет контроллер, являющийся программно-аппаратной подсистемой ВМ. К одному контроллеру шины USB можно подсоединить до 127 устройств по топологии «звезда». В первых своих версиях (USB 1) шина обеспечивала две скорости обмена — высокую (до 12 Мбит/с) и низкую (1,5 Мбит/с). ПУ могут иметь разные скорости обмена. В стандарте USB 2 максимальная скорость увеличена до 480 Мбит/с, а в разрабатывающемся стандарте USB 3 — до 4,8 Гбит/с.

Вариант шины USB, известный как Wireless USB, позволяет организовать беспроводную связь с высокой скоростью передачи информации (до 480 Мбит/с на расстоянии 3 метра и до 110 Мбит/с на расстоянии 10 метров).

Шина FireWire

FireWire — это еще одна стандартная технология шины последовательной передачи данных для соединения компьютера с периферией. ПУ подключаются к ВМ с помощью 6-проводной линии, где 4 провода используются для передачи информации, а два — для подачи питания на ПУ. Шина обеспечивает высокую скорость передачи информации (до 800 Мбит/с, в перспективе — 3,2 Гбит/с). Поддерживается режим пакетной передачи данных: данные разбиваются на пакеты с интервалами между ними, в которых посылается служебная информация, например «Стоп» или «Пуск». Шина поддерживает как асинхронную, так и изохронную передачу информации. Напомним, что при асинхронной передаче получение каждого пакета данных проверяется, и, если он не получен или принят с повреждением, передача повторяется и ошибки исправляются. В изохронном режиме данные передаются пакетами, следующими друг за другом, не ожидая подтверждения получения.

По производительности FireWire превосходит USB.

Интерфейс Bluetooth

Интерфейс Bluetooth. Дословно «Bluetooth» переводится как «Синий зуб». Название взято из истории Скандинавии. Король Харальд Блютус (Harald Bluetooth)

был предводителем викингов, вошедшим в историю как король-объединитель скандинавских земель. Этим разработчики интерфейса видимо хотели подчеркнуть «объединяющую» миссию технологии.

На сегодняшний день Bluetooth является одной из наиболее перспективных технологий беспроводной передачи данных. Работающий на частоте 2,4 ГГц приемопередатчик, каким является Bluetooth-чип, позволяет в зависимости от степени мощности устанавливать связь в пределах 10 или 100 метров. Этого достаточно для покрытия стандартного дома, при этом нет необходимости в том, чтобы соединяемые устройства находились в зоне прямой видимости друг друга. Скорость передачи данных в первых версиях интерфейса составляла 2–3 Мбит/с. В стандарте Bluetooth 3.0 она определена уже величиной 480 Мбит/с.

Интерфейс Bluetooth состоит из трех частей: приемо-передатчика, контроллера связи и управляющего устройства, осуществляющего связь с терминалом (рис. 7.20). Терминалом может быть любой прибор, будь то мобильный телефон, КПК или ноутбук. Приемопередатчик и контроллер связи, как правило, выполнены на отдельных микросхемах, а вот функции управляющего устройства может выполнять и процессор терминала при достаточной собственной мощности. Схема проста в реализации, что сказывается на популярности интерфейса Bluetooth.

Рис. 7.20. Схема организации Bluetooth-связи

Bluetooth работает по принципу FHSS (Frequency-Hopping Spread Spectrum): передатчик разбивает данные на пакеты и передает их на разных частотах, скачкообразно переходя с одной частоты на другую. Смена частот происходит 1600 раз в секунду по псевдослучайному алгоритму или по шаблону, составленному из 79 частот. «Понять» друг друга могут только те устройства, которые настроены на один и тот же шаблон передачи — для посторонних приборов переданная информация будет обычным шумом.

Конкурентом Bluetooth в области беспроводных интерфейсов можно считать упоминавшийся выше Wireless USB (WUSB), также обеспечивающий скорость 480 Мбит/с (для WUSB это только начальный уровень — предполагается дальнейшее увеличение пропускной способности до 1 Гбит/с и выше). В то же время существенным преимуществом Bluetooth можно считать значительно меньшее энергопотребление.

Интерфейс IrDA

Инфракрасный интерфейс IrDA (Infra red Data Assotiation) позволяет соединяться с периферийным оборудованием без кабеля при помощи ИК-излучения с длиной

волны 880 нм. Порт IrDA позволяет устанавливать связь на коротком расстоянии до одного метра в режиме точка-точка. В цели IrDA входили низкое потребление и экономичность, поэтому интерфейс IrDA использует узкий ИК-диапазон с малой мощностью потребления, что позволяет создать недорогую аппаратуру.

Устройство инфракрасного интерфейса подразделяется на два основных блока: преобразователь и блок кодирования/декодирования (рис. 7.21).

Рис. 7.21. Схема организации инфракрасной связи

Блоки обмениваются данными на уровне электрических сигналов. Перед передачей данные пакуются в кадры, состоящие из 10 битов, где байт данных обрамляется одним старт-битом в начале и одним стоп-битом в конце кадра. Порт IrDA использует универсальный асинхронный приемопередатчик UART (Universal Asynchronous Receiver Transmitter) и работает со скоростью передачи данных 2400–115 200 бит/с.

Перед началом передачи устройство выясняет, нет ли в ближайшей окрестности активности в ИК-диапазоне (не ведется ли какая-либо передача в пределах его досягаемости). Если такая активность обнаружена, то программе, выдающей запрос, посылается соответствующее сообщение, а передача откладывается. Если оба соединяющихся устройства являются компьютерами, то любое из них может быть ведущим. Выбор зависит от того, какое из устройств первым проявит инициативу. В процессе установления связи два устройства «договариваются» о максимальной скорости, с которой они оба могут работать, и ряде других параметров обмена. Все первичные передачи, выполняемые до фазы переговоров, по умолчанию ведутся на скорости 9,6 Кбит/с. Любая станция, не принимающая в данный момент времени участия в обмене, перед началом передачи должна прослушивать канал не менее 500 мс, чтобы убедиться в отсутствии трафика в ИК-диапазоне. С другой стороны, станция, участвующая в обмене, должна вести передачу не более 500 мс.

Контрольные вопросы

1. Перечислите основные виды структур взаимосвязей вычислительной машины.
2. Какие параметры включает в себя полная характеристика шины?
3. Что такое транзакция, из каких этапов она состоит?
4. В чем заключается основное различие между ведущими и ведомыми устройствами?
5. Что такое широковещательный режим записи?
6. Какие шины в составе ВМ образуют иерархию шин?

7. Что такое «широковещательный опрос» и как он реализуется?
8. Какая характеристика вычислительной машины зависит от ширины адресной шины?
9. В чем заключаются основные преимущества и недостатки мультиплексирования адреса и данных?
10. Какие группы сигнальных линий образуют шину управления?
11. Определите задачи арбитража шин.
12. Перечислите и охарактеризуйте алгоритмы динамической смены приоритетов.
13. Сформулируйте преимущества и недостатки централизованного и децентрализованного арбитража.
14. Чем определяются приоритеты устройств при цепочечном арбитраже?
15. Поясните схему арбитража Тауба.
16. Какими способами организуются опросные виды арбитража?
17. Каким образом выполняются транзакции в шинах с синхронным протоколом?
18. Поясните последовательность действий в процедуре квитирования установления связи.
19. Какие средства обеспечивают исключение бесконечного ожидания ответа в асинхронном протоколе?
20. Для каких шин наиболее характерен пакетный режим передачи информации?
21. В чем суть и достоинства конвейеризации транзакций?
22. Какие дополнительные преимущества предоставляет расщепление транзакций?
23. Какими средствами можно увеличить полосу пропускания шины?
24. Перечислите способы ускорения транзакций на шине.
25. Охарактеризуйте основные тенденции стандартизации шин малого и большого интерфейса.

Глава 8

Системы ввода/вывода

Помимо центрального процессора (ЦП) и памяти, третьим ключевым элементом архитектуры ВМ является *система ввода/вывода* (СВВ), обеспечивающая обмен информацией между ядром ВМ и разнообразными периферийными устройствами (ПУ). Технические и программные средства СВВ несут ответственность за физическое и логическое сопряжение *ядра вычислительной машины* и ПУ.

Технически система ввода/вывода в рамках ВМ реализуется совокупностью *модулей ввода/вывода* (МВВ). Модуль ввода/вывода выполняет сопряжение ПУ с ядром вычислительной машины и различные коммуникационные операции между ними. Две основные функции МВВ:

- обеспечение интерфейса с ЦП и памятью (*«большой» интерфейс*);
- обеспечение интерфейса с одним или несколькими периферийными устройствами (*«малый» интерфейс*).

Анализируя архитектуру известных ВМ, можно выделить три основных способа подключения СВВ к ядру вычислительной машины (рис. 8.1).

Рис. 8.1. Место системы ввода/вывода в архитектуре вычислительной машины: *а* — с раздельными шинами памяти и ввода/вывода; *б* — с совместно используемыми линиями данных и адреса; *в* — подключение на общих правах с процессором и памятью

В варианте с раздельными шинами памяти и ввода/вывода (см. рис. 8.1, *а*) обмен информацией между ЦП и памятью физически отделен от ввода/вывода, поскольку

обеспечивается полностью независимыми шинами. Это дает возможность обращаться к памяти одновременно с выполнением ввода/вывода. Кроме того, данный архитектурный вариант ВМ позволяет специализировать каждую из шин, учесть формат пересылаемых данных, особенности синхронизации обмена и т. п. В частности, шина ввода/вывода, с учетом характеристик реальных ПУ, может иметь меньшую пропускную способность, что позволяет снизить затраты на ее реализацию. Недостатком решения можно считать большое количество точек подключения к ЦП.

Второй вариант — с совместно используемыми линиями данных и адреса (см. рис. 8.1, *б*). Память и СВВ имеют общие для них линии адреса и линии данных, разделяя их во времени. В то же время управление памятью и СВВ, а также синхронизация их взаимодействия с процессором осуществляются независимо по раздельным линиям управления. Это позволяет учесть особенности процедур обращения к памяти и к модулям ввода/вывода и добиться наибольшей эффективности доступа к ячейкам памяти и периферийным устройствам.

Третий тип архитектуры ВМ предполагает подключение СВВ к системной шине на общих правах с процессором и памятью (см. рис. 8.1, *в*). Преимущества и недостатки такого подхода обсуждались при рассмотрении вопросов организации шин (глава 7).

Потенциально возможен также вариант подключения периферийных устройств к системной шине напрямую, без использования МВВ, но против него можно выдвинуть сразу несколько аргументов. Во-первых, в этом случае ЦП пришлось бы оснащать универсальными схемами для управления любым ПУ. При большом разнообразии периферийных устройств, имеющих к тому же различные принципы действия, такие схемы оказываются чересчур сложными и избыточными. Во-вторых, пересылка данных при вводе и выводе происходит значительно медленнее, чем при обмене между ЦП и памятью, и было бы невыгодно задействовать для обмена информацией с ПУ высокоскоростную системную шину. И наконец, в ПУ часто используются иные форматы данных и длина слова, чем в ВМ, к которым они подключены.

Адресное пространство системы ввода/вывода

Как и в случае памяти, операции ввода/вывода также предполагают наличие некоторой системы адресации, позволяющей выбрать один из модулей СВВ, а также одно из подключенных к нему периферийных устройств. Адрес модуля и ПУ является составной частью соответствующей команды.

Адресное пространство ввода/вывода может быть совмещено с адресным пространством памяти или может быть выделенным.

При *совмещении адресного пространства* для адресации модулей ввода/вывода отводится определенная область адресов (рис. 8.2). Обычно все операции с модулем ввода/вывода осуществляются с использованием входящих в него внутренних регистров: управления, состояния, данных. Фактически процедура ввода/вывода сводится к записи информации в одни регистры МВВ и считыванию ее из других регистров. Это позволяет рассматривать регистры МВВ как ячейки основной памяти и работать с ними с помощью обычных команд обращения к памяти, при этом

в системе команд ВМ вообще могут отсутствовать специальные команды ввода и вывода. Так, модификацию регистров МВВ можно производить непосредственно с помощью арифметических и логических команд. Адреса регистрам МВВ назначаются в области адресного пространства памяти, отведенной под систему ввода/вывода.

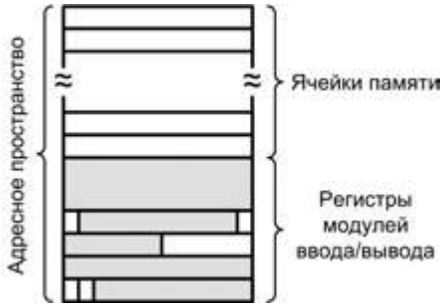


Рис. 8.2. Распределение совмещенного адресного пространства

Такой подход представляется вполне оправданным, если учесть, что ввод/вывод обычно составляет малую часть всех операций, выполняемых вычислительной машиной, чаще всего не более 1% от общего числа команд в программе.

К достоинствам совмещенного адресного пространства можно отнести:

- унификацию операций обращения к памяти и периферийным устройствам;
- возможность увеличения числа подключаемых периферийных устройств;
- возможность внепроцессорного обмена данными между периферийными устройствами, если в системе команд есть команды пересылки между ячейками памяти;
- возможность обмена информацией не только с аккумулятором, но и с любым регистром центрального процессора.

Недостатки совмещенного адресного пространства:

- сокращение области адресного пространства памяти;
- усложнение декодирующих схем адресов в СВВ;
- трудности распознавания операций передачи информации при вводе/выводе среди других операций.

Совмещенное адресное пространство используется в вычислительных машинах MIPS и SPARC.

В случае *выделенного адресного пространства* для обращения к модулям ввода/вывода применяются специальные команды и отдельная система адресов. Это позволяет разделить тракты для работы с памятью и тракты ввода/вывода, что дает возможность совмещать во времени обмен с памятью и ввод/вывод. Кроме того, адресное пространство памяти может быть использовано по прямому назначению в полном объеме. В вычислительных машинах фирмы IBM и ВМ на базе процессоров фирмы Intel система ввода/вывода, как правило, организуется в соответствии с концепцией выделенного адресного пространства.

Достоинства выделенного адресного пространства:

- адрес периферийного устройства в команде ввода/вывода может быть коротким, благодаря чему сокращается длина таких команд и упрощаются схемы дешифрации адреса;
- программы становятся более наглядными, так как операции ввода/вывода выполняются с помощью специальных команд;
- разработка СВВ может проводиться отдельно от разработки памяти.

Недостатки выделенного адресного пространства:

- ввод/вывод производится только через аккумулятор центрального процессора;
- малая разрядность адреса периферийного устройства ограничивает число подключаемых ПУ.

Периферийные устройства

Связь ВМ с внешним миром осуществляется с помощью самых разнообразных периферийных устройств. Каждое ПУ подключается к модулю ввода/вывода посредством индивидуальной шины. Интерфейс, по которому организуется взаимодействие МВВ и ПУ, часто называют *малым*. Индивидуальная шина обеспечивает обмен данными и управляющими сигналами, а также информацией о состоянии участников обмена. Все множество ПУ можно свести к трем категориям [143]:

- для общения с пользователем;
- для общения с ВМ;
- для связи с удаленными устройствами.

Примерами первой группы служат видеотерминалы и принтеры. Ко второй группе причисляются внешние запоминающие устройства (магнитные и оптические диски, магнитные ленты и т. п.), датчики и исполнительные механизмы. Отметим двойственную роль внешних ЗУ, которые, с одной стороны, представляют собой часть памяти ВМ, а с другой — являются периферийными устройствами. Наконец, устройства третьей категории позволяют ВМ обмениваться информацией с удаленными объектами, которые могут относиться к двум первым группам. В роли удаленных объектов могут выступать также другие ВМ.

Обобщенная структура ПУ показана на рис. 8.3. Интерфейс с МВВ реализуется в виде сигналов управления, состояния и данных. *Данные* представлены совокупностью битов, которые должны быть переданы в модуль ввода/вывода или получены из него. *Сигналы управления* определяют функцию, которая должна быть выполнена периферийным устройством. Это может быть стандартная для всех устройств функция — посылка данных в МВВ или получение данных из него, либо специфичная для данного типа ПУ функция, такая, например, как позиционирование головки магнитного диска или перемотка магнитной ленты. *Сигналы состояния* характеризуют текущее состояние устройства, в частности включено ли ПУ и готово ли оно к передаче данных.

Логика управления — это схемы, координирующие работу ПУ в соответствии с направлением передачи данных. Задачей *преобразователя* является трансформация информационных сигналов, имеющих самую различную физическую природу,

в электрические сигналы, а также обратное преобразование. Обычно совместно с преобразователем используется *буферная память*, обеспечивающая временное хранение данных, пересылаемых между МВВ и ПУ.

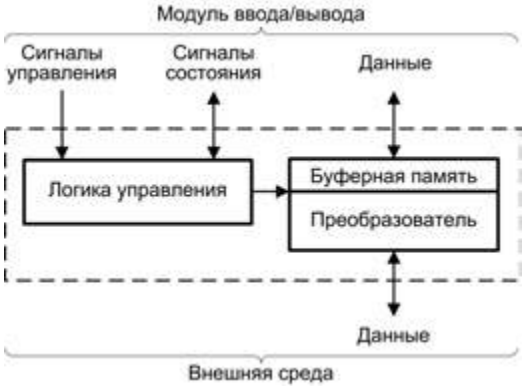


Рис. 8.3. Структура периферийного устройства

Модули ввода/вывода

Функции модуля

Модуль ввода/вывода в составе вычислительной машины отвечает за управление подключенными к нему одним или несколькими ПУ и за обмен данными между этими устройствами, с одной стороны, и основной памятью или регистрами ЦП — с другой. Основные функции МВВ можно сформулировать следующим образом:

- локализация данных;
- управление и синхронизация;
- обмен информацией;
- буферизация данных;
- обнаружение ошибок.

Локализация данных

Под *локализацией данных* будем понимать возможность обращения к одному из ПУ, а также адресации данных на нем.

Адрес ПУ обычно содержится в адресной части команд ввода/вывода. Как уже отмечалось, в состав СВВ могут входить несколько модулей ввода/вывода. К каждому МВВ может быть подключено несколько ПУ, поэтому каждому модулю назначается определенный диапазон адресов, независимо от того, является ли пространство адресов совмещенным или раздельным. Старшие разряды обеспечивают выбор одного из МВВ в рамках системы ввода/вывода. Младшие разряды адреса представляют собой уникальные адреса регистров данного модуля или подключенных к нему ПУ.

Одной из функций МВВ является проверка «попадания» поступившего адреса в выделенный данному модулю диапазон адресов. При положительном ответе

модуль должен обеспечить декодирование поступившего адреса и перенаправление информации к адресуемому объекту или от него.

Для простейших периферийных устройств (клавиатура, принтер и т. п.) адрес ПУ однозначно определяет и расположение данных на этом устройстве. Для более сложных ПУ, таких как внешние запоминающие устройства, информация о местонахождении данных требует детализации. Так, для ЗУ на магнитном диске необходимо указать номер цилиндра, номер сектора и т. п. Эта часть адресной информации передается в МВВ не по шине адреса, а в виде служебных сообщений, пересылаемых по шине данных. Обработка такой информации в модуле, естественно, сложнее, чем выбор нужного регистра или ПУ. В частности, она может требовать от МВВ организации процедуры поиска на носителе информации.

Управление и синхронизация

Функция *управления и синхронизации* заключается в том, что МВВ должен координировать перемещение данных между внутренними ресурсами ВМ и периферийными устройствами. При разработке системы управления и синхронизации модуля ввода/вывода необходимо учитывать целый ряд факторов.

Прежде всего, нужно принимать во внимание, что центральный процессор может взаимодействовать одновременно с несколькими ПУ, причем быстродействие подключаемых к МВВ периферийных устройств варьируется в очень широких пределах — от нескольких байтов в секунду в терминалах до десятков миллионов байтов в секунду при обмене с магнитными дисками. Если в системе используются шины, каждое взаимодействие между ЦП и МВВ включает в себя одну или несколько процедур арбитража.

В отличие от обмена с памятью, процессы ввода/вывода и работа ЦП протекают не синхронно. Очередная порция информации может быть выдана на устройство вывода лишь тогда, когда это устройство готово их принять. Аналогично, ввод допустим только в случае доступности информации на устройстве ввода. Несинхронный характер процессов ввода/вывода предполагает обмен сигналами, который для двухпроводной системы синхронизации (применительно к операции вывода) можно описать следующим образом:

1. Центральный процессор с помощью сигнала ДД = 1 (данные достоверны) извещает о доступности данных, подлежащих выводу.
2. Приняв данные, устройство вывода сообщает процессору об их получении сигналом ДП = 1 (данные приняты).
3. Получив подтверждение, ЦП обнуляет сигнал ДД и снимает данные с шины, после чего может выставить на шину новые данные.
4. Обнаружив, что ДД = 0, устройство вывода, в свою очередь, устанавливает в нулевое состояние сигнал ДП, после чего оно готово для обработки принятых данных (и обрабатывает их до получения очередного сигнала ДД = 1).

Таким образом, модуль ввода/вывода обязан снабдить центральный процессор информацией о собственной готовности к обмену, а также о готовности подключенных к модулю ПУ. Помимо этого, процессор должен обладать оперативной информацией и об иных событиях, происходящих в СВВ.

Обмен информацией

Основной функцией МВВ является обеспечение *обмена информацией*. Со стороны «большого» интерфейса — это обмен с ЦП, а со стороны «малого» интерфейса — обмен с ПУ. В таком плане требования к МВВ диктуются последовательностью операций, выполняемых процессором при вводе/выводе:

1. Выбор требуемого периферийного устройства.
2. Определение состояния МВВ и ПУ.
3. Выдача указания модулю ввода/вывода на подключение нужного ПУ к процессору.
4. Получение от МВВ подтверждения о подключении затребованного ПУ к процессору.
5. Распознавание сигнала готовности устройства к передаче очередной порции информации.
6. Прием (передача) порции информации.
7. Циклическое повторение двух предшествующих пунктов до завершения передачи информации в полном объеме.
8. Логическое отсоединение ПУ от процессора.

С учетом описанной процедуры функция *обмена информацией* с ЦП включает в себя:

- *распознавание команды*: МВВ получает команды из ЦП в виде сигналов на шине управления;
- *пересылку данных* между МВВ и ЦП;
- *извещение о состоянии*: из-за того что ПУ — медленные устройства, важно знать состояние модуля ввода/вывода. Так, в момент получения запроса на пересылку данных в центральный процессор МВВ может быть не готов выполнить эту пересылку, поскольку еще не завершил предыдущую команду. Этот факт должен быть сообщен процессору с помощью соответствующего сигнала. Возможны также сигналы, уведомляющие о возникших ошибках;
- *распознавание адреса*: МВВ обязан распознавать адрес каждого ПУ, которым он управляет.

Помимо обмена с процессором, МВВ должен обеспечивать функцию обмена информацией с ПУ. Такой обмен также включает в себя передачу данных, команд и информации о состоянии.

Буферизация

Несмотря на различия в скорости обмена информацией для разных ПУ, все они в этом плане значительно отстают от ЦП и памяти. Такое различие компенсируется за счет буферизации. При выводе информации на ПУ данные пересылаются из основной памяти в МВВ с большой скоростью. В модуле эти данные буферизируются и затем направляются в ПУ со скоростью, свойственной последнему. При вводе из ПУ данные буферизируются так, чтобы не заставлять память работать в режиме медленной передачи. Таким образом, МВВ должен обеспечивать работу как со скоростью памяти, так и со скоростью ПУ.

Обнаружение ошибок

Еще одной из важнейших функций МВВ является обнаружение ошибок, возникающих в процессе ввода/вывода. Центральный процессор следует оповещать о каждом случае обнаружения ошибки. Ошибки могут быть вызваны самыми разнообразными факторами:

- загрязнение и влага;
- повышенная или пониженная температура окружающей среды;
- электромагнитное облучение;
- скачки напряжения питания.
- старение элементной базы;
- ошибки в системном программном обеспечении;
- ошибки в пользовательском программном обеспечении.

Степень влияния каждого из этих факторов зависит от типа и конструкции МВВ и ПУ. Так, к загрязнению наиболее чувствительны оптические и механические элементы ПУ, в то время как работа электронных компонентов СВВ в большей степени зависит от температуры внешней среды, электромагнитного воздействия и стабильности питающего напряжения.

Фактор старения характерен как для механических, так и для электронных элементов СВВ. В механических элементах он выражается в виде износа, следствием чего может быть неточное позиционирование головок считывания/записи на внешних запоминающих устройствах или неправильная подача бумаги в принтерах. Старение электронных элементов обычно выражается в изменении электрических параметров схем, приводящем к нарушению управления и синхронизации. Так, отклонения в параметрах электронных компонентов могут вызвать недопустимый «перекос» сигналов, передаваемых между ЦП и МВВ или внутри МВВ.

Источником ошибок может стать и несовершенство системного программного обеспечения (ПО):

- непредвиденные последовательности команд или кодовые комбинации;
- некорректное распределение памяти;
- недостаточный размер буфера ввода/вывода;
- недостаточно продуманные и проверенные комбинации системных модулей.

Среди ошибок, порождаемых пользовательским ПО, наиболее частыми являются:

- нарушение последовательности выполнения программы;
- некорректные процедуры.

Вероятность возникновения ошибки внутри процессора для современных ЦП оценивается величиной порядка 10^{-18}, в то время как для остальных составляющих ВМ она лежит в диапазоне 10^{-8}–10^{-12}.

Способы обнаружения и исправления ошибок ввода/вывода практически не отличаются от рассмотренных применительно к памяти.

Структура модуля

Структура МВВ в значительной мере зависит от числа и сложности периферийных устройств, которыми он управляет, однако в самом общем виде такой модуль можно представить в форме, показанной на рис. 8.4.

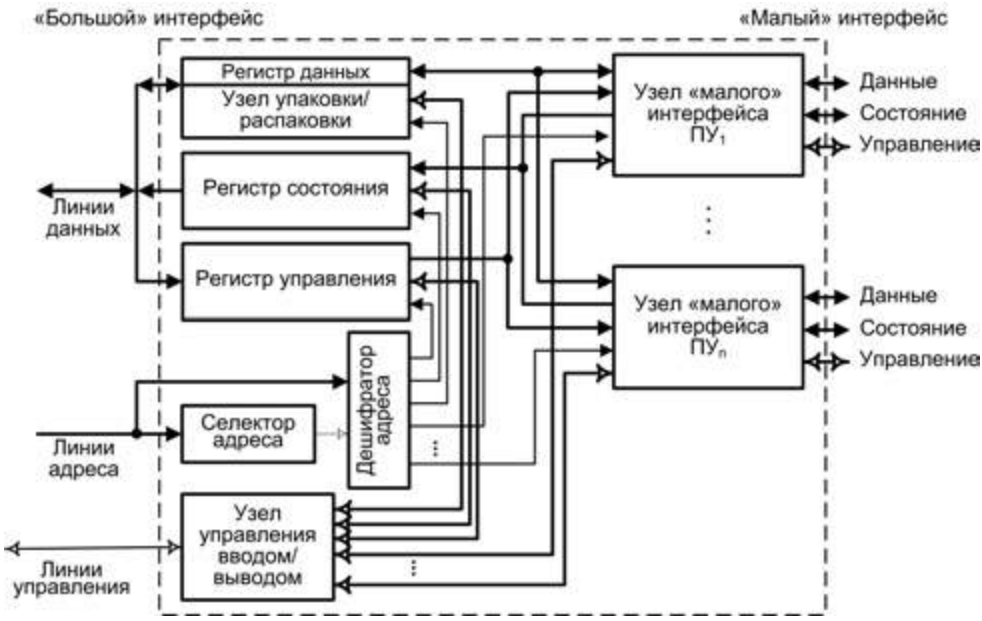


Рис. 8.4. Структура модуля ввода/вывода

Связь модуля ввода/вывода с ядром ВМ осуществляется посредством системной или специализированной шины. С этой стороны МВВ реализуется так называемый «большой» интерфейс. Большие различия в архитектуре систем команд и шин ВМ являются причиной того, что со стороны «большого» интерфейса модули ввода/вывода достаточно трудно унифицировать, и часто МВВ, созданные для одних ВМ, не могут быть использованы в других. Тем не менее в структурном плане они достаточно схожи.

Данные, передаваемые между МВВ и ядром вычислительной машины, буферизируются в регистре данных, благодаря чему удается компенсировать различие в быстродействии ядра ВМ и периферийных устройств. Разрядность регистра, как правило, совпадает с количеством линий в шине, подключаемой со стороны «большого» интерфейса (обычно 4 или 8 байтов). В то же время большинство ПУ ориентировано на побайтовый обмен информацией. Побайтовая пересылка информации по «широкой» системной шине — крайне неэффективное решение, поэтому со стороны «малого» интерфейса регистр данных часто дополняют узлом упаковки/распаковки (на схеме не показан). Этот узел при вводе обеспечивает последовательное побайтовое заполнение регистра данных (упаковку), а при выводе — последовательную побайтовую выдачу содержимого регистра на ПУ (распаковку).

В результате обмен данными через «большой» интерфейс выполняется за один такт. МВВ может содержать несколько регистров данных, что позволяет независимо хранить текущие данные каждого из периферийных устройств, подключенных к данному модулю ввода/вывода.

Помимо регистра данных, в составе МВВ имеются также регистр управления и регистр состояния (либо совмещенный регистр управления/состояния).

В *регистре управления* (РУ) фиксируются поступившие из ЦП команды управления модулем или подключенными к нему периферийными устройствами. Отдельные разряды регистра могут представлять такие команды, как очистка регистров МВВ, сброс ПУ, начало чтения, начало записи и т. п. В сложных МВВ присутствует несколько регистров управления, например регистр управляющих сигналов для модуля в целом и отдельные РУ для каждого из ПУ.

Регистр состояния (РС) служит для хранения битов состояния МВВ и подключенных к нему ПУ. Содержимое определенного разряда регистра может характеризовать, например, готовность устройства ввода к приему очередной порции данных, занятость устройства вывода или нахождение ПУ в автономном режиме (offline). В МВВ не исключается наличие и более одного регистра состояния.

Процедура ввода/вывода предполагает возможность работы с каждым регистром МВВ или периферийным устройством по отдельности. Такая возможность обеспечивается системой адресации. Каждому модулю в адресном пространстве ввода/вывода (совмещенном или раздельном) выделяется уникальный набор адресов, количество адресов в котором зависит от числа адресуемых элементов. Поступивший из ЦП адрес с помощью селектора адреса проверяется на принадлежность к диапазону, выделенному данному МВВ. В случае подтверждения дешифратор адреса выполняет раскодирование адреса, разрешая работу с соответствующим регистром модуля или ПУ.

Узел управления вводом/выводом, по сути, играет роль местного устройства управления МВВ. На него возлагаются две задачи: обеспечение взаимодействия с ЦП и координация работы всех составляющих МВВ. Связь с ЦП реализуется посредством линий управления, по которым из ЦП в модуль поступают сигналы, служащие для синхронизации операций ввода и вывода. В обратном направлении передаются сигналы, информирующие о происходящих в модуле событиях, например сигналы прерывания. Часть линий управления может задействоваться модулем для арбитража. Вторая функция узла управления реализуется с помощью внутренних сигналов управления.

Со стороны «малого» интерфейса МВВ обеспечивает подключение периферийных устройств и взаимодействие с ними. Эта часть МВВ более унифицирована, поскольку периферийные устройства всегда «подгоняются» под один из стандартных протоколов. Каждое из периферийных устройств обслуживается своим узлом «малого» интерфейса, который реализует принятый для данного ПУ стандартный протокол взаимодействия.

При управлении широким спектром ПУ модуль должен по возможности освобождать ЦП от знания деталей конкретных ПУ, так чтобы ЦП мог управлять любым

устройством с помощью простых команд чтения и записи. МВВ при этом берет на себя задачи синхронизации, согласования форматов данных и т. п.

Модуль ввода/вывода, который берет на себя детальное управление ПУ и общается с ЦП только с помощью команд высокого уровня, часто называют *каналом ввода/вывода* или *процессором ввода/вывода*. Наиболее примитивный МВВ, требующий детального управления со стороны ЦП, называют *контроллером ввода/вывода* или *контроллером устройства*. Как правило, контроллеры ввода/вывода типичны для микроЭВМ, а каналы ввода/вывода — для больших универсальных ВМ.

Методы управления вводом/выводом

В ВМ находят применение три способа организации ввода/вывода (В/ВЫВ):

- ввод/вывод с опросом;
- ввод/вывод по прерываниям;
- прямой доступ к памяти.

В первых двух способах все действия, связанные с процедурой ввода/вывода, выполняются под полным контролем центрального процессора, причем информация, пересылаемая между ПУ и памятью, сначала попадает в процессор, после чего он передает эту информацию в пункт назначения.

Ввод/вывод с опросом

Наиболее простым методом управления вводом/выводом является ввод/вывод с опросом. Здесь ввод/вывод происходит под полным контролем центрального процессора и реализуется специальной процедурой ввода/вывода. ЦП с помощью команды ввода/вывода сообщает модулю ввода/вывода, а через него и периферийному устройству о предстоящей операции. Адрес модуля и ПУ, к которому производится обращение, указывается в адресной части команды ввода или вывода. Модуль исполняет затребованное действие, после чего устанавливает в единицу соответствующий бит в своем регистре состояния. Ничего другого, чтобы уведомить ЦП, модуль не предпринимает. Следовательно, для определения момента завершения операции или пересылки очередного элемента блока данных процессор должен периодически опрашивать и анализировать содержимое регистра состояния МВВ.

Иллюстрация процедуры ввода с опросом блока данных с устройства ввода приведена на рис. 8.5. Данные читаются пословно. Для каждого читаемого слова ЦП должен оставаться в цикле проверки, пока не определит, что слово находится в регистре данных МВВ, то есть доступно для считывания.

Процедура начинается с выдачи процессором команды ввода, в которой указан адрес конкретного МВВ и конкретного ПУ. Существуют четыре типа команд ввода/вывода, которые может получить МВВ: управление, проверка, чтение и запись.

Команды управления используются для активизации ПУ и указания требуемой операции. Например, в устройство памяти на магнитной ленте может быть выдана команда перемотки или продвижения на одну запись. Для каждого типа ПУ характерны специфичные для него команды управления.

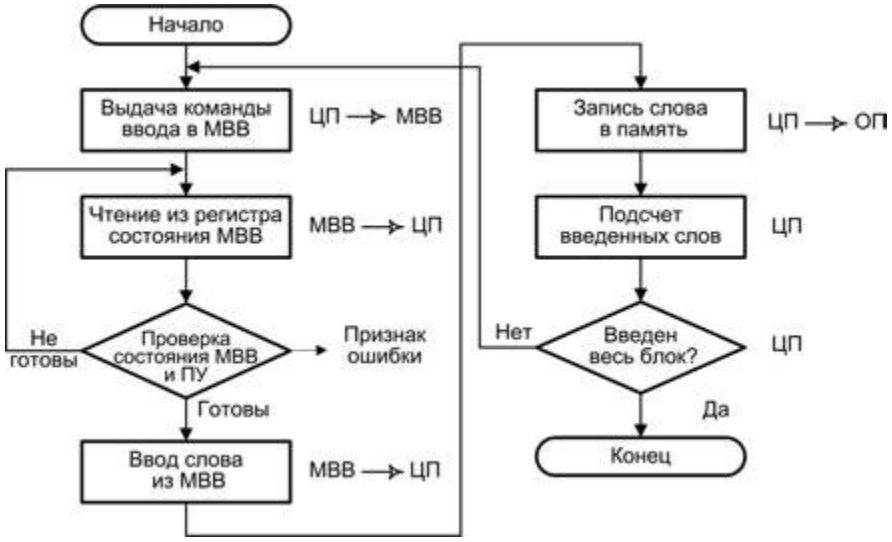


Рис. 8.5. Ввод данных с опросом

Команда проверки применяется для проверки различных ситуаций, возникающих в МВВ и ПУ в процессе ввода/вывода. С помощью таких команд ЦП способен выяснить, включено ли ПУ, готово ли оно к работе, завершена ли последняя операция ввода/вывода и не возникли ли в ходе ее выполнения какие-либо ошибки. Действие команды сводится к установке или сбросу соответствующих разрядов регистра состояния МВВ.

Команда чтения побуждает модуль получить элемент данных из ПУ и занести его в регистр данных (РД). ЦП может получить этот элемент данных, запросив МВВ поместить его на шину данных.

Команда записи заставляет модуль принять элемент данных (байт или слово) с шины данных и переслать его в РД с последующей передачей в ПУ.

Если к МВВ подключено несколько ПУ, то в процедуре ввода/вывода нужно производить циклический опрос всех устройств, с которыми в данный момент выполняются операции В/ВЫВ.

Основной недостаток рассматриваемого способа ввода/вывода — неэффективное использование процессора. Для определения момента готовности МВВ к пересылке очередной порции информации ЦП должен циклически опрашивать модуль на наличие сигнала готовности и никаких иных полезных действий в это время не выполняет. Кроме того, пересылка даже одного слова требует выполнения нескольких команд. ЦП должен тратить время на анализ битов состояния МВВ, запись в МВВ битов управления, чтение или запись данных со скоростью, определяемой периферийным устройством. Все это также отрицательно сказывается на эффективности ввода/вывода.

Главным аргументом в пользу ввода/вывода с опросом является простота МВВ, поскольку основные функции по управлению вводом/выводом берет на себя про-

цессор. При одновременной работе с несколькими ПУ приоритет устройств легко изменить программными средствами (последовательностью опроса). Наконец, подключение к МВВ новых периферийных устройств или отключение ранее подключенных также реализуется без особых сложностей.

Ввод/вывод по прерываниям

Альтернативой предыдущему способу может быть вариант, когда ЦП выдает команду ввода/вывода, а затем продолжает делать другую полезную работу, не занимаясь анализом готовности ПУ. Когда ПУ готово к обмену данными, оно через МВВ извещает об этом процессор с помощью запроса на прерывание. ЦП осуществляет передачу очередного элемента данных, после чего возобновляет выполнение прерванной программы. Подобную возможность предоставляет ранее рассмотренная система прерывания программ вычислительной машины.

Рассмотрим процедуру ввода блока данных с использованием ввода/вывода по прерываниям (рис. 8.6). Оставим без внимания такие подробности, как сохранение и восстановления контекста, действия, выполняемые при завершении пересылки блока данных, а также в случае возникновения ошибок.

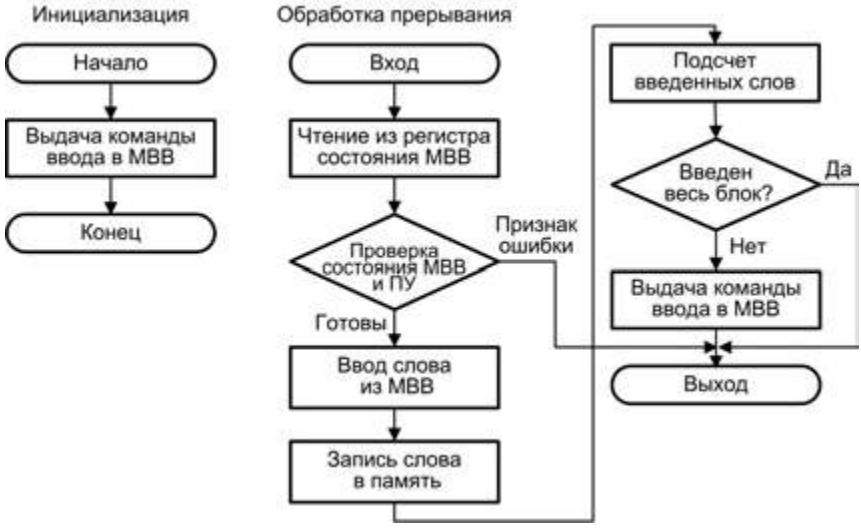


Рис. 8.6. Ввод данных по прерыванию

Процедура реализуется следующим образом. ЦП выдает команду чтения, а затем продолжает выполнение других заданий, например другой программы. Получив команду, МВВ приступает к вводу элемента данных с ПУ. Когда считанное слово оказывается в регистре данных модуля, МВВ формирует на управляющей линии сигнал прерывания ЦП. Выставив запрос, МВВ помещает введенную информацию на шину данных, после чего он готов к следующей операции ввода/вывода. ЦП в конце каждого цикла команды проверяет наличие запросов прерывания. Когда от МВВ приходит такой сигнал, ЦП сохраняет контекст текущей программы и обрабатывает

прерывание. В рассматриваемом случае ЦП читает слово из модуля, записывает его в память и выдает модулю команду на считывание очередного слова. Далее ЦП восстанавливает контекст прерванной программы и возобновляет ее выполнение.

Этот метод эффективнее ввода/вывода с опросом, поскольку устраняет ненужные ожидания, однако обработка прерывания занимает достаточно много времени ЦП. Кроме того, каждое слово, пересылаемое из памяти в модуль В/ВЫВ или в противоположном направлении, как и при методе с опросом, проходит через ЦП.

Прямой доступ к памяти

Хотя ввод/вывод по прерываниям эффективнее ввода/вывода с опросом, оба этих метода страдают двумя недостатками:

- темп передачи при вводе/выводе ограничен скоростью, с которой ЦП в состоянии опросить и обслужить устройство;
- ЦП вовлечен в управление передачей, для каждой пересылки он должен выполнить определенное количество команд.

Когда пересылаются большие объемы данных, требуется более эффективный способ ввода/вывода — *прямой доступ к памяти* (ПДП). ПДП предполагает наличие на системной шине дополнительного модуля — *контроллера прямого доступа к памяти* (КПДП), способного брать на себя функции ЦП по управлению системной шиной и обеспечивать прямую пересылку информации между ОП и ПУ без участия центрального процессора. В сущности, КПДП — это и есть модуль ввода/вывода, реализующий режим прямого доступа к памяти.

Если ЦП желает прочитать или записать блок данных, он прежде всего должен поместить в КПДП (рис. 8.7) информацию, характеризующую предстоящее действие. Этот процесс называется инициализацией КПДП и включает в себя занесение в контроллер следующих четырех параметров:

- вид запроса (чтение или запись);
- адрес устройства ввода/вывода;
- адрес начальной ячейки блока памяти, откуда будет извлекаться или куда будет вводиться информация;
- количество слов, подлежащих чтению или записи.

Первый параметр определяет направление пересылки данных: из ОП в ПУ или наоборот. За исходную точку обычно принимается память, поэтому под чтением понимают считывание данных из ОП и выдачу их на устройство вывода, а под записью — прием данных из устройства ввода и запись в ОП. Вид запроса запоминается в схеме логики управления контроллера.

К КПДП обычно могут быть подключены несколько ПУ, а адрес УВВ конкретизирует, какое из них должно участвовать в предстоящем обмене данными. Этот адрес запоминается в логике управления КПДП.

Третий параметр — адрес начальной ячейки — хранится в регистре адреса (РА) контроллера. После передачи каждого слова содержимое РА автоматически увеличивается на единицу, то есть в нем формируется адрес следующей ячейки ОП.

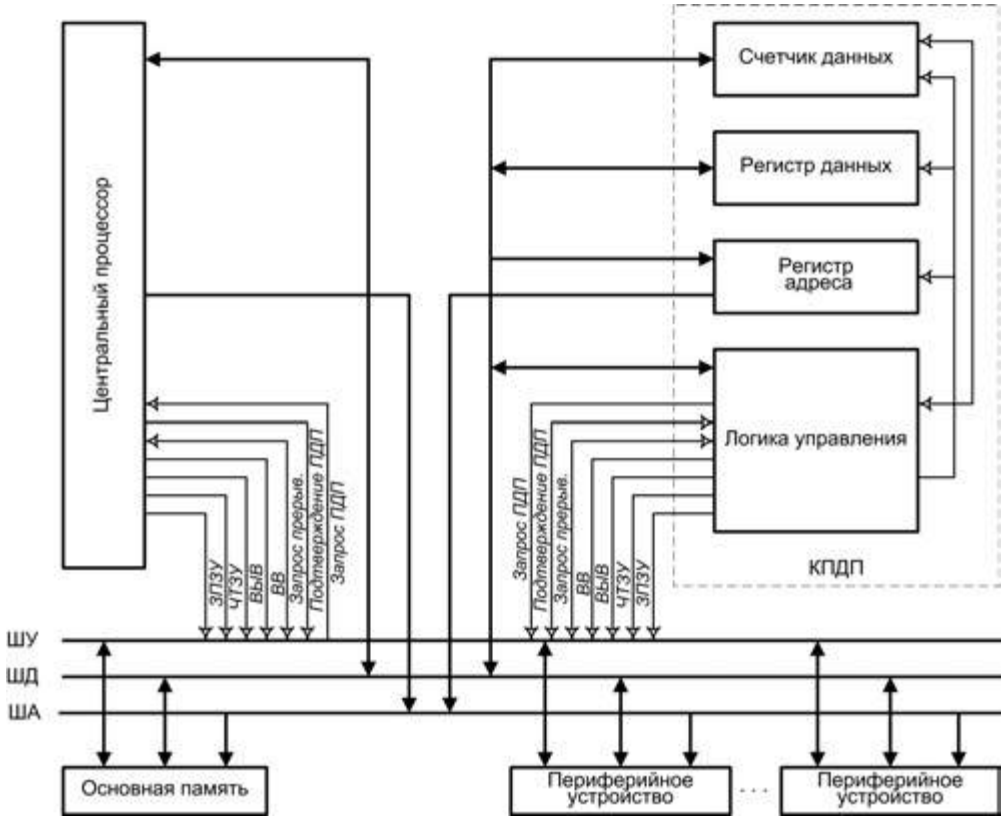


Рис. 8.7. Организация прямого доступа к памяти

Размер блока в словах заносится в счетчик данных (СД) контроллера. После передачи каждого слова содержимое СД автоматически уменьшается на единицу. Нулевое состояние СД свидетельствует о том, что пересылка блока данных завершена.

После инициализации процесс пересылки информации может быть начат в любой момент. Инициаторами обмена вправе выступать как ЦП, так и ПУ. Устройство, желающее начать В/ВЫВ, извещает об этом контроллер подачей соответствующего сигнала. Получив такой сигнал, КПДП выдает в ЦП сигнал Запрос ПДП. В ответ ЦП освобождает шины адреса и данных, а также те линии шины управления, по которым передаются сигналы, управляющие операциями на шине адреса (ША) и шине данных (ШД). К таким, прежде всего, относятся линии ЧТЗУ, ЗПЗУ, ВЫВ, ВВ и линия выдачи адреса на ША. Далее ЦП отвечает контроллеру сигналом Подтверждение ПДП, который для последнего означает, что ему делегированы права на управление системной шиной и можно приступать к пересылке данных.

Процесс пересылки каждого слова блока состоит из двух этапов.

При выполнении операции чтения (ОП → ПУ) на первом этапе КПДП выставляет на шину адреса содержимое РА (адрес текущей ячейки ОП) и формирует сигнал

ЧТЗУ. Считанное из ячейки ОП слово помещается на шину данных. На втором этапе КПДП выставляет на ША адрес устройства вывода и формирует сигнал ВЫВ, который обеспечивает передачу слова с шины данных в ПУ.

При выполнении операции записи (ПУ → ОП) КПДП сначала выдает на шину адреса номер (адрес) устройства ввода и формирует сигнал ВВ, по которому введенные данные поступают на шину данных. На втором этапе КПДП помещает на ША адрес ячейки ОП, куда должны быть занесены данные, и выдает сигнал ЗПЗУ. Этим сигналом информация с ШД записывается в ячейку ОП.

Как при чтении, так и при записи происходит буферизация пересылаемого слова в регистре данных (РД) контроллера. Это необходимо для компенсации различий в скорости работы ОП и ПУ, в силу чего сигналы ВЫВ и ВВ формируются контроллером лишь при получении от ПУ подтверждения о готовности. Буферизация сводится к тому, что после первого этапа слово с ШД заносится в РД, а перед вторым — возвращается из РД на шину данных.

После пересылки каждого слова логика управления прибавляет единицу к содержимому РА (формирует адрес следующей ячейки ОП) и уменьшает на единицу содержимое СД (ведет подсчет переданных слов).

Когда пересылка завершена (при нулевом значении в СД), КПДП снимает сигнал Запрос ПДП, в ответ на что ЦП снимает сигнал Подтверждение ПДП и вновь берет на себя управление системной шиной, то есть ЦП вовлечен в процесс ввода/вывода только в начале и конце передачи.

Эффективность ПДП зависит от организации совместного использования системной шины при пересылке блока. Здесь может применяться один из трех режимов:

- блочная пересылка;
- пропуск цикла;
- прозрачный режим.

При *блочной пересылке* КПДП полностью захватывает системную шину с момента начала пересылки и до момента завершения передачи всего блока. На весь этот период ЦП не имеет доступа к шине.

В режиме *пропуска цикла* КПДП после передачи каждого слова на один цикл шины освобождает системную шину, предоставляя ее на это время процессору. Поскольку КПДП все равно должен ждать готовности ПУ, это позволяет ЦП эффективно распорядиться данным обстоятельством.

В *прозрачном режиме* КПДП имеет доступ к системной шине только в тех циклах, когда ЦП в ней не нуждается. Это обеспечивает наиболее эффективную работу процессора, но может существенно замедлять операцию пересылки блока данных. Здесь многое зависит от решаемой задачи, поскольку именно она определяет интенсивность использования шины процессором.

В отличие от обычного прерывания, в пределах цикла команды имеется несколько точек, где КПДП вправе захватить шину (рис 8.8). Отметим, что это не прерывание: процессору не нужно запоминать контекст задачи.

Рис. 8.8. Точки возможного вмешательства в цикл команды при прямом доступе к памяти и при обычном прерывании

Механизм ПДП может быть реализован различными путями. Некоторые возможности показаны на рис. 8.9.

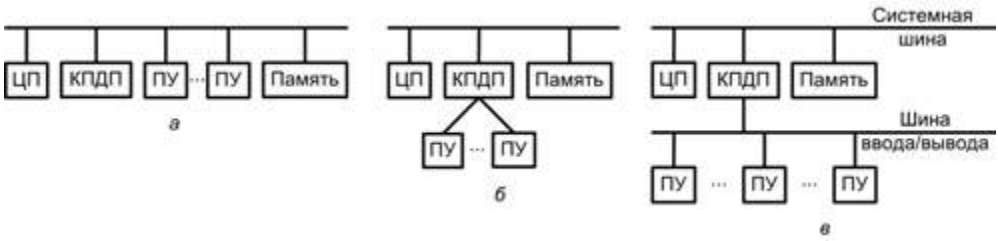


Рис. 8.9. Возможные конфигурации систем прямого доступа к памяти

В первом примере (см. рис. 8.9, *а*) все ПУ совместно используют общую системную шину. КПДП работает как заменитель ЦП и обмен данными между памятью и ПУ производится как ввод/вывод с опросом. Хотя этот вариант может быть достаточно дешевым, эффективность его невысока. Как и в случае ввода/вывода с опросом, осуществляемого процессором, каждая пересылка требует двух циклов шины.

Число необходимых циклов шины может быть уменьшено при объединении функций КПДП и ПУ. Это означает (см. рис. 8.9, *б*), что между КПДП и одним или несколькими ПУ есть другой тракт, не включающий системную шину. Логика ПДП может быть частью ПУ, либо это может быть отдельный КПДП, управляющий одним или несколькими периферийными устройствами. Допустим и еще один шаг в том же направлении (см. рис. 8.9, *в*) — соединение КПДП с ПУ посредством шины ввода/вывода. Это уменьшает число интерфейсов В/ВЫВ в КПДП и делает такую конфигурацию легко расширяемой. В двух последних вариантах системная шина задействуется КПДП только для обмена данными с памятью. Обмен данными между КПДП и ПУ реализуется минуя системную шину.

Каналы и процессоры ввода/вывода

По мере развития систем В/ВЫВ их функции усложняются. Главная цель такого усложнения — максимальное высвобождение ЦП от управления процессами

ввода/вывода. Некоторые пути решения этой задачи уже были рассмотрены. Следующими шагами в преодолении проблемы могут быть:

1. Расширение возможностей МВВ и предоставление ему прав процессора со специализированным набором команд, ориентированных на операции ввода/вывода. ЦП дает указание такому процессору В/ВЫВ выполнить хранящуюся в памяти ВМ программу ввода/вывода. Процессор В/ВЫВ извлекает и исполняет команды этой программы без участия центрального процессора и прерывает ЦП только после завершения всей программы ввода/вывода.
2. Рассмотренному в пункте 1 процессору ввода/вывода придается собственная локальная память, при этом возможно управление множеством устройств В/ВЫВ с минимальным привлечением ЦП.

В первом случае МВВ называют *каналом ввода/вывода* (КВВ), а во втором — *процессором ввода/вывода*. В принципе, различие между каналом и процессором ввода/вывода достаточно условно, поэтому в дальнейшем будем пользоваться термином «канал».

Концепция системы ввода/вывода с КВВ характерна для больших универсальных вычислительных машин (мэйнфреймов), где проблема эффективной организации ввода/вывода и максимального высвобождения центрального процессора в пользу его основной функции стоит наиболее остро. СВВ с каналами ввода/вывода была предложена и реализована в ВМ семейства IBM 360 и получила дальнейшее развитие в семействах IBM 370 и IBM 390.

В ВМ с каналами ввода/вывода центральный процессор практически не участвует в непосредственном управлении периферийными устройствами, делегируя эту задачу специализированному процессору, входящему в состав КВВ. Все функции ЦП сводятся к запуску и остановке операций в КВВ, а также проверке состояния канала и подключенных к нему ПУ. Для этих целей ЦП использует лишь несколько (от 4 до 8) команд ввода/вывода. Например, в IBM 360 таких команд четыре:

- «Начать ввод-вывод»;
- «Остановить ввод-вывод»;
- «Проверить ввод-вывод»;
- «Проверить канал».

КВВ реализует операции ввода/вывода путем выполнения так называемой *канальной программы*. Канальные программы для каждого ПУ, с которым предполагается обмен информацией, описывают нужную последовательность операций ввода-вывода и хранятся в основной памяти ВМ либо локальной памяти канала. Роль команд в канальных программах выполняют *управляющие слова канала* (УСК), структура которых отличается от структуры обычной машинной команды. Типовое УСК содержит:

- *код операции*, определяющий для КВВ и ПУ тип операции: «Записать» (вывод информации из ОП в ПУ), «Прочитать» (ввод информации из ПУ в ОП), «Управление» (перемещение головок НМД, магнитной ленты и т. п.);
- *указатели* — дополнительные предписания, задающие более сложную последовательность операций В/ВЫВ, например при вводе пропускать отдельные

записи или наоборот — с помощью одной команды вводить «разбросанный» по ОП массив как единый;

- *адрес данных*, указывающий область памяти, используемую в операции ввода-вывода;
- *счетчик данных*, хранящий значение длины передаваемого блока данных.

Кроме того, в УСК может содержаться идентификатор ПУ и информация о его уровне приоритета, указания по действиям, которые должны быть произведены при возникновении ошибок и т. п.

Центральный процессор инициирует ввод/вывод, оповещая канал о необходимости выполнить канальную программу, находящуюся в ОП, и указывая начальный адрес этой программы в памяти ВМ. КВВ следует этим указаниям и управляет пересылкой данных. Пересылка информации каналом ведется непосредственно между периферийным устройством (ПУ) и основной памятью, минуя центральный процессор. Таким образом, в вычислительной машине с КВВ управление вводом/выводом строится иерархическим образом. В операциях ввода/вывода участвуют три типа устройств:

- процессор (первый уровень управления);
- канал ввода/вывода (второй уровень);
- периферийное устройство (третий уровень).

Каждому типу устройств соответствует свой вид управляющей информации:

- процессору — команды ввода/вывода;
- каналу — управляющие слова канала;
- периферийному устройству — приказы.

Структура ВМ с канальной системой ввода/вывода показана на рис. 8.10.

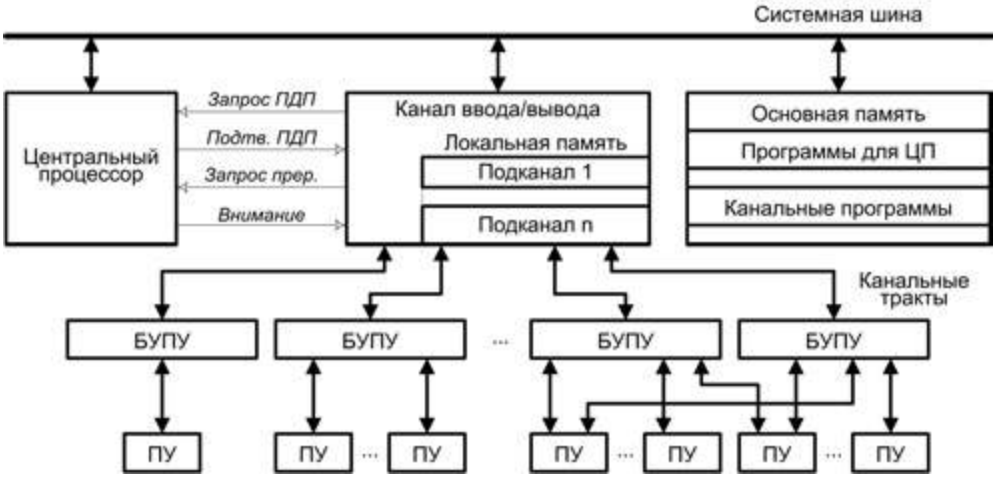


Рис. 8.10. ВМ с канальной системой ввода/вывода

Обмен информацией между КВВ и основной памятью осуществляется посредством системной шины ВМ. ПУ подключаются к каналу не непосредственно, а через блоки управления периферийными устройствами (БУПУ). БУПУ принимает от канала приказы по управлению периферийным устройством (чтение, запись, перемещение носителя или магнитной головки и т. п.) и преобразует их в сигналы управления, свойственные данному типу ПУ. Обычно один БУПУ может обслуживать несколько однотипных ПУ, но для подключения быстродействующих периферийных устройств часто применяются индивидуальные блоки управления. В свою очередь, некоторые ПУ могут подключаться одновременно к нескольким БУПУ. Это позволяет воспользоваться свободным трактом другого БУПУ при занятости данного БУПУ обслуживанием одного из подключенных к нему ПУ. Физически БУПУ может быть самостоятельным устройством или интегрирован с ПУ или каналом.

Обмен информацией между БУПУ и КВВ обеспечивается так называемыми *канальными трактами*. Обычно каждое БУПУ связано с одним из канальных трактов, но возможно также подключение блока управления сразу к нескольким трактам, что дает возможность избежать нежелательных задержек при занятости одного из них.

В пределах канала ввода/вывода считается, что каждое ПУ подключено к своему *подканалу*. Подканалы имеют свои уникальные логические номера, с помощью которых канальная программа адресуется к конкретному ПУ. Физически подканал реализуется в виде участка памяти, в котором хранятся параметры операции ввода/вывода, выполняемой данным ПУ: текущие значения адреса и счетчика данных, код и указатели операции ввода/вывода, адрес следующего УСК и др. Для хранения этих параметров обычно используется локальная память канала.

Обмен информацией между ПУ и ОП, как уже упоминалось, реализуется в режиме прямого доступа к памяти, при этом для взаимодействия ЦП и канала задействованы сигналы Запрос ПДП и Подтверждение ПДП.

Чтобы известить ЦП об окончании текущей канальной программы или об ошибках, возникших при ее выполнении, КВВ выдает в ЦП сигнал Запрос прерывания. В свою очередь, ЦП может привлечь внимание канала сигналом Внимание.

Способ организации взаимодействия ПУ с каналом определяется соотношением быстродействия ОП и ПУ. По этому признаку ПУ образуют две группы: быстродействующие (накопители на магнитных дисках (НМД), накопители на магнитных лентах (НМЛ)) со скоростью приема и выдачи информации около 1 Мбайт/с и медленнодействующие (дисплеи, печатающие устройства и др.) со скоростями порядка 1 Кбайт/с и менее. Быстродействие основной памяти обычно значительно выше. С учетом производительности ПУ в КВВ реализуются два режима работы: *мультиплексный* (режим разделения времени) и *монопольный*.

В *мультиплексном режиме* несколько периферийных устройств разделяют канал во времени, при этом каждое из параллельно работающих с каналом ПУ связывается с КВВ на короткие промежутки времени только после того, как ПУ будет готово к приему или выдаче очередной порции информации (байта, группы байтов и т. д.). Такая схема принята в *мультиплексном канале ввода/вывода*. Если в течение

сеанса связи пересылается один байт или несколько байтов, образующих одно машинное слово, канал называется *байт-мультиплексным*. Канал, в котором в пределах сеанса связи пересылка данных выполняется поблочно, носит название *блок-мультиплексного*.

В *монопольном режиме* после установления связи между каналом и ПУ последнее монополизирует канал на все время до завершения инициированной процессором канальной программы и всех предусмотренных этой программой пересылок данных между ПУ и ОП. На все время выполнения канальной программы канал оказывается недоступным для других ПУ. Данную процедуру обеспечивает *селекторный канал ввода/вывода*. Отметим, что в блок-мультиплексном канале в рамках сеанса связи пересылка блока осуществляется в монопольном режиме.

Канальная подсистема

В последовавших за IBM 360 семействах универсальных ВМ IBM 370 и особенно, IBM 390 концепция системы ввода/вывода на базе каналов получила дальнейшее развитие и вылилась в так называемую *канальную подсистему ввода/вывода* (КПВВ). Главная идея заключается в интегрировании отдельных КВВ в единый специализированный процессор ввода/вывода с большим числом канальных трактов и подканалов. Блоки управления периферийными устройствами обычно подключаются к нескольким канальным трактам, что позволяет динамически менять путь пересылки информации с учетом текущей их загруженности. Кроме того, разные канальные тракты могут обладать различной пропускной способностью и при выборе трактов для подключения определенных ПУ может быть учтено их быстродействие.

Одной из наиболее совершенных канальных подсистем обладают ВМ семейства IBM 390. В ней предусмотрено использование до 65 536 подканалов и до 256 канальных трактов. Реализованы два типа канальных трактов: параллельный и последовательный.

Параллельные канальные тракты по своим возможностям и принципу действия аналогичны рассмотренным ранее мультиплексному и селекторному каналам, но в отличие от них являются универсальными, то есть могут работать в байт-мультиплексном, блок-мультиплексном и монопольном режимах. Такие канальные тракты в КПВВ называют параллельными, поскольку они обеспечивают пересылку информации параллельным кодом.

Для работы с ПУ, соединенными с КПВВ волоконно-оптическими линиями, используются *последовательные канальные тракты*, реализующие протокол ESCON (Enterprise Systems Connection Architecture). Последовательный канальный тракт рассчитан на передачу информации только в последовательном коде и только в монопольном режиме. Для подключения блоков управления периферийными устройствами (БУПУ) к ESCON-тракту служат специальные устройства, называемые ESCON-директорами. Каждое такое устройство может обеспечить одновременное подключение до 60 БУПУ и одновременную передачу информации от 30 из них со скоростью до 10 Мбайт/с.

Кроме того, в КПВВ предусмотрены специальные коммуникационные канальные тракты для подключения к сетям ВМ, модемам, другим системам.

В принципе, основное преимущество КПВВ — динамическое перераспределение канальных трактов — в какой-то мере может быть реализовано и в рамках каждого отдельного канала. Однако объединение всех канальных ресурсов в единую канальную подсистему позволяет применить оптимальную стратегию динамического распределения и использования этих ресурсов и благодаря этому достичь качественно нового уровня эффективности системы ввода/вывода.

Контрольные вопросы

1. Поясните достоинства и недостатки трех вариантов подключения системы ввода/вывода к процессору ВМ.
2. Сформулируйте достоинства, недостатки и область применения двух способов организации адресного пространства ввода/вывода.
3. Дайте развернутую характеристику структуры ПУ, отображая ее элементы в каждый из трех типов ПУ.
4. В чем состоит локализация данных, выполняемая модулем ввода/вывода?
5. Опишите содержание процедуры «рукопожатия» при выполнении операции ввода.
6. Конкретизируйте последовательность действий процессора при обмене информацией с жестким диском.
7. Выберите конкретную скорость работы ЦП. Рассчитайте емкость буферной памяти МВВ для обмена с клавиатурой, символьным принтером и оптическим диском.
8. Проведите маленькое исследование: спрогнозируйте вероятность возникновения ошибки мыши, лазерного принтера, оптического диска. Ответы обоснуйте.
9. Для конкретного ЦП определите структуры МВВ для мыши, клавиатуры, жесткого диска. Необходимость элементов структуры обоснуйте.
10. Сравните известные вам методы управления вводом/выводом по трем параметрам: достоинствам, недостаткам, области применения.
11. Поясните классификацию методов ввода/вывода по прерыванию.
12. Охарактеризуйте известные вам режимы прямого доступа к памяти, сформулируйте их достоинства и недостатки.
13. Опишите процесс вывода пяти слов и ввода семи слов при трех вариантах реализации ПДП.
14. Сравните ввод/вывод по прерыванию с вводом/выводом при ПДП. Для какого режима ПДП эти методы наиболее близки и почему?
15. Проведите сравнительный анализ контроллера ПДП и канала ввода/вывода. В чем их сходство? Чем они отличаются друг от друга?
16. Опишите процесс взаимодействия ЦП и КВВ. Какая при этом используется управляющая информация?
17. Опишите задачи посредника между КВВ и ПУ.

Глава 9

Процессоры

Ранее уже были рассмотрены основные составляющие центрального процессора. В данной главе основное внимание уделено вопросам общей архитектуры процессоров как единого устройства и способам повышения их производительности.

Конвейеризация вычислений

Совершенствование элементной базы уже не приводит к кардинальному росту производительности ВМ. Более перспективными в этом плане представляются архитектурные приемы, среди которых один из наиболее значимых — *конвейеризация*.

Для пояснения идеи конвейера сначала обратимся к рис. 9.1, *а*, где показан отдельный функциональный блок (ФБ). Исходные данные помещаются во входной регистр $Рг_{вх}$, обрабатываются в функциональном блоке, а результат обработки фиксируется в выходном регистре $Рг_{вых}$. Если максимальное время обработки в ФБ равно T_{max}, то новые данные могут быть занесены во входной регистр $Р_{вх}$ не ранее, чем спустя T_{max}.

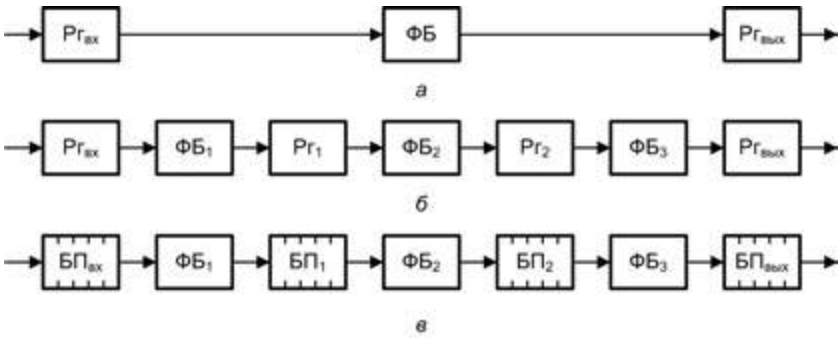


Рис. 9.1. Обработка информации: *а* — в одиночном блоке; *б* — в конвейере с регистрами; *в* — в конвейере с буферной памятью

Теперь распределим функции, выполняемые в функциональном блоке ФБ, между тремя независимыми блоками, образующими последовательность: $ФБ_1$, $ФБ_2$ и $ФБ_3$, причем так, чтобы максимальное время обработки в каждом $ФБ_i$ было одинаковым и равнялось $\frac{T_{max}}{3}$. Между блоками разместим буферные регистры $Рг_i$ (рис. 9.1, *б*), предназначенные для хранения результата обработки в $ФБ_i$, на случай, если следующий за ним функциональный блок еще не готов использовать этот результат. В рассмотренной схеме данные на вход конвейера могут подаваться с интервалом $\frac{T_{max}}{3}$ (втрое чаще), и хотя задержка от момента поступления первой единицы данных в $Рг_{вх}$ до момента появления результата ее обработки на выходе $Рг_{вых}$ по-прежнему составляет $T_{max,}$ последующие результаты появляются на выходе $Рг_{вых}$ уже с интервалом $\frac{T_{max}}{3}$.

На практике редко удается добиться того, чтобы задержки в каждом $ФБ_i$ были одинаковыми, и тогда производительность конвейера снижается, поскольку период поступления входных данных должен быть не меньше, чем время обработки в самом «медленном» функциональном блоке. Для устранения этого недостатка или, по крайней мере, частичной его компенсации каждый буферный регистр $Рг_i$ следует заменить буферной памятью $БП_i$, способной хранить множество данных и организованной по принципу FIFO — «первым вошел — первым вышел» (рис. 9.1, *в*). Обработав элемент данных, $ФБ_i$ заносит результат в $БП_i$, извлекает из $БП_{i-1}$ новый элемент данных и приступает к очередному циклу обработки, причем эта последовательность осуществляется каждым функциональным блоком независимо от других блоков. Обработка в каждом блоке может продолжаться до тех пор, пока не ликвидируется предыдущая очередь или пока не переполнится следующая очередь. Если емкость буферной памяти достаточно велика, различия во времени обработки не сказываются на производительности, тем не менее желательно, чтобы средняя длительность обработки во всех $ФБ_i$ была одинаковой.

По способу синхронизации работы ступеней конвейеры могут быть синхронными и асинхронными. Для традиционных ВМ характерны *синхронные конвейеры*. Ступени конвейеров в процессоре обычно располагаются близко друг от друга, благодаря чему тракты распространения сигналов синхронизации получаются достаточно короткими и фактор «перекоса» сигналов[1] становится не столь существенным. *Асинхронные конвейеры* оказываются полезными, если связь между ступенями не столь сильна, а длина сигнальных трактов между разными ступенями сильно изменяется. Примером асинхронных конвейеров могут служить систолические массивы (систолическая обработка будет рассмотрена в главе 13).

Синхронные линейные конвейеры

Эффективность синхронного конвейера во многом зависит от правильного выбора длительности тактового периода $T_К$. Минимально допустимую величину $T_К$ можно

[1] Понятие «перекос» сигнала было пояснено в разделе «Протокол шины» главы 7.

определить как сумму наибольшего из времен обработки на отдельной ступени конвейера T_{CMAX} и времени записи результатов обработки в буферные регистры между ступенями конвейера $T_{БР}$:

$$T_К = T_{CMAX} + T_{БР}.$$

Из-за вероятного «перекоса» в поступлении тактирующего сигнала в разные ступени конвейера предыдущую формулу следует дополнить еще одним элементом — максимальной величиной «перекоса» $T_{ПК}$:

$$T_К = T_{CMAX} + T_{БР} + T_{ПК}.$$

Каждая ступень может содержать множество логических трактов обработки. $T_К$ определяется наиболее длинными трактами во всех ступенях конвейера. При разработке конвейера необходимо учитывать, что для двух смежных элементов, обрабатываемых одной и той же ступенью, обработка первого элемента может проходить по тракту максимальной длины, а второго элемента — по минимальному тракту. В итоге результат обработки второго элемента может появиться на выходе ступени прежде, чем в выходном регистре ступени будет запомнен предыдущий результат. Чтобы избежать подобной ситуации, сумма $T_{БР} + T_{ПК}$ не должна превышать минимальное время обработки в ступени T_{CMIN}, откуда

$$T_{БР} \leqslant T_{CMIN} - T_{ПК}.$$

Выбор длительности тактового периода для конвейера должен осуществляться с соблюдением соотношения:

$$T_{CMAX} + T_{БР} + T_{ПК} \leqslant T_К \leqslant T_{CMAX} + T_{CMIN} - T_{ПК}.$$

Несмотря на очевидные преимущества выбора $T_К$ равным нижнему пределу, проектировщики ВМ обычно ориентируются на среднее значение между нижним и верхним пределами.

Метрики эффективности конвейеров

Чтобы охарактеризовать эффект, достигаемый за счет конвейеризации вычислений, обычно используют три метрики: ускорение, эффективность и производительность.

Под *ускорением* понимается отношение времени обработки без конвейера и времени обработки при наличии конвейера. Теоретически наилучшее время обработки входного потока из N значений T_{NK} на конвейере с K ступенями и тактовым периодом $T_К$ можно определить выражением:

$$T_{NK} = (K + (N - 1))T_К.$$

Формула отражает тот факт, что до появления на выходе конвейера результата обработки первого элемента должно пройти K тактов, а последующие результаты будут следовать в каждом такте.

В процессоре без конвейера общее время выполнения составляет $NKT_К$. Таким образом, ускорение вычислений S за счет конвейеризации вычислений можно описать формулой:

$$S = \frac{NKT_K}{(K + (N-1))T_K} = \frac{NK}{K + (N-1)}.$$

При $N \to \infty$ ускорение стремится к величине K, равной количеству ступеней в конвейере.

Еще одной метрикой, характеризующей конвейерный процессор, является *эффективность* E — доля ускорения, приходящаяся на одну ступень конвейера:

$$E = \frac{S}{K} = \frac{N}{K + (N-1)}.$$

В качестве третьей метрики часто выступает *пропускная способность* или *производительность* P — эффективность, деленная на длительность тактового периода:

$$P = \frac{N}{T_K(K + (N-1))}.$$

При $N \to \infty$ эффективность стремится к единице, а производительность — к частоте тактирования конвейера F $\left(F = \frac{1}{T_K}\right)$.

Нелинейные конвейеры

Конвейер не всегда представляет собой линейную цепочку этапов. В ряде ситуаций оказывается выгодным, когда функциональные блоки соединены между собой не последовательно, а в соответствии с логикой обработки, при этом одни блоки в цепочке могут пропускаться, а другие — образовывать циклические структуры. Это позволяет с помощью одного и того же конвейера одновременно вычислять более одной функции, однако если эти функции конфликтуют между собой, то конвейер трудно загрузить полностью. Структура нелинейного конвейера, одновременно вычисляющего две функции X и Y, приведена на рис. 9.2.

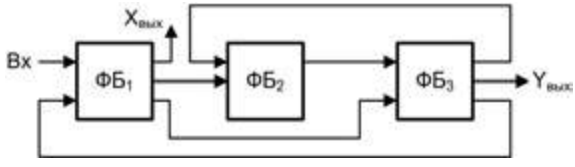


Рис. 9.2. Нелинейный конвейер

Конвейер команд

В архитектуре вычислительных машин можно найти множество объектов, где конвейеризация обеспечивает существенный прирост производительности ВМ, например операционные устройства и память, однако наиболее ощутимый эффект достигается при конвейеризации этапов машинного цикла. Идея конвейера команд была

предложена в 1956 году академиком С. А. Лебедевым. Как известно, цикл команды представляет собой последовательность этапов. Возложив реализацию каждого из них на самостоятельное устройство и последовательно соединив такие устройства, мы получим классическую схему конвейера команд. На рис. 9.3 показан конвейер с семью ступенями, соответствующими семи этапам цикла команды, рассмотренным в главе 3:

- выборка команды (ВК);
- декодирование команды (ДК);
- вычисление исполнительных адресов (ВА);
- выборка операндов (ВО);
- исполнение операции (ИО);
- запись результата (ЗР);
- формирование адреса следующей команды (ФАСК).

Рис. 9.3. Логика работы конвейера команд

В диаграмме предполагается, что каждая команда обязательно проходит все семь ступеней, хотя этот случай не совсем типичен. Так, команда загрузки регистра не требует этапа ЗР. Кроме того, здесь принято, что все этапы могут выполняться одновременно. Без конвейеризации выполнение девяти команд заняло бы $9 \times 7 = 63$ единицы времени. Использование конвейера позволяет сократить время обработки до 15 единиц.

Конфликты в конвейере команд

Полученное в примере число 15 характеризует лишь потенциальную производительность конвейера команд. На практике, в силу возникающих в конвейере конфликтных ситуаций, достичь такой производительности не удается. Конфликтные ситуации в конвейере принято обозначать термином *риск* (hazard), а обусловлены они могут быть тремя причинами:

- попыткой нескольких команд одновременно обратиться к одному и тому же ресурсу ВМ (структурный риск);

- взаимосвязью команд по данным (риск по данным);
- неоднозначностью при выборке следующей команды в случае команд, изменяющих последовательность вычислений (риск по управлению).

Структурный риск (конфликт по ресурсам) имеет место, когда несколько команд, находящихся на разных ступенях конвейера, пытаются одновременно использовать один и тот же ресурс, чаще всего — память. Так, в типовом цикле команды (см. рис. 9.3) сразу три этапа (ВК, ВО и ЗР) связаны с обращением к памяти. Подобных конфликтов частично удается избежать за счет блочной организации основной памяти и использования кэш-памяти — имеется вероятность того, что команды будут обращаться либо к разным банкам ОП, либо одна из них станет обращаться к основной памяти, а другая — к кэш-памяти. С этих позиций кэш-память выгоднее разделять на кэш-память команд и кэш-память данных. Конфликты из-за одновременного обращения к памяти могут и не возникать, поскольку для многих команд этапы выборки операнда и записи результата не требуются. В целом, влияние структурного риска на производительность конвейера по сравнению с другими видами рисков сравнительно невелико.

Риск по данным, в противоположность структурному риску, — типичная и регулярно возникающая ситуация. Для пояснения сущности взаимосвязи команд по данным положим, что две команды в конвейере (*i* и *j*) предусматривают обращение к одной и той же переменной *x*, причем команда *i* предшествует команде *j*. В общем случае между *i* и *j* ожидаются три типа конфликтов по данным (рис. 9.4):

- «*чтение после записи*» (ЧПЗ): команда *j* читает *x* до того, как команда *i* успела записать новое значение *x*, то есть команда *j* ошибочно получит старое значение *x* вместо нового.
- «*запись после чтения*» (ЗПЧ): команда *j* записывает новое значение *x* до того, как команда *i* успела прочитать *x*, то есть команда *i* ошибочно получит новое значение *x* вместо старого.
- «*запись после записи*» (ЗПЗ): команда *j* записывает новое значение *x* прежде, чем команда *i* успела записать в качестве *x* свое значение, и окончательно *x* ошибочно будет содержать значение, определенное командой *i*, а не командой *j*.

Возможен и четвертый случай, когда команда *j* читает *x* прежде команды *i*. Этот случай не вызывает никаких конфликтов, поскольку как *i*, так и *j* получат верное значение *x*.

Наиболее частый вид конфликтов по данным — ЧПЗ, поскольку операция чтения в цикле команды (этап ВО) предшествует операции записи (этап ЗР). По той же причине конфликты типа ЗПЧ большой проблемы не представляют. Сложности появляются, только если структура конвейера допускает запись прежде чтения или если команды в конвейере обрабатываются в последовательности, отличной от предписанной программой. Такое возможно, если командам в конвейере разрешается «догонять» предшествующие им команды, приостановленные из-за какого-то конфликта. Конфликт типа ЗПЗ также не вызывает особых проблем в конвейерах, где команды следуют в порядке, определенном программой, и могут производить запись только на этапе ЗР. В худшем случае, когда одна команда догоняет другую

из-за приостановки последней, имеет место конфликт по ресурсу — попытка одновременного доступа к одной и той же ячейке.

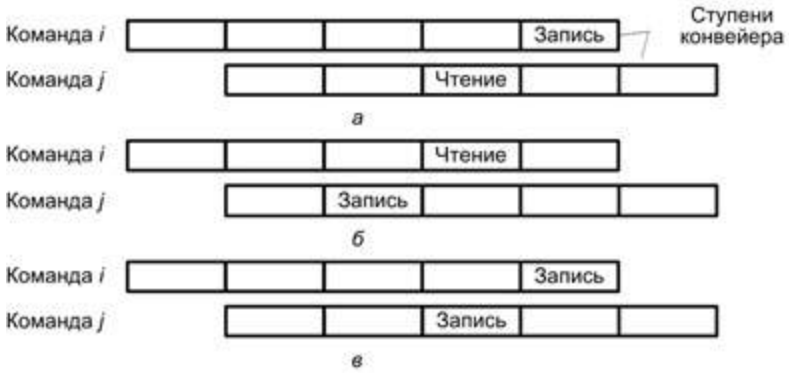


Рис. 9.4. Конфликты по данным: *а* — «чтение после записи»; *б* — «запись после чтения»; *в* — «запись после записи»

В борьбе с конфликтами по данным выделяют два аспекта: своевременное обнаружение потенциального конфликта и его устранение. Признаком возникновения конфликта по данным между двумя командами i и j служит невыполнение хотя бы одного из трех условий Бернстейна (Bernstein's Conditions):

- для ЧПЗ: $O(i) \cap I(j) = \varnothing$;
- для ЗПЧ: $I(i) \cap O(j) = \varnothing$;
- для ЗПЗ: $O(i) \cap O(j) = \varnothing$.

Здесь $O(k)$ — множество ячеек, изменяемых командой k; $I(l)$ — множество ячеек, читаемых командой l; $\varnothing$ — пустое множество; $\cap$ — операция пересечения множеств.

Условия можно распространить и на большее число команд: для трех команд подобных уравнений будет 9, для четырех команд — 18 (по три на каждую пару). Соблюдение соотношений является достаточным, но не необходимым условием, поскольку в ряде случаев конфликты могут и не возникать.

Для борьбы с конфликтами по данным применяются как программные, так и аппаратные методы.

Программные методы ориентированы на устранение самой возможности конфликтов еще на стадии компиляции программы. Оптимизирующий компилятор пытается создать такой объектный код, чтобы между командами, склонными к конфликтам, находилось достаточное количество нейтральных в этом плане команд программы. Если такое не удается, то между конфликтующими командами компилятор вставляет необходимое количество команд типа «Нет операции».

Фактическое разрешение конфликтов возлагается на аппаратные методы. Наиболее очевидным решением является остановка команды j на несколько тактов с тем, чтобы команда i успела завершиться или, по крайней мере, миновать ступень конвейера, на которой возникает конфликт. Соответственно задерживаются

и команды, следующие в конвейере за *j*-й командой. Данную ситуацию называют «*пузырьком*» в конвейере. Иногда приостанавливают только команду *j*, не задерживая следующие за ней команды. Это более эффективный прием, но его реализация усложняет конвейер.

Понятно, что остановки конвейера снижают его эффективность, и разработчики ВМ всячески стремятся сократить общее число остановок или хотя бы их длительность. Поскольку наиболее частые конфликты по данным — это ЧПЗ, основные усилия тратятся на противодействие именно этому типу конфликтов. Среди известных методов борьбы с ЧПЗ наибольшее распространение получил прием *ускоренного продвижения информации* (forwarding). Обычно между двумя соседними ступенями конвейера располагается буферный регистр, через который предшествующая ступень передает результат своей работы на последующую ступень, то есть передача информации возможна лишь между соседними ступенями конвейера. При ускоренном продвижении, когда для выполнения команды требуется операнд, уже вычисленный предыдущей командой, этот операнд может быть получен непосредственно из соответствующего буферного регистра, минуя все промежуточные ступени конвейера. С данной целью в конвейере предусматриваются дополнительные тракты пересылки информации (тракты опережения, тракты обхода), снабженные средствами коммутации трактов.

Наибольшие проблемы при создании эффективного конвейера обусловлены командами, изменяющими естественный порядок вычислений[1]. Простейший конвейер ориентирован на линейные программы. В нем ступень выборки извлекает команды из ячеек памяти с последовательными адресами, используя для этого счетчик команд (СК). Адрес очередной команды в линейной программе формируется автоматически, за счет прибавления к содержимому СК числа, равного длине текущей команды в байтах. Реальные программы практически никогда не бывают линейными. В них обязательно присутствуют команды управления, изменяющие последовательность вычислений: безусловный и условный переход, вызов процедуры, возврат из процедуры и т. п. Доля подобных команд в программе оценивается как 10–20% (по некоторым источникам она существенно больше). Выполнение команд, изменяющих последовательность вычислений (в дальнейшем будем их называть командами перехода), может приводить к приостановке конвейера на несколько тактов, из-за чего производительность процессора снижается. Приостановки конвейера при выполнении команд перехода обусловлены двумя факторами.

Первый фактор характерен для любой команды перехода и связан с выборкой команды из точки перехода (по адресу, указанному в команде перехода). То, что текущая команда относится к командам перехода, становится ясным только после прохождения ступени декодирования, что для конвейера, взятого в качестве примера, происходит спустя два такта от момента поступления команды на конвейер. За это время на первые ступени конвейера уже поступят новые команды, извлеченные в предположении, что естественный порядок вычислений не будет нарушен. В случае

[1] В фон-неймановской ВМ команды размещаются в ячейках памяти и извлекаются для выполнения в том же порядке, в каком они следуют в программе. Такую последовательность выполнения команд программы называют естественной.

перехода эти начальные ступени нужно очистить и загрузить в конвейер команду, расположенную по адресу перехода, для чего требуется исполнительный адрес последней. Поскольку в командах перехода обычно указаны лишь способ адресации и адресный код, исполнительный адрес предварительно должен быть вычислен, что и делается на третьей ступени конвейера. Таким образом, реализация перехода в конвейере требует определенных дополнительных операций, выполнение которых равносильно остановке конвейера, как минимум, на два такта.

Вторая причина нарушения ритмичности работы конвейера имеет отношение только к командам условного перехода. Для пояснения сути проблемы воспользуемся ранее приведенным примером (см. рис. 9.3), несколько изменив постановку задачи (рис. 9.5).

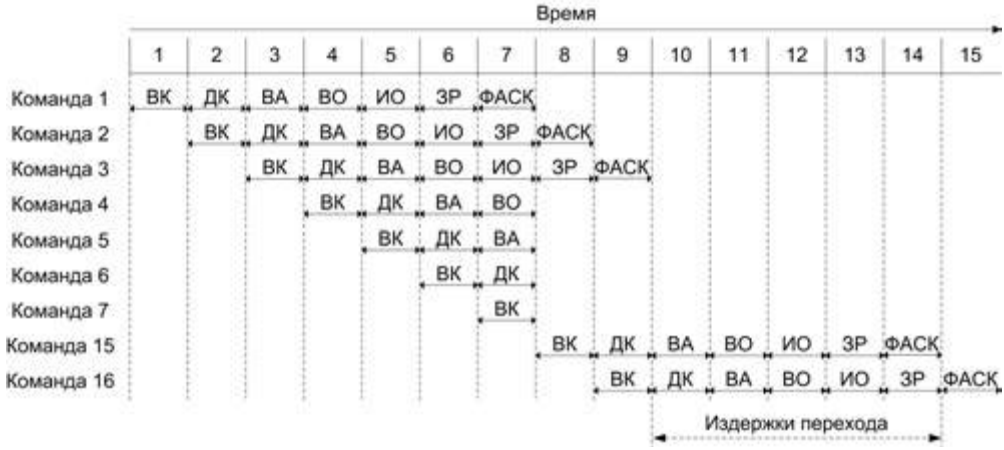


Рис. 9.5. Влияние команды условного перехода на работу конвейера команд

Пусть команда 3 — это условный переход к команде 15. До завершения команды 3 невозможно определить, какая из команд (4-я или 15-я) должна выполняться следующей, поэтому конвейер просто загружает следующую команду в последовательности (команду 4) и продолжает свою работу. В варианте, показанном на рис. 9.3, переход не произошел, и получена максимально возможная производительность. На рис. 9.5 переход имеет место, о чем неизвестно до 7-го шага. В этой точке конвейер должен быть очищен от ненужных команд, выполнявшихся до данного момента. Лишь на шаге 8 на конвейер поступает нужная команда 15, из-за чего в течение тактов от 10 до 14 не будет завершена ни одна другая команда. Это и есть издержки из-за невозможности предвидения исхода команды условного перехода. Как видно, они существенно больше, чем для команд безусловного перехода (когда анализировать условие перехода нет нужды). Если же условие перехода не подтверждается, издержки вовсе отсутствуют.

Выборка команды из точки перехода

Для сокращения задержек, обусловленных выборкой команды из точки перехода, применяются несколько подходов:

- вычисление исполнительного адреса перехода на ступени декодирования команды;
- использование буфера адресов перехода;
- использование кэш-памяти для хранения команд, расположенных в точке перехода;
- использование буфера цикла.

При первом из перечисленных подходов в результате декодирования команды выясняется не только ее принадлежность к командам перехода, но также способ адресации и адресный код точки перехода. Это позволяет сразу же приступить к вычислению исполнительного адреса перехода, не дожидаясь передачи команды на третью ступень конвейера, и тем самым сократить время остановки конвейера с двух тактов до одного. Для реализации этой идеи в состав ступени декодирования вводятся дополнительные сумматоры, с помощью которых и вычисляется исполнительный адрес точки перехода.

Второй подход предполагает использование буфера адресов перехода (BTB, Branch Target Buffer), представляющего собой кэш-память небольшой емкости. В BTB хранятся исполнительные адреса точек перехода, которые были вычислены при выполнении нескольких последних команд, причем тех, что завершились переходом. В роли тегов выступают адреса соответствующих команд. Перед выборкой очередной команды ее адрес (содержимое счетчика команд) сравнивается с адресами команд, представленных в BTB. Для команды, найденной в буфере адресов перехода, исполнительный адрес точки перехода не вычисляется, а берется из BTB, благодаря чему выборка команды из точки перехода может быть начата на один такт раньше. Команда, ссылка на которую в BTB отсутствует, обрабатывается стандартным образом. Если это команда перехода, то полученный при ее выполнении исполнительный адрес точки перехода заносится в BTB, при условии, что команда завершилась переходом. При замещении информации в BTB обычно применяется алгоритм LRU, рассмотренный ранее применительно к кэш-памяти.

Применение BTB дает наибольший эффект, когда отдельные команды перехода в программе выполняются многократно, что типично для циклов. Обычно BTB используется не самостоятельно, а в составе других, более сложных схем компенсации конфликтов по управлению.

Кэш-память команд, расположенных в точке перехода (BTIC, Branch Target Instruction Cache), — это усовершенствованный вариант BTB, где в кэш-память помимо исполнительного адреса команды в точке перехода записывается также и код этой команды. За счет увеличения емкости BTIC позволяет при повторном выполнении команды перехода исключить не только этап вычисления исполнительного адреса точки перехода, но и этап выборки расположенной там команды. Преимущества данного подхода в наибольшей степени проявляются при многократном исполнении одних и тех же команд перехода, главным образом при реализации программных циклов.

Буфер цикла представляет собой маленькую быстродействующую память, входящую в состав первой ступени конвейера, где производится выборка команд.

В буфере сохраняются коды *n* последних команд в той последовательности, в которой производилась выборка команд. Когда имеет место переход, аппаратура сначала проверяет, нет ли нужной команды в буфере, и если это так, то команда извлекается из буфера. Стратегия наиболее эффективна при реализации циклов и итераций, чем и объясняется название буфера. Если буфер достаточно велик, чтобы охватить все тело цикла, выборку команд из памяти достаточно выполнить только один раз в первой итерации, поскольку необходимые для последующих итераций команды уже находятся в буфере.

По принципу использования буфер цикла похож на BTIC, с той разницей, что в нем сохраняется последовательность выполнения команд, а сам буфер меньше по емкости и дешевле.

Среди ВМ, где реализован буфер цикла, можно упомянуть некоторые вычислительные машины фирмы CDC (Star 100, 6600, 7600) и суперЭВМ CRAY-1. Специализированная версия буфера цикла имеется и в микропроцессоре Motorola 68010, где буфер используется для особых циклов, включающих в себя команду «Уменьшение и переход по условию».

Методы решения проблемы условного перехода

Несмотря на важность аспекта вычисления исполнительного адреса точки перехода, основные усилия проектировщиков ВМ направлены на решение проблемы условных переходов, поскольку именно последние приводят к наибольшим издержкам в работе конвейера. Для устранения или частичного сокращения этих издержек были предложены различные подходы, из которых наиболее актуальна идея предсказания перехода.

Предсказание переходов

Предсказание переходов на сегодняшний день рассматривается как один из наиболее эффективных способов борьбы с конфликтами по управлению. Идея заключается в том, что еще до момента выполнения команды условного перехода или сразу же после ее поступления на конвейер делается предположение о наиболее вероятном исходе такой команды (переход произойдет или не произойдет). Последующие команды подаются на конвейер в соответствии с предсказанием. Для иллюстрации вернемся к примеру (см. рис. 9.5), где команда 3 является командой условного перехода к команде 15. Пусть для команды 3 предсказано, что переход не произойдет. Тогда вслед за командой 3 на конвейер будут поданы команды 4, 5, 6 и т. д. Если предсказан переход, то после команды 3 на конвейер подаются команды 15, 16, 17, ... При ошибочном предсказании конвейер необходимо вернуть к состоянию, с которого началась выборка «ненужных» команд (очистить начальные ступени конвейера), и приступить к загрузке, начиная с «правильной» точки, что по эффекту эквивалентно приостановке конвейера. Цена ошибки может оказаться достаточно высокой, но при правильных предсказаниях существенен и выигрыш — конвейер функционирует ритмично без остановок и задержек, причем выигрыш тем больше, чем выше точность предсказания. Термин «*точность*

предсказания» в дальнейшем будем трактовать как процентное отношение числа правильных предсказаний к их общему количеству. В работе [62] дается следующая оценка: чтобы снижение производительности конвейера из-за его остановок по причине конфликтов по управлению не превысило 10%, точность предсказания переходов должна быть выше 97,7%.

К настоящему моменту известно более двух десятков различных способов реализации идеи предсказания переходов [119, 167–169], отличающихся друг от друга исходной информацией, на основании которой делается прогноз, сложностью реализации и, главное, точностью предсказания. При классификации схем предсказания переходов обычно выделяют два подхода: статический и динамический, в зависимости от того, когда и на базе какой информации делается предсказание.

Количественные показатели эффективности большинства из приводимых в учебнике методов предсказания переходов базируются на результатах исследований, опубликованных в [61, 82, 87, 141, 149, 166].

Статическое предсказание переходов

Статическое предсказание переходов осуществляется на основе некоторой априорной информации о подлежащей выполнению программе. Предсказание делается на этапе компиляции программы и в процессе вычислений уже не меняется. Главное различие между известными механизмами статического прогнозирования заключается в виде информации, используемой для предсказания, и ее трактовке. Исходная информация может быть получена двумя путями: на основе анализа кода программы или в результате ее профилирования (термин «профилирование» поясняется ниже).

Известные стратегии статического предсказания для команд УП можно классифицировать следующим образом:

- переход происходит всегда (ПВ);
- переход не происходит никогда (ПН);
- предсказание определяется по результатам профилирования;
- предсказание определяется кодом операции команды перехода;
- предсказание зависит от направления перехода;
- при первом выполнении команды переход имеет место всегда.

В первом из перечисленных вариантов предполагается, что *каждая команда условного перехода в программе обязательно завершится переходом*, и, с учетом такого предсказания, дальнейшая выборка команд производится, начиная с адреса перехода. В основе второй стратегии лежит прямо противоположный подход: *ни одна из команд условного перехода в программе никогда не завершается переходом*, поэтому выборка команд продолжается в естественной последовательности.

Интуитивное представление, что обе стратегии должны приводить к верному предсказанию в среднем в 50% случаев, на практике не подтверждается. Так, по результатам экспериментов, приведенным в [141], предсказание об обязательном переходе оказалось правильным в среднем для 76% команд УП, а по данным [107] — 68%.

В других источниках встречаются и иные численные оценки точности прогноза при стратегии ПВ, в частности 60% и 71,9%. В то же время в работе [170] отмечается, что для ряда программ, связанных с целочисленной обработкой, процент правильных предсказаний может опускаться ниже 50%. Цифры свидетельствуют, что успешность стратегии ПВ существенно зависит от характера программы и методов программирования, что, естественно, можно рассматривать как недостаток. Тем не менее стратегия все же используется в ряде ВМ, в частности MIPS-X, SuperSPARC, микропроцессорах i486 фирмы Intel. Связано это, скорее всего, с простотой реализации и с тем, что для определенного класса программ стратегию можно считать достаточно эффективной.

Схожая ситуация характерна и для стратегии ПН, где предполагается, что ни одна из команд УП в программе никогда не завершается переходом. Несмотря на схожесть с ПВ, процент правильных исходов здесь обычно ниже, особенно в программах с большим количеством циклов. Стратегия ПН реализована в конвейерах микропроцессоров M68020 и MC88000, вычислительной машине VAX 11/780.

Сравнительная оценка стратегий ПВ и ПН показывает, что ни одна из них не имеет явных преимуществ над другой. Тем не менее анализ большого количества программ [110] показывает, что условные переходы имеют место более чем в 50% случаев, поэтому если стоимость реализации двух рассмотренных вариантов одинакова, то предпочтение следует отдать стратегии ПВ.

В третьем из перечисленных способов статического предсказания назначение командам УП наиболее вероятного исхода производится *по результатам профилирования подлежащих выполнению программ*. Под *профилированием* подразумевается выполнение программы при некотором эталонном наборе исходных данных, сопровождающееся сбором информации об исходах каждой команды условного перехода. Тем командам, которые чаще завершались переходом, назначается стратегия ПВ, а всем остальным — ПН. Выбор фиксируется в специальном бите кода операции команд УП. Некоторые компиляторы самостоятельно проводят профилирование программы и по его результатам устанавливают этот бит в формируемом объектном коде. При выполнении программы поведение конвейера команд определяется после выборки команды по состоянию упомянутого бита. Основной недостаток метода связан с тем, что изменение набора исходных данных для профилирования может существенно менять поведение одних и тех же команд условного перехода. Точность правильного предсказания, по данным исследований, достигает 75%. Стратегия используется в процессорах MIPS 4x00 и PowerPC 603.

При предсказании *на основе кода операции команды перехода* для одних команд предполагается, что переход произойдет, для других — его не случится. Например, для команды перехода по переполнению логично предположить, что перехода не будет. На практике назначение разным видам команд УП наиболее вероятного исхода чаще производится по результатам профилирования программ.

Эффективность предсказания зависит от характера программы. Этим, в частности, объясняется различие в выводах, полученных на основе разных исследований. Так, согласно [110] рассматриваемая стратегия приводит к успеху в среднем более чем в 75% случаев, а иные исследования [141] дают цифру 86,7%.

Достаточно логичным представляется предсказание *исходя из направления перехода*. Если адрес выполняемой команды УП (содержимое счетчика команд) больше указанного в этой команде адреса точки перехода, говорят о переходе «назад», и для такой команды назначается стратегия ПВ. Переход к адресу, превышающему адрес команды перехода, считается переходом «вперед», и такой команде назначается стратегия ПН. В основе рассматриваемого подхода лежит статистика по множеству программ, согласно которой большинство команд УП в программах используются для организации циклов, причем, как правило, переходы происходят к началу цикла (переходы «назад»). Согласно [93], для команд, выполняющих переход «назад», фактический переход имеет место в 85% случаев. Для переходов «вперед» эта цифра составляет 65%. Таким образом, для команд условного перехода «назад» логично принять, что переход происходит всегда. Среди ВМ, где описанная стратегия является основной, можно упомянуть MicroSPARC-2 и PA-7x00.

В последней из рассматриваемых статических стратегий при *первом выполнении любой команды условного перехода делается предсказание, что переход обязательно будет*. Предсказания на последующее выполнение команды зависят от правильности начального предсказания. Стратегию можно считать статической только частично, поскольку она однозначно определяет исход команды лишь при первом ее выполнении. Далее стратегия становится динамической. Точность прогноза в соответствии с данной стратегией выше, чем у всех предшествующих, что подтверждают результаты, приведенные в [141]. К сожалению, при больших объемах программ вариант практически нереализуем из-за того, что нужно отслеживать слишком много команд условного перехода.

Динамическое предсказание переходов

В *динамических стратегиях* решение о наиболее вероятном исходе команды УП принимается в ходе вычислений исходя из информации о предшествующих переходах (истории переходов), собираемой в процессе выполнения программы. В целом, динамические стратегии по сравнению со статическими обеспечивают более высокую точность предсказания. Прежде чем приступить к рассмотрению конкретных динамических механизмов предсказания переходов, остановимся на некоторых положениях, общих для всех динамических подходов.

Идея динамического предсказания переходов предполагает накопление информации о том, как завершались команды УП при их предшествующем исполнении. История переходов фиксируется в форме таблицы, каждый элемент которой состоит из m битов. Нужный элемент таблицы выбирается с помощью k-разрядной двоичной комбинации — шаблона (pattern). Этим объясняется общепринятое название таблицы предыстории переходов — *таблица истории для шаблонов* (PHT, Pattern History Table).

При описании динамических схем предсказания переходов их часто расценивают как один из видов автоматов Мура, при этом содержимое элементов PHT трактуется как информация, отображающая текущее состояние автомата. В работе [167] рассмотрены различные варианты подобных автоматов, однако реально интерес представляют лишь три варианта, которые условно обозначим A1, A2 и A3.

Автомат A1 имеет только два состояния, поэтому каждый элемент РНТ состоит из одного бита ($m = 1$), значение которого отражает исход последнего выполнения команды условного перехода. Диаграмма состояний автомата A1 приведена на рис. 9.6.

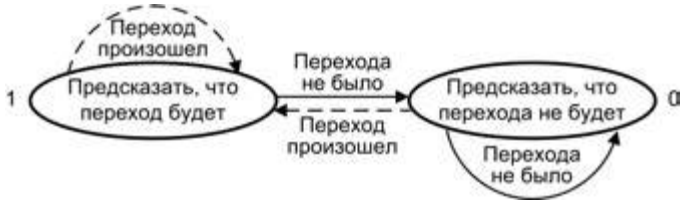


Рис. 9.6. Диаграмма состояний автомата A1

Если команда завершилась переходом, то в соответствующий элемент РНТ заносится единица, иначе — ноль. Очередное предсказание совпадает с итогом предыдущего выполнения команды. После исполнения очередной команды содержимое элемента корректируется в соответствии с ее итогом.

Два других автомата предполагают большее число состояний, поэтому в них используются РНТ с многоразрядными элементами. Чаще всего ограничиваются двумя разрядами ($m = 2$) и, соответственно, автоматами с четырьмя состояниями.

В двухразрядном автомате A2 элементы РНТ отражают исходы двух последних выполненных команд условного перехода и заполняются по схеме регистра сдвига. После исполнения очередной команды УП содержимое выделенного этой команде элемента РНТ сдвигается влево на один разряд, а в освободившуюся позицию заносится единица (если переход был) или ноль (если перехода не было). При наличии в элементе РНТ хотя бы одной единицы для очередного выполнения команды делается предсказание, что переход будет. При нулевом значении элемента РНТ считается, что перехода не будет. Диаграмма состояний для такого автомата показана на рис. 9.7.

Рис. 9.7. Диаграмма состояний двухразрядного автомата A2

Двухразрядная схема автомата A2 используется относительно редко. Большую известность получила трехразрядная модель. Она, в частности, реализована в процессоре HP PA 8000.

Элементы PHT в автомате A3 можно рассматривать как реверсивные счетчики, работающие в режиме с насыщением. При поступлении на конвейер команды условного перехода происходит обращение к соответствующему счетчику PHT, и в зависимости от текущего состояния счетчика делается прогноз, определяющий дальнейший порядок извлечения команд программы. После прохождения ступени исполнения, когда фактический исход команды становится ясным, содержимое счетчика увеличивается на единицу (если команда завершилась переходом) или уменьшается на единицу (если перехода не было). Счетчики работают в режиме насыщения. Это означает, что добавление единиц сверх максимального числа в счетчике, а также вычитание единиц при нулевом содержимом счетчика уже не производятся.

Интуитивное представление о том, что с увеличением глубины предыстории (увеличением m) точность предсказания должна возрастать, на практике не подтверждается. Результаты исследований, приведенные в [141], показали незначительную разницу в точности предсказания при $m = 3$ и $m = 2$, а при $m > 3$ точность предсказания начинает снижаться. Как следствие, в большинстве известных процессоров используются PHT с двухразрядными элементами ($m = 2$).

Логика предсказания переходов применительно к двухразрядным счетчикам известна как *алгоритм Смита* [138]. Алгоритм предполагает четыре состояния счетчика:

- 00 — перехода не будет (сильное предсказание);
- 01 — перехода не будет (слабое предсказание);
- 10 — переход будет (слабое предсказание);
- 11 — переход будет (сильное предсказание).

Таким образом, основанием для предсказания служит состояние старшего разряда счетчика. Если он содержит единицу, то принимается, что переход произойдет, в противном случае предполагается, что перехода не будет.

Диаграмма состояний двухразрядного автомата A3 приведена на рис. 9.8.

Рис. 9.8. Диаграмма состояний двухразрядного автомата A3

Поскольку вариант PHT со счетчиками получил наиболее широкое распространение, в дальнейшем будем ссылаться именно на него.

После определения способов учета истории переходов и логики предсказания необходимо остановиться на особенностях использования таблицы, в частности на том, какая информация выступает в качестве шаблона (номера входа) для доступа к PHT и какого рода история фиксируется в элементах таблицы. Именно различия в способах использования PHT определяют ту или иную стратегию предсказания.

В качестве шаблонов для доступа к PHT могут быть взяты:

- адрес команды условного перехода;
- содержимое регистра глобальной истории;
- содержимое регистра локальной истории;
- комбинация предшествующих вариантов.

Схема, где для доступа к PHT выбран адрес команды условного перехода (содержимое счетчика команд), позволяет учитывать поведение каждой конкретной команды УП. При многократном выполнении большинства команд условного перехода наблюдается повторяемость исхода: переход либо, как правило, происходит, либо, как правило, не происходит (имеет место *бимодальное распределение исходов*). Каждой команде условного перехода в PHT соответствует свой счетчик (рис. 9.9). Когда команда завершается переходом, содержимое счетчика увеличивается на единицу, а в противном случае — уменьшается на единицу (с соблюдением логики счета с насыщением).

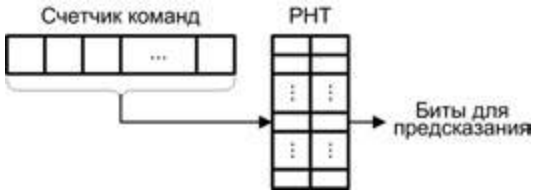


Рис. 9.9. Доступ к PHT, исходя из адреса команды перехода

Схема обеспечивает достаточно высокий процент правильных предсказаний для тех команд УП, которые в ходе вычислений выполняются многократно, например предназначены для управления циклом.

В реальных схемах предсказания переходов размер таблицы PHT ограничен. Типичное количество элементов PHT (элементарных счетчиков) в разных процессорах варьируется от 256 до 4096. Для выбора определенного входа в PHT (нужного счетчика) применяется k-разрядный шаблон, где k определяется размером массива. Для упомянутых выше размеров PHT значение k лежит в диапазоне от 8 до 12. Если обращение к PHT определяется счетчиком команд, разрядность которого обычно больше, чем k, в качестве шаблона выступают k младших битов СК (рис. 9.10). Как

следствие, команды условного перехода, адреса которых в младших *k* разрядах совпадают, будут обращаться к одному и тому же элементу PHT, и история выполнения одной команды будет накладываться на историю выполнения других, что, естественно, будет влиять на точность предсказания. Ситуация известна как *эффект наложения* (aliasing).

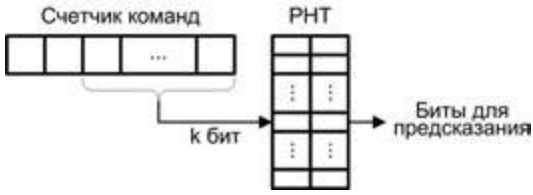


Рис. 9.10. Доступ к PHT, исходя из k младших разрядов адреса команды перехода

В зависимости от типа программы и других факторов наложение может приводить к повышению точности предсказания, ее ухудшению либо вообще не сказываться на точности предсказания. Соответственно, эффекты наложения классифицируют как *конструктивный*, *деструктивный* и *нейтральный*. Многочисленные исследования показали, что чаще всего наложение положительно сказывается на точности предсказаний. Связано это с тем, что исход отдельной команды условного перехода в значительной мере зависит от поведения предшествующих ей команд УП, связь которых с рассматриваемой командой не столь очевидна. Иными словами, между исходами команд условного перехода в программе существует некоторая взаимосвязь, учет которой дает возможность повысить процент правильных предсказаний. Это наблюдение стало побудительной причиной для создания схем, где улучшение точности предсказания во многом определяется учетом эффекта наложения. Речь идет о схемах с регистрами глобальной и локальной истории.

Регистр глобальной истории (GHR, Global History Register) представляет собой *l*-разрядный сдвиговый регистр (рис. 9.11). После выполнения очередной команды условного перехода содержимое регистра сдвигается на один разряд влево, а в освободившуюся позицию заносится единица (если исходом команды был переход) или ноль (если перехода не было). Следовательно, кодовая комбинация в GHR отражает историю выполнения последних по времени *l* команд условного перехода.

В качестве шаблона для доступа к PHT выступают *k* младших разрядов GHR (чаще всего $l = k$). Каждой *k*-разрядной комбинации исходов последовательно выполнявшихся команд УП в PHT соответствует свой элемент (счетчик). Таким образом, выбор счетчика в PHT определяется тем, какая комбинация исходов имела место в *k* предшествовавших командах перехода. Содержимое счетчика используется для предсказания исхода текущей команды перехода и впоследствии модифицируется по результату фактического выполнения команды.

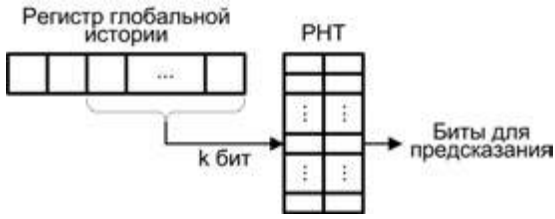


Рис. 9.11. Доступ к PHT, исходя из содержимого регистра глобальной истории

Регистр локальной истории (LHR, Local History Register) по логике работы аналогичен регистру глобальной истории, но предназначен для фиксации последовательных исходов одной отдельной команды УП. Таким образом, каждой команде УП выделен свой регистр локальной истории, и в схемах предсказания с LHR присутствует так называемая *таблица локальной истории*, представляющая собой массив регистров локальной истории (рис. 9.12).

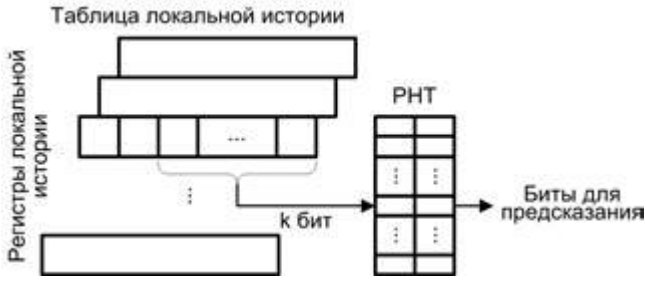


Рис. 9.12. Доступ к PHT, исходя из регистров локальной истории

С учетом положительного влияния эффекта наложения на точность предсказаний в некоторых схемах этот эффект еще более усугубляется. Так, в ряде динамических методов предсказания шаблон для доступа к PHT формируется путем объединения адреса команды перехода и содержимого GHR (либо LHR), при этом используется одна из двух операций: конкатенация (сцепление) или сложение по модулю 2 («исключающее ИЛИ»).

При конкатенации k-разрядный шаблон для обращения к PHT образуется посредством взятия q младших битов из одного источника, к которым пристыковываются $k–q$ младших разрядов, взятых из второго источника (рис. 9.13).

Эффективность предсказания на основе подобного шаблона зависит от соотношения количества разрядов (q и $k–q$), выбранных от каждого из двух источников. Здесь многое определяется и характером программы.

Вариант со сложением по модулю 2 предполагает побитовое применение операции «Исключающее ИЛИ» к k младшим разрядам обоих источников (рис. 9.14).

Результаты экспериментов, приведенные в [141], позволяют дать сравнительную оценку рассмотренных схем динамического предсказания.

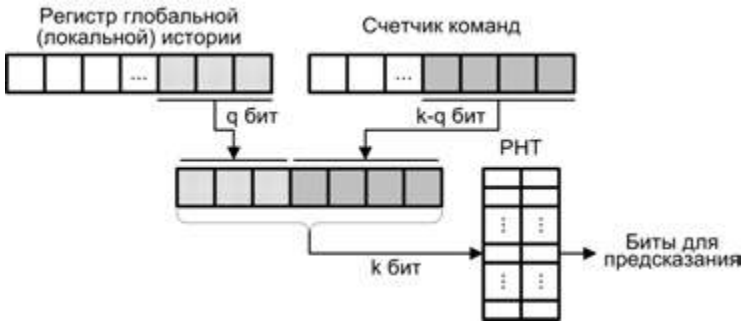


Рис. 9.13. Формирование шаблона для доступа к PHT путем конкатенации

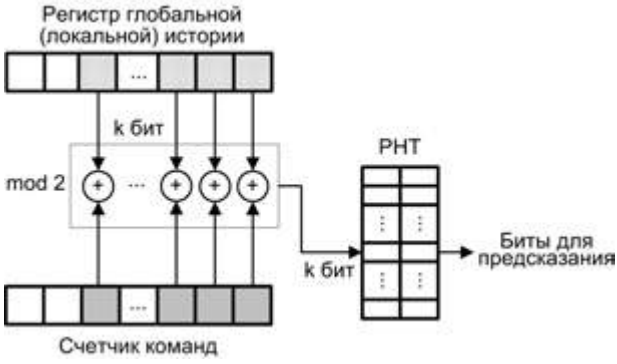


Рис. 9.14. Формирование шаблона для доступа к PHT путем сложения по модулю 2

Первый вывод, вытекающий из представленных данных, — с увеличением размера PHT точность предсказания возрастает. Среди четырех рассмотренных схем наилучшую точность обеспечивает схема со сложением по модулю два. Схема со счетчиком команд относится к наименее эффективным. При малых объемах PHT вариант с конкатенацией дает наихудшие результаты, но при больших PHT эта схема превосходит модель со счетчиком команд.

После изложения основных идей динамического предсказания переходов следует рассмотреть их модификации, нашедшие место в реальных процессорах. В целом весь спектр динамических стратегий можно свести к следующим группам:

- одноуровневые или бимодальные;
- двухуровневые или коррелированные;
- гибридные;
- асимметричные.

Одноуровневые схемы предсказания переходов. В многочисленных исследованиях, проводившихся на самых разнообразных программах, была отмечена интересная закономерность. В поведении многих команд условного перехода явно прослеживается тенденция повторяемости исхода: одни команды программы, как правило,

завершаются переходом, в то время как другие — остаются без оного, то есть имеет место бимодальное распределение исходов. Идея одноуровневых схем предсказания, известных также под вторым названием — *бимодальные схемы* [119], сводится к отделению команд, имеющих склонность завершаться переходом, от команд, при выполнении которых переход обычно не происходит. Такая дифференциация позволяет для каждой команды выбрать наиболее подходящее предсказание. Для реализации идеи в составе схемы предсказания достаточно иметь лишь одну таблицу, каждый элемент которой отображает историю исходов одной команды УП. Для обращения к элементу, ассоциированному с определенной командой УП, используется адрес этой команды (или его младшие биты). Таким образом, одноуровневые схемы предсказания переходов можно определить как схемы, содержащие один уровень таблиц истории переходов (обычно единственную таблицу), адресуемых с помощью адреса команды условного перехода.

В первом из возможных вариантов одноуровневых схем строится сравнительно небольшая таблица, куда заносятся адреса *n* последних команд УП, при выполнении которых переход не случился. В сущности, это ранее упоминавшийся *буфер адресов перехода* (BTB), но с противоположным правилом занесения информации (фиксируются адреса команд, завершившихся без перехода). Таблица обычно реализуется на базе ассоциативной кэш-памяти. При поступлении на конвейер очередной команды УП ее адрес сравнивается с адресами, хранящимися в таблице. При совпадении делается предположение о том, что перехода не будет, в противном случае предсказывается переход. С этих позиций, для каждой новой команды УП по умолчанию принимается, что переход произойдет. После того как фактический исход становится очевидным, содержимое таблицы корректируется. Если команда завершилась переходом, соответствующая запись из таблицы удаляется. Если же перехода не было, то адрес команды заносится в таблицу при условии, что до этого он там отсутствовал. При заполнении таблицы опираются на алгоритмы LRU или FIFO. По точности предсказания стратегия превосходит большинство стратегий статического предсказания.

Вторая схема ориентирована на то, что команды программы извлекаются из кэш-памяти команд. Каждая ячейка кэш-памяти содержит дополнительный разряд, который используется только применительно к командам условного перехода. Состояние разряда отражает исход предыдущего выполнения команды (1 — переход был, 0 — перехода не было). Новое предсказание совпадает с результатом предшествующего выполнения данной команды. При занесении такой команды в кэш-память рассматриваемый разряд устанавливается в единицу. Это напоминает стратегию «при первом выполнении переход обязательно происходит» с той лишь разницей, что первое предсказание, хотя и носит статический характер, происходит в ходе заполнения кэш-памяти, то есть динамически. После выполнения команды состояние дополнительного бита корректируется: если переход имел место, в него заносится единица, а в противном случае — ноль. Точность предсказания по данным, приведенным в [141], в среднем составляет 90%. Главный недостаток схемы заключается в дополнительных затратах времени на обновление состояния контрольного разряда в кэш-памяти команд.

В третьей схеме (рис. 9.15) PHT состоит из одноразрядных элементов и носит название *таблицы истории переходов* (BHT, Branch History Table).

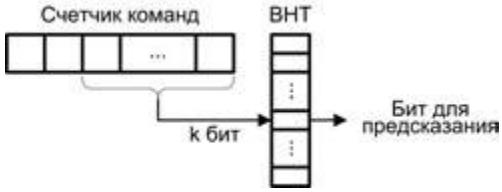


Рис. 9.15. Однобитовая бимодальная схема предсказания

Состояние элемента BHT определяет, произошел ли переход в ходе последнего выполнения команды условного перехода (1) или нет (0). Каждой команде УП в таблице соответствует свой элемент, для обращения к которому используются *k* младших разрядов адреса команды. Предсказание совпадает с исходом предыдущего выполнения команды. Если команда условного перехода участвует в организации цикла, то стратегия всегда приводит к неправильному предсказанию перехода в первой и последней итерациях цикла.

Схема была реализована в процессорах Alpha 21064 и AMD K5. Согласно результатам большинства исследований, средняя точность успешного прогноза с помощью однобитовой бимодальной схемы не превышает 78%. В то же время в работе [141] получено значение 90,4%.

Точность предсказания перехода существенно повышается с увеличением разрядности элементов BHT. В четвертой из рассматриваемых одноуровневых схем каждый элемент BHT состоит из двух битов, выполняющих функцию двухразрядного счетчика, работающего в режиме с насыщением. Иными словами, реализуется алгоритм Смита. Каждый счетчик отображает историю выполнения одной команды УП, то есть адресуется младшими разрядами счетчика команд (рис. 9.16).

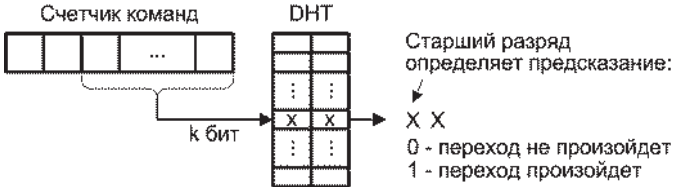


Рис. 9.16. Одноуровневая схема предсказания с таблицей DHT

Для обозначения таблицы истории переходов в данной схеме используют аббревиатуру DHT (Decode History Table). Слово «decode» (декодирование) в названии отражает особенность работы с таблицей. Если к обычной BHT обращение происходит при выборке любой команды, вне зависимости от того, на самом ли деле она команда условного перехода, поиск в DHT начинается только после декодирования команды, то есть когда выяснилось, что данная команда является командой условного перехода. Реализуется DHT на базе обычного ЗУ с произвольным доступом.

Последняя из обсуждаемых одноуровневых схем предсказания известна как *схема Смита* или *бимодальный предиктор*. Отличие от предыдущего варианта выражается лишь в способе реализации BHT (в схеме Смита сохранено именно это название таблицы). Таблица истории переходов организуется на базе кэш-памяти с ассоциативным отображением (рис. 9.17).

Рис. 9.17. Схема Смита

В качестве ассоциативного признака (тега) при поиске нужного счетчика выступает адрес команды условного перехода. Такой подход позволяет ускорить поиск нужного счетчика и устраняет эффект наложения, но связан со значительными аппаратными затратами. По результатам моделирования бимодальной схемы с двухразрядными счетчиками точность предсказания можно оценить как 92,6%, хотя на некоторых программах точность предсказания составила 53,9%. Несмотря на это, бимодальная схема с двухразрядными счетчиками довольно распространена. На ее основе построены схемы предсказания переходов в процессорах Alpha 21164, R10000, PowerPC 620, UltraSPARC и др.

В заключение отметим, что во многих одноуровневых решениях таблица истории переходов (DHT или BHT) совмещена с буфером адреса перехода (BTB), что позволяет сэкономить на вычислении исполнительных адресов точек перехода.

Двухуровневые схемы предсказания переходов. Одноуровневые схемы предсказания ориентированы на те команды УП, очередной исход которых существенно зависит от их собственных предыдущих исходов. В то же время для многих команд программы наблюдается сильная зависимость не от собственных исходов, а от результатов выполнения других предшествующих им команд УП. Это обстоятельство призваны учесть *двухуровневые адаптивные схемы* предсказания переходов, впервые предложенные в [167]. Такие схемы часто называют *коррелированными*, подчеркивая тот факт, что они отражают взаимозависимость команд условного перехода.

В коррелированных схемах предсказания переходов выделяются два уровня таблиц. В роли таблицы первого уровня может выступать регистр глобальной истории (GHR), и тогда двухуровневую схему предсказания называют *глобальной*. В качестве таблицы первого уровня может также быть взят массив регистров локальной

истории (LHR), где отдельный регистр отражает последовательность последних исходов одной команды УП. Такая схема предсказания носит название *локальной*. Каждый элемент таблицы второго уровня служит для хранения истории переходов отдельной команды УП. Таблица второго уровня обычно состоит из двухразрядных счетчиков и организована в виде матрицы (рис. 9.18). Содержимое счетчика команд (адрес команды УП) определяет один из регистров в таблице первого и одну строку в таблице второго уровня. В свою очередь, кодовая комбинация (шаблон), хранящаяся в выбранном регистре таблицы первого уровня, определяет нужный счетчик в указанном ряду таблицы второго уровня. Выбранный счетчик используется для формирования предсказания. После выполнения команды содержимое регистра и счетчика обновляется.

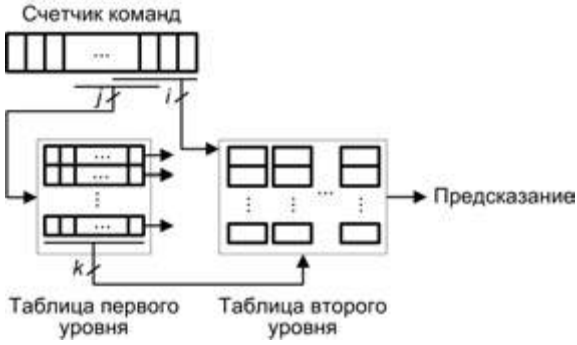


Рис. 9.18. Общая структура двухуровневой схемы предсказания переходов

Из описания логики двухуровневого предсказания следует, что выбор нужного счетчика обусловлен двумя источниками — адресом команды, для которой делается предсказание, и шаблоном, отражающим историю предшествующих переходов. Того же эффекта можно добиться при помощи схемы двухразрядного бимодального предиктора, если для доступа к BHT использовать шаблон, сформированный путем сложения по модулю 2. Такая схема известна под названием *gshare*.

Гибридные схемы предсказания переходов. Для всех ранее рассмотренных стратегий характерна сильная зависимость точности предсказания от особенностей программ, в рамках которых эти стратегии реализуются. Одна и та же самая схема, прекрасно проявляя себя с одними программными продуктами, с другими может давать совершенно неудовлетворительные результаты. Кроме того, необходимо учитывать еще один фактор. Прежде уже отмечалось, что точность предсказания повышается с увеличением глубины предыстории переходов, но происходит это лишь после накопления соответствующей информации, на что требуется определенное время. Период накопления предыстории принято называть *временем «разогрева»*. Пока идет «разогрев», точность предсказания весьма низка. Иными словами, ни одна из элементарных стратегий предсказания переходов не является универсальной — со всех сторон лучшей в любых ситуациях. *Гибридные* или *соревновательные схемы* объединяют в себе несколько различных механизмов предсказания — элементарных предикторов. Идея состоит в том, чтобы в каждой конкретной ситуации

задействовать тот элементарный предиктор, от которого в данном случае можно ожидать наибольшей точности предсказания.

Гибридная схема предсказания переходов, предложенная Макфарлингом [119], содержит два элементарных предиктора, отличающихся по своим характеристикам (размером таблиц предыстории и временем «разогрева») и работающих независимо друг от друга. Выбор предиктора, наиболее подходящего в данной ситуации, обеспечивается селектором, представляющим собой таблицу двухразрядных счетчиков, которые часто называют *счетчиками выбора предиктора* (рис. 9.19).

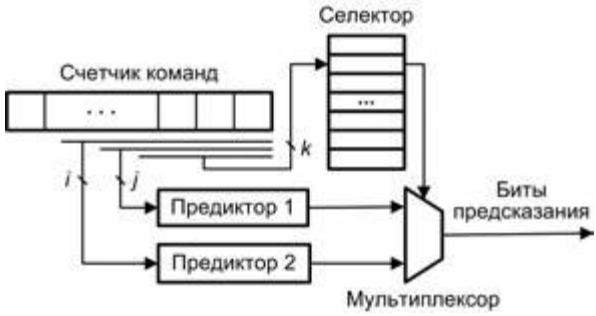


Рис. 9.19. Гибридный предиктор Макфарлинга

Адресация конкретного счетчика в таблице (индексирование) осуществляется k младшими разрядами адреса команды условного перехода, для которой осуществляется предсказание. Обновление таблиц истории в каждом из предикторов производится обычным образом, как это происходит при их автономном использовании. В свою очередь, изменение состояния счетчиков селектора выполняется по следующим правилам. Если оба предиктора одновременно дали одинаковое предсказание (верное или неверное), содержимое счетчика не изменяется. При правильном предсказании от первого предиктора и неверном от второго содержимое счетчика увеличивается, а в противоположном случае — уменьшается на единицу. Выбор предиктора, на основании которого делается результирующая оценка, реализуется с помощью мультиплексора, управляемого старшим разрядом соответствующего счетчика селектора.

В работе [82] идея гибридного механизма была обобщена на случай n предикторов. Общая структура такой схемы предсказания переходов показана на рис. 9.20. При выполнении команды УП предсказания формируются одновременно всеми предикторами, однако реальные действия осуществляются на основании только одного из них.

Выбор подходящего предиктора обеспечивает механизм селекции (рис. 9.21). В схеме имеется буфер предыстории переходов (BTB), в котором каждая запись дополнена n двухразрядными счетчиками выбора предиктора, по числу используемых элементарных предикторов. Счетчики позволяют отследить самый предпочтительный элементарный предиктор для каждой команды УП, представленной в BTB.

Рис. 9.20. Общая схема гибридного предиктора

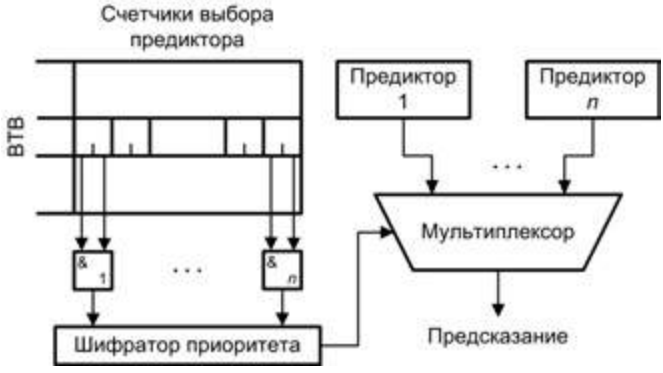


Рис. 9.21. Механизм выбора предиктора

При записи в BTB нового элемента во все ассоциированные с ним счетчики заносится число 3. Для каждой команды условного перехода предсказание генерируется всеми n предикторами, но во внимание принимаются только те из них, для которых соответствующий счетчик выбора предиктора содержит число 3. Если это число встретилось более чем в одном счетчике, выбор единственного предиктора, на основании которого и делается окончательное предсказание, обеспечивает шифратор приоритета.

После выполнения команды условного перехода содержимое соответствующих ей счетчиков выбора обновляется, при этом действует следующий алгоритм. Если среди предикторов, счетчики которых содержали число 3, хотя бы один дал верное предсказание, то содержимое всех счетчиков, связанных с неверно сработавшими предикторами, уменьшается на единицу. В противном случае содержимое всех счетчиков, связанных с предикторами, прогноз которых подтвердился, увеличивается на единицу. Такая политика гарантирует, что по крайней мере в одном из счетчиков будет число 3. Еще одно преимущество рассматриваемой схемы выбора состоит в том, что она позволяет, например, отличить предиктор-«оракул», давший

правильное предсказание последние пять раз, от предиктора, верно определившего исход последние четыре раза. Стандартные счетчики с насыщением такую дифференциацию не обеспечивают.

По имеющимся оценкам, точность предсказания переходов с помощью гибридных стратегий в среднем составляет 97,13%, что существенно выше по сравнению с прочими вариантами.

Асимметричная схема предсказания переходов. Асимметричная схема сочетает в себе черты гибридных и коррелированных схем предсказания. От гибридных схем она переняла одновременное срабатывание нескольких различных элементарных предикторов. В асимметричной схеме таких предикторов три, и каждый из них использует собственную таблицу РНТ. Для доступа к таблицам, аналогично коррелированным схемам, используется как адрес команды условного перехода, так и содержимое регистра глобальной истории (рис. 9.22).

Рис. 9.22. Структура асимметричной схемы предсказания переходов

Шаблон для обращения к каждой из трех РНТ формируется по-разному (применены различные функции хэширования[1]). При выполнении команды условного перехода каждый из трех предикторов выдвигает свое предположение, но окончательное решение принимается по мажоритарной схеме. После завершения команды условного перехода содержимое всех трех таблиц обновляется.

Правильный подбор алгоритмов хэширования позволяет практически исключить влияние на точность предсказания эффекта наложения. Средняя точность предсказания с помощью асимметричной схемы, полученная в экспериментах, составила 72,6%.

Суперконвейерные процессоры

Эффективность конвейера находится в прямой зависимости от того, с какой частотой на его вход подаются объекты обработки. *Суперконвейеризация* (термин

[1] Хеширование — преобразование входного массива данных произвольной длины в выходную битовую строку фиксированной длины таким образом, чтобы изменение входных данных приводило к непредсказуемому изменению выходных данных.

впервые был применен в 1988 году) позволяет улучшить производительность процессора за счет повышения частоты, с которой команды подаются на конвейер и перемещаются по нему. В обычном конвейере эта частота ограничена временем обработки в самой «медленной» ступени конвейера. В суперконвейере возможность увеличения частоты достигается путем выявления «медленных» ступеней и разбиения их на k меньших ступеней таким образом, чтобы время обработки в каждой из них не превышало аналогичного показателя для остальных ступеней конвейера.

Концепцию суперконвейеризации проиллюстрируем на примере ранее рассмотренного конвейера. На рис. 9.23 приведен стандартный конвейер из семи ступеней. Принято, что время выполнения действий на ступенях ВК, ВО и ЗР, связанных с обращением к памяти, вдвое больше, чем в прочих ступенях (длина стрелок на рисунке характеризует время выполнения операции на каждой из ступеней). Серые прямоугольники, расположенные между ступенями, представляют буферные регистры. Частоту тактирования процессора определяют «медленные» ступени (ВК, ВО и ЗР). Повысить ее можно, разделив каждую из этих ступеней на две подступени и расположив между последними буферные регистры (рис. 9.24).

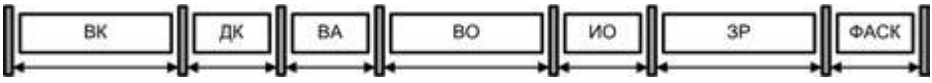


Рис. 9.23. Стандартный конвейер, где быстродействие ограничено «медленными» ступенями

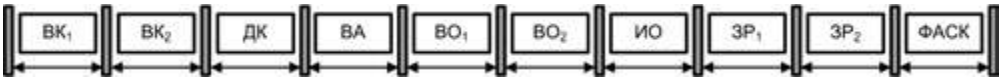


Рис. 9.24. Суперконвейер

Сокращение длительности обработки на ступенях конвейера (повышение частоты продвижения по конвейеру) улучшает производительность процессора. При наличии в конвейере «медленных» ступеней конвейер будет несбалансирован, и на «быстрых» ступенях имеют место потери времени, показанные рис. 9.25, *а* пунктирными стрелками. При выполнении сотен и тысяч команд программы суммарные потери времени могут быть весьма ощутимыми. В суперконвейере ступени сбалансированы, и потери времени будут минимальными (рис. 9.25, *б*).

В сущности, суперконвейеризация сводится к увеличению количества ступеней конвейера путем дробления «медленных» ступеней на несколько простых. Главное требование — возможность реализации операции в каждой ступени конвейера наиболее простыми техническими средствами, а значит, с минимальными затратами времени. Вторым, не менее важным условием является одинаковость задержки во всех ступенях.

Показателем для причисления процессора к суперконвейерным служит число ступеней в конвейере команд. К суперконвейерным относят процессоры, где таких ступеней больше шести. Первым серийным суперконвейерным процессором

считается MIPS R4000, конвейер команд которого включает в себя восемь ступеней. Суперконвейеризация здесь стала следствием разбиения этапов выборки команды и выборки операнда.

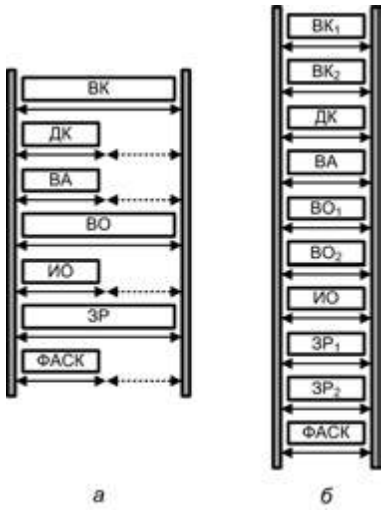


Рис. 9.25. Длительность такта конвейера: *а* — в конвейере с несбалансированными ступенями; *б* — в суперконвейере со сбалансированными ступенями

Наряду с понятием суперконвейеризации применение нашел и другой термин — «гиперконвейеризация», который компания Intel использовала при описании процессора Pentium IV с его конвейером команд из 20 ступеней[1].

К сожалению, выигрыш, достигаемый за счет суперконвейеризации, на практике может оказаться лишь умозрительным. Удлинение конвейера ведет не только к усугублению проблем, характерных для любого конвейера, но и к возникновению дополнительных сложностей. В длинном конвейере возрастает вероятность конфликтов. Дороже обходится ошибка предсказания перехода — приходится очищать большее число ступеней конвейера, на что требуется больше времени. Усложняется логика взаимодействия ступеней конвейера. Отражением этих проблем стало то, что Intel в своих разработках сначала увеличила длину конвейера до 31 ступени (ядро Prescoot), а в последующих моделях (процессор Core 2) сократила число ступеней до 14.

Суперскалярные процессоры

Поскольку возможности по совершенствованию элементной базы уже практически исчерпаны, дальнейшее повышение производительности ВМ лежит в плоскости

[1] В ряде источников понятие «гиперконвейер» связывают не только с количеством ступеней, но также и с тем, что разные ступени одного и того же конвейера могут работать на индивидуальных тактовых частотах

архитектурных решений. Один из наиболее эффективных подходов в этом плане — введение в вычислительный процесс различных уровней параллелизма. Ранее рассмотренный конвейер команд — типичный пример такого подхода. Тем же целям служат и арифметические конвейеры, где конвейеризации подвергается процесс выполнения арифметических операций. Дополнительный уровень параллелизма реализуется в векторных и матричных процессорах, но только при обработке многокомпонентных операндов типа векторов, массивов или строк. Здесь высокое быстродействие достигается за счет одновременной обработки всех компонентов вектора, массива или строки, однако подобные операнды характерны лишь для достаточно узкого круга решаемых задач. Основной объем вычислительной нагрузки обычно приходится на скалярные вычисления, то есть на обработку одиночных операндов. Для подобных вычислений дополнительный параллелизм реализуется значительно сложнее, но тем не менее возможен, и примером могут служить суперскалярные процессоры.

Суперскалярным (этот термин впервые был использован в 1987 году [40]) называется центральный процессор (ЦП), который одновременно выполняет более чем одну скалярную команду. Это достигается за счет включения в состав ЦП нескольких самостоятельных функциональных блоков, каждый из которых отвечает за свой класс операций и может присутствовать в процессоре в нескольких экземплярах. Так, в микропроцессоре Pentium III блоки целочисленной арифметики и операций с плавающей запятой дублированы, а в микропроцессорах Pentium 4 и Athlon — троированы.

Суперскалярность предполагает параллельную работу нескольких функциональных блоков, что возможно лишь при одновременном выполнении нескольких скалярных команд. Последнее условие хорошо сочетается с конвейерной обработкой, при этом желательно, чтобы таких конвейеров было несколько, например два или три. Разумеется, в этом случае ступень выборки команд, общая для всех конвейеров, должна в каждом такте извлекать из памяти сразу несколько команд. За этой ступенью располагается блок диспетчеризации, отвечающий за распределение команд по конвейерам.

На рис. 9.26 приведен пример процессора с двумя конвейерами, один из которых содержит операционное устройство для чисел с фиксированной запятой (ФЗ), а другой — для чисел с плавающей запятой (ПЗ). Сначала поясним назначение показанной на рисунке очереди команд, которая для построения суперскалярного процессора не является принципиально необходимой, но тем не менее присутствует в большинстве современных процессоров. Очередь связана с техническим приемом, известным как *предвыборка команд*. Суть его в том, что при каждой возможности производится считывание команд из памяти, опережающее ход вычислений, и их размещение в быстродействующем *буфере предвыборки*. Буфер организован по принципу очереди (FIFO), а команды в буфер поступают в порядке их выполнения в программе. Высокое быстродействие буфера и наличие в нем значительного количества команд позволяет одновременно загружать все конвейеры процессора.

Рис. 9.26. Суперскалярный процессор с двумя конвейерами

В каждом такте блок диспетчеризации извлекает из очереди и декодирует две команды. Если одна из них обрабатывает числа с ФЗ, а другая — числа с ПЗ, обе команды одновременно подаются на конвейеры (каждая на предназначенный для нее конвейер).

В рассмотренном примере каждый конвейер специализирован под определенный тип данных и содержит в своем составе соответствующее операционное устройство (ОПУ). В реальных процессорах более распространен иной подход, когда имеется комплекс функциональных блоков, специализированных под определенные операции, а все конвейеры вплоть до ступени исполнения операции идентичны (рис. 9.27). В этом случае команды на конвейеры подаются без учета их типа, а диспетчеризация (распределение по соответствующим функциональным блокам) производится после прохождения командами ступеней, предшествующих их исполнению. Подобная форма суперскалярного процессора используется в большинстве современных процессоров.

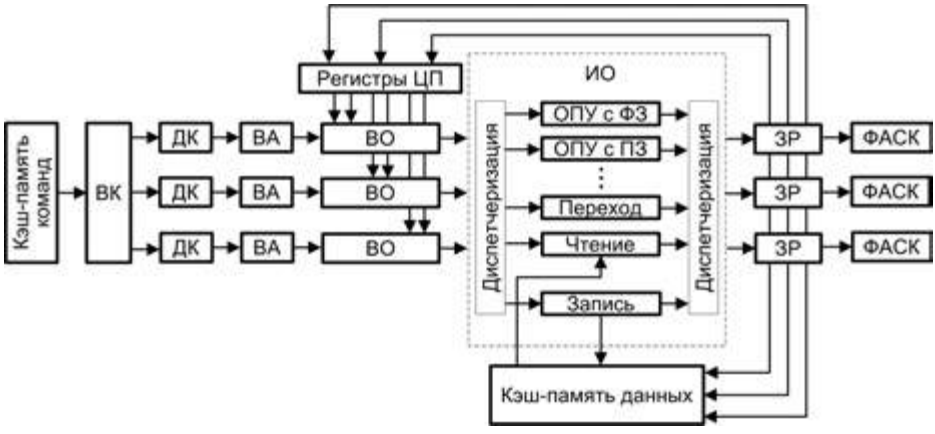


Рис. 9.27. Суперскалярный конвейер со специализированными функциональными блоками

По разным оценкам, применение суперскалярного подхода приводит к повышению производительности ВМ в пределах от 1,8 до 8 раз.

Для сравнения эффективности суперскалярного и суперконвейерного режимов на рис. 9.28 показан процесс выполнения восьми последовательных скалярных команд (без учета этапа ФАСК). Верхняя диаграмма иллюстрирует суперскалярный процессор с двумя конвейерами, а нижняя — один суперконвейер, где каждый этап реализуется двумя ступенями.

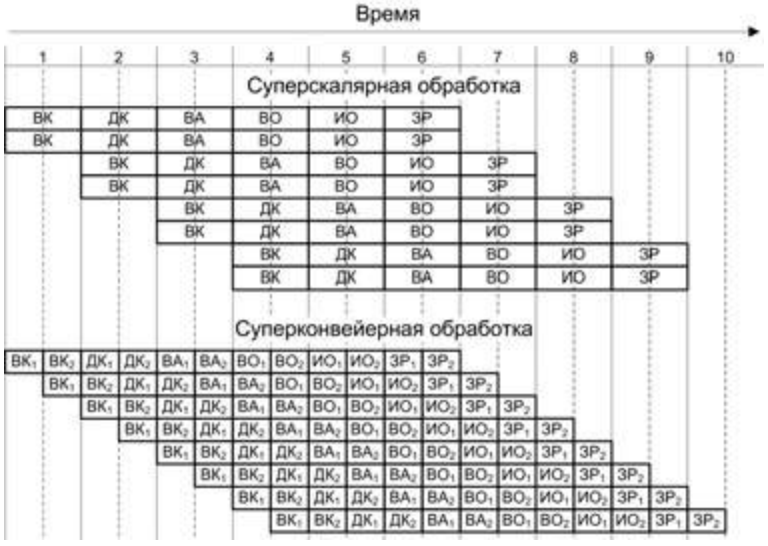


Рис. 9.28. Сравнение суперскалярного и суперконвейерного подходов

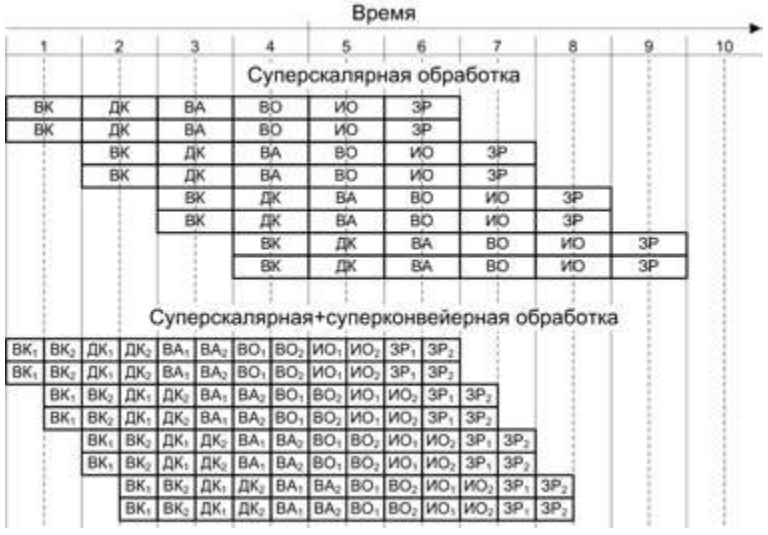


Рис. 9.29. Сравнение эффективности стандартной и совмещенной схем суперскалярных вычислений

В современных процессорах суперскалярность обычно совмещают с элементами суперконвейеризации. Одной из первых этот подход использовала фирма AMD в своих микропроцессорах Athlon и Duron, причем охватывал он не только конвейер команд, но и блок обработки чисел в форме с плавающей запятой. Возможный эффект от подобного совмещения иллюстрирует рис. 9.29.

Особенности реализации суперскалярных процессоров

Поскольку в реальных суперскалярных процессорах множественность функциональных блоков сочетается с множественностью конвейеров команд, таким процессорам присущи все виды зависимостей, характерные для одинарных конвейеров, при этом положение дел усугубляется тем, что конвейеров несколько. В суперскалярных процессорах одновременная работа нескольких конвейеров становится источником дополнительных проблем, связанных с последовательностью поступления команд на исполнение и последовательностью завершения команд.

Первая из упомянутых проблем возникает, когда очередность выдачи команд на исполнение в функциональные блоки отличается от последовательности, предписанной программой. Подобная ситуация известна как *неупорядоченная выдача команд* (out-of-order issue). Термин *упорядоченная выдача команд* (in-order issue) применяют, когда команды покидают ступени, предшествующие ступени исполнения, в порядке, предопределенном программой. В обоих случаях завершение команд обычно неупорядочено (*неупорядоченное завершение команд* — out-of-order completion), и это является второй проблемой. Например, в последовательности

MUL R1, R2, R3{R1 ← R2 × R3}

ADD R4, R5, R6 {R4 ← R5 + R6},

даже если команда умножения MUL поступит в функциональный блок до команды сложения ADD, умножение может потребовать много циклов, из-за чего команда MUL будет завершена позже, чем ADD. В современных микропроцессорах выполнение каждой команды разбивается на простейшие микрооперации, которые далее выполняются суперскалярным процессорным ядром в удобном порядке.

В суперскалярных процессорах, с их множественными конвейерами и неупорядоченными выдачами/завершениями, взаимозависимость команд представляет серьезную проблему. Кроме того, существует еще один фактор, характерный только для суперскалярных процессоров — конфликт по функциональному блоку, когда на него претендуют несколько команд, поступивших из разных конвейеров.

Пусть имеется последовательность команд

ADD R1, R2, R3 {R1 ← R2 + R3}

SUB R4, R5, R6 {R4 ← R5 – R6}.

Зависимости между командами здесь нет, однако если в ЦП имеется только одно АЛУ, одновременное выполнение требуемых операций невозможно.

Стратегии выдачи и завершения команд

В режиме параллельного выполнения нескольких команд процессор должен определить, в какой очередности ему следует:

- выбирать команды из памяти;
- выполнять эти команды;
- позволять командам изменять содержимое регистров и ячеек памяти.

Для достижения максимальной загрузки всех ступеней своих конвейеров суперскалярный процессор должен варьировать порядок исполнения и завершения команд программы, но так, чтобы получаемый результат был идентичен результату при выполнении команд в порядке, определенном программой. Значит, процессор обязан учитывать все виды зависимостей и конфликтов.

В самом общем виде стратегии выдачи и завершения команд можно сгруппировать в такие категории:

- упорядоченная выдача и упорядоченное завершение;
- упорядоченная выдача и неупорядоченное завершение;
- неупорядоченная выдача и неупорядоченное завершение.

Проанализируем каждый из этих вариантов на примере суперскалярного процессора с двумя конвейерами [143]. Процессор способен одновременно выбирать и декодировать две команды, причем передача обеих команд на декодирование должна также производиться одновременно. В состав процессора входят три отдельных функциональных блока (ФБ) и два устройства, обеспечивающие запись результата. В рассматриваемом примере предполагается существование следующих ограничений на выполнение программного кода из шести команд (I1–I6):

- I1 требует для своего выполнения двух циклов процессора;
- I3 и I4 имеют конфликт за обладание одним и тем же ФБ;
- I5 зависит от значения, вычисляемого командой I4;
- I5 и I6 конфликтуют за обладание одним и тем же ФБ.

Упорядоченная выдача и упорядоченное завершение. Наиболее простым в реализации вариантом является выдача декодированных команд на исполнение в том порядке, в котором они должны выполняться по программе (упорядоченная выдача), с сохранением той же последовательности записи результатов (упорядоченное завершение). Хотя такая стратегия и применялась в первых процессорах типа Pentium, сейчас она практически не встречается. Тем не менее ее целесообразно рассмотреть в качестве точки отсчета при сравнении различных стратегий выдачи и завершения. Согласно данному принципу, все что затрудняет завершение команды в одном конвейере, останавливает и другой конвейер, так как команды должны покидать конвейеры в порядке поступления. Пример использования подобной стратегии показан на рис. 9.30, *а*.

Цикл	ДК		ИО			ЗР	
1	I1	I2					
2	I3	I4	I1	I2			
3	I3	I4	I1				
4		I4			I3	I1	I2
5	I5	I6			I4		
6		I6		I5		I3	I4
7				I6			
8						I5	I6

а

Цикл	ДК		ИО			ЗР	
1	I1	I2					
2	I3	I4	I1	I2			
3		I4	I1		I3	I2	
4	I5	I6			I4	I1	I3
5		I6		I5		I4	
6				I6		I5	
7						I6	

б

Цикл	ДК		Окно	ИО			ЗР	
1	I1	I2						
2	I3	I4	I1, I2	I1	I2			
3	I5	I6	I3, I4	I1		I3	I2	
4			I4, I5, I6		I6	I4	I1	I3
5			I5		I5		I4	I6
6							I5	

в

Рис. 9.30. Пример выполнения шести команд в соответствии со стратегиями: *а* — упорядоченной выдачи и упорядоченного завершения; *б* — упорядоченной выдачи и неупорядоченного завершения; *в* — неупорядоченной выдачи и неупорядоченного завершения

Здесь производятся одновременная выборка и декодирование двух команд. Чтобы принять очередные команды, процессор должен ожидать, пока освободятся обе части ступени декодирования. Для упорядочивания завершения выдача команд приостанавливается, если возникает конфликт за общий функциональный блок или если функциональному блоку для формирования результата требуется более чем один такт процессора.

В рассматриваемом примере время задержки от декодирования первой команды до записи последнего результата составляет 8 тактов.

Упорядоченная выдача и неупорядоченное завершение. Стратегии с неупорядоченным завершением дают возможность одному из конвейеров продолжать работать в случае «затора» в другом, при этом команды, стоящие в программе «позже», могут быть фактически выполнены раньше предыдущих, «застрявших» в другом конвейере. Естественно, процессор должен гарантировать, что результаты не будут записаны в память, а регистры не будут модифицироваться в неправильной последовательности, поскольку при этом могут получиться ошибочные результаты.

Стратегию с упорядоченной выдачей и неупорядоченным завершением иллюстрирует рис. 9.30, *б*. При заданных условиях допускается, что команда I2 может быть завершена еще до окончания исполнения команды I1. Это позволяет команде I3 завершиться на один такт раньше, вследствие чего результаты выполнения команд I1 и I3 записываются в одном и том же такте. При неупорядоченной выдаче в любой момент времени в стадии исполнения может находиться любое число команд, а степень параллелизма ограничена только числом функциональных блоков. По сравнению с предыдущей стратегией, возможность неупорядоченного завершения команд привела к сокращению времени выполнения шести команд на один цикл процессора.

Неупорядоченная выдача и неупорядоченное завершение. Неупорядоченная выдача развивает предыдущую концепцию, разрешая процессору нарушать предписанный программой порядок выдачи команд на исполнение. Чтобы обеспечить неупорядоченную выдачу команд, в конвейере необходимо максимально развязать ступени

декодирования и исполнения. Это обеспечивается с помощью буферной памяти, называемой *окном команд*. Каждая декодированная команда сначала помещается в окно команд. Процессор может продолжать выборку и декодирование новых команд вплоть до полного заполнения буфера. Выдача команд из буфера на исполнение определяется не последовательностью их поступления, а мерой готовности. Иными словами, любая команда, для которой уже известны значения всех операндов, при условии что функциональный блок, требуемый для ее исполнения, свободен, немедленно выдается из буфера на исполнение.

Стратегию иллюстрирует рис. 9.30, *в*. В каждом цикле процессора две команды из ступени декодирования пересылаются в окно команд (с учетом ограничения на размер буфера). Выдача команд из буфера производится по мере их готовности. Так, в рассматриваемом примере возможна выдача команды I6 до выдачи команды I5 (напомним, что I5 зависит от I4, а I6 — нет). Таким образом, сберегается один такт, как в ступени исполнения операции (ИО), так и в ступени записи результата (ЗР), и сквозная экономия по сравнению с рис. 9.30, *б* составляет один цикл процессора. На рисунке изображено окно команд, но оно не является дополнительной ступенью конвейера.

Стратегии неупорядоченной выдачи и неупорядоченного завершения также свойственны ранее рассмотренные ограничения. Команда не может быть выдана, если она приводит к зависимости или конфликту. Разница заключается в том, что к выдаче готово больше команд, и это позволяет уменьшить вероятность приостановки конвейера.

Аппаратная поддержка суперскалярных операций

Из предыдущих рассуждений следует, что неупорядоченные выдача и завершение команд — это дополнительный потенциал повышения производительности суперскалярного процессора, для реализации которого, вместе с тем, необходимо решить две проблемы:

- устранить зависимость команд по данным (речь идет о зависимостях типа ЧПЗ и ЗПЗ), то есть исключить использование в качестве операнда «устаревшего» значения регистра и не допускать, чтобы очередная команда программы из-за нарушения последовательности выполнения команд занесла свой результат в регистр еще до того, как это сделала предшествующая команда;
- сохранить такой порядок выполнения команд, чтобы общий итог вычислений остался идентичным результату, получаемому при строгом соблюдении программной последовательности.

Несмотря на то что обе задачи в принципе могут быть решены чисто программными средствами еще на этапе компиляции программы, в реальных суперскалярных процессорах для этих целей имеются соответствующие аппаратные средства. Каждая из перечисленных проблем решается своими методами и своими аппаратными средствами, хотя эти методы и средства часто взаимосвязаны. Для устранения зависимости по данным используется прием, известный как *переименование регистров*. Способ решения второй проблемы обобщенно называют *переупорядочиванием команд* или *откладыванием исполнения команд*.

Переименование регистров

Когда команды выдаются и завершаются упорядоченно, каждый регистр в любой точке программы содержит именно то значение, которое диктуется программой. При неупорядоченной выдаче и завершении команд запись в регистры может происходить также неупорядоченно, и отдельные команды, обратившись к какому-то регистру, вместо нужного получат «устаревшее» или «опережающее» значение. Пусть имеется последовательность команд:

I1: MUL R2, R0, R1 {R2 ← R0 × R1}

I2: ADD R0, R1, R2 {R0 ← R1 + R2}

I3: SUB R2, R0, R1 {R2 ← R0 − R1}.

Неупорядоченные выдачи/завершения могут привести к неверному результату, например:

- команда I2 была исполнена до того, как I1 успела записать в регистр R2 свой результат, то есть I2 использовала «старое» содержимое R2;
- команда I3 исполнена раньше, чем I1, в результате чего неверный результат будет получен в I2, а по завершении цепочки из трех команд в R2 останется результат I1 вместо результата I3.

Вводя новые регистры R0a и R2a, получим иную последовательность:

I1: MUL R2, R0, R1 {R2 ← R0 × R1}

I2: ADD R0a, R1, R2 {R0a ← R1 + R2}

I3: SUB R2a, R0a, R1 {R2a ← R0a − R1},

где возможность конфликта устранена. Такой метод известен как *переименование регистров* (register renaming).

Основная идея переименования регистров состоит в том, что каждый новый результат записывается в один из свободных в данный момент дополнительных регистров, при этом ссылки на заменяемый регистр во всех последующих командах соответственным образом корректируются. Программист, составляющий программу, имеет дело с именами логических регистров. Число физических регистров аппаратного регистрового файла (АРФ) обычно больше числа логических. «Лишние» регистры АРФ используются в процедуре переименования для временного хранения результатов до момента разрешения конфликтов по данным, после чего значение из регистра временного хранения переписывается на свое «штатное» место. В некоторых процессорах «лишние» регистры в АРФ отсутствуют, а для поддержки переименования предусмотрены специальные структуры, например так называемый *буфер переименования*.

На данном этапе будем считать, что дополнительные физические регистры входят в состав АРФ. Когда выполняется команда, предусматривающая запись результата в какой-то из логических регистров, например в R_i, для временного хранения выделяется один из свободных в данный момент физических регистров АРФ (R_j). Во всех последующих командах, где в качестве логического регистра операнда упоминается R_i, ссылка на него заменяется ссылкой на физический регистр R_j. Таким

образом, различные команды, где указан один и тот же логический регистр, могут обращаться к различным физическим регистрам.

Номера логических регистров динамически отображаются на номера физических регистров посредством *таблиц подстановки* (lookup table), которые обновляются после декодирования каждой команды. Очередной результат записывается в новый физический регистр, но значение каждого логического регистра запоминается, благодаря чему легко восстанавливается в случае, если выполнение команды должно быть прервано из-за возникновения исключительной ситуации или неправильного предсказания направления условного перехода.

Переименование регистров может быть реализовано и по-другому — с помощью *буфера переименования*. Проиллюстрируем этот способ, вернувшись к ранее приведенной последовательности из трех команд. Схема содержит *M* физических регистров и буфер переименования (БП) на *N* входов (рис. 9.31). Будем считать, что число логических регистров также равно *M*, то есть в качестве временных регистров переименования могут использоваться только ячейки буфера переименования.

Рис. 9.31. Регистры и буфер переименования

Каждому физическому регистру придан бит «Значение достоверно» (ЗД), единичное значение которого свидетельствует о том, что в регистре содержится корректное значение и оно может быть взято в качестве операнда команды.

Буфер переименования представляет собой ассоциативное запоминающее устройство или набор регистров с ассоциативным доступом. Каждая ячейка или регистр БП идентифицируется своим порядковым номером (0, 1, ..., $N - 1$). Информация, хранящаяся в ячейке или регистре буфера переименования, представляется пятью полями:

- *Вход занят* (ВЗ). Однобитовое поле, единичное значение которого говорит о том, что этот вход буфера переименования недоступен.
- *Номер переименованного регистра* (Рг). В поле содержится номер логического регистра, для временной замены которого выделена данная ячейка буфера переименования.
- *Значение*. В поле хранится текущее содержимое регистра, указанного в поле Рг.

- *Значение достоверно* (ЗД). Однобитовое поле, единичное значение которого подтверждает достоверность содержимого поля «Значение» (если значение еще не вычислено, то ЗД = 0).
- *Последнее переименование* (ПП). Если в буфере переименования несколько ячеек через поле Pr ссылаются на один и тот же регистр, единица в однобитовом поле ПП будет только у той ячейки, где находится последняя ссылка на данный регистр.

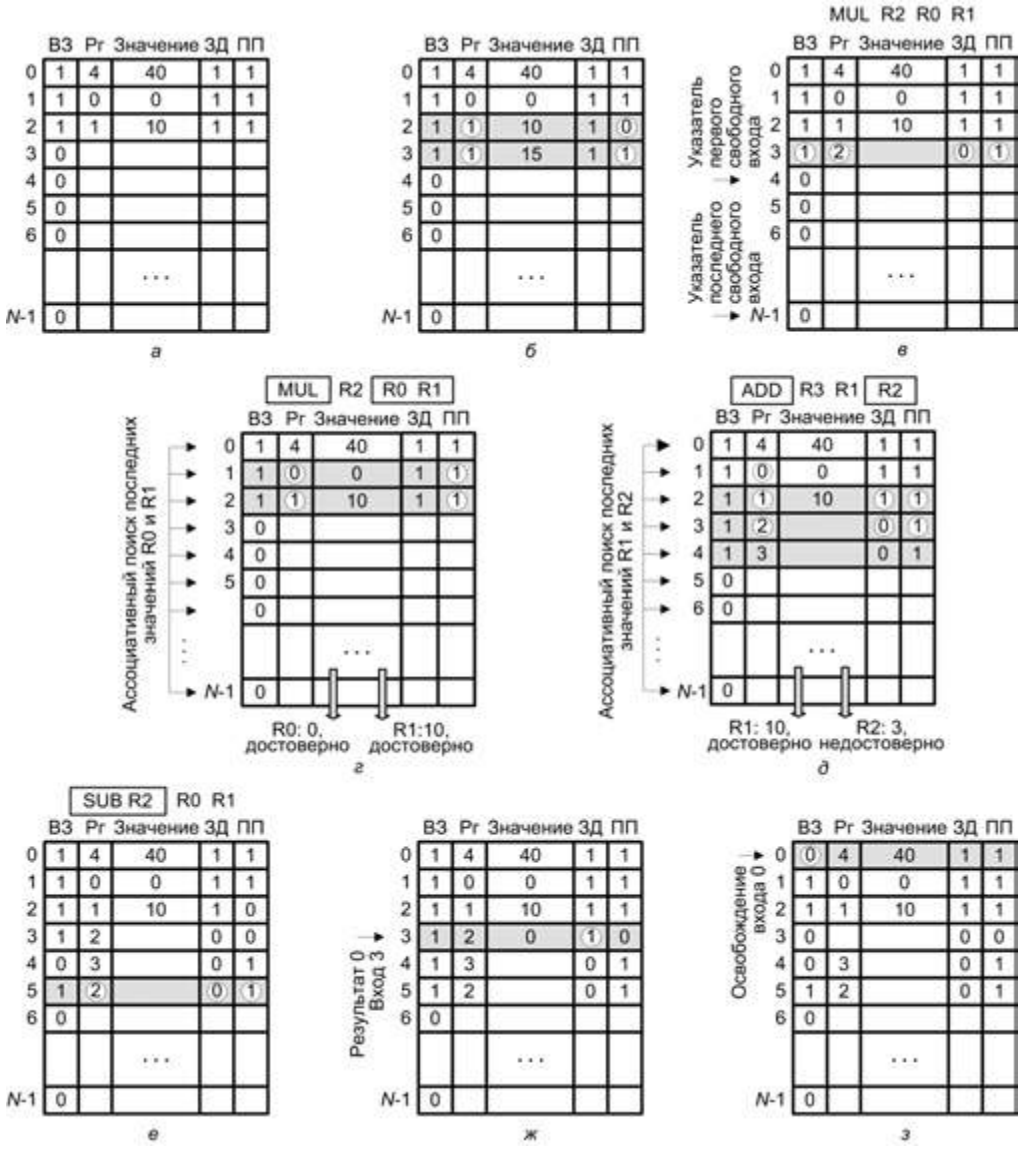


Рис. 9.32. Иллюстрация процессов в буфере переименования

В исходный момент (рис. 9.32, *а*) в буфере переименования заполнены три первых входа, о чем свидетельствуют единицы в поле ВЗ. Это состояние БП свидетельствует о том, что в предшествующих командах предполагалась запись результатов в логические регистры R4, R0 и R1, и хотя результаты этих команд уже получены (в поле ЗД записана единица), вычисленные значения еще не переписаны в соответствующие физические регистры.

Поле ПП введено из-за того, что регистры могут переименовываться многократно. На рис. 9.32, *б* показан случай последовательного переименования регистра R1. Единица в поле ПП входа 3 указывает, что последнему переименованию регистра R1 соответствует именно данный вход. У всех остальных входов, ссылающихся на «устаревшие» значения R1, в этом поле будет 0. Если очередной команде требуется значение из регистра R1, то из двух возможных чисел 10 и 15 будет взято 15, то есть значение того входа, где в поле ПП содержится единица.

Всякий раз, когда встречается команда, предполагающая запись результата в регистр, необходимо в буфере переименования выделить для этого регистра один из свободных входов (свободную ячейку) и правильно инициализировать его поля. Обычно буфер заполняется циклически, для чего имеются указатели первого и последнего свободного входов. Когда запрашивается свободный вход БП, используется первый из указателей, после чего он соответствующим образом корректируется. При освобождении одной из позиций в буфере переименования корректируется значение указателя последнего свободного входа. После выделения входа БП производится инициализация его полей: устанавливаются в единицу поля ВЗ и ПП, в поле Рг заносится номер регистра, а в поле ЗД помещается 0, означающий, что поле «Значение» еще не содержит достоверной информации. Рисунок 9.32, *в* иллюстрирует состояние буфера переименования после выделения входа для регистра R2 в команде MUL.

Теперь рассмотрим, каким образом из буфера извлекаются значения операндов (поиск операндов в БП производится, если они отсутствуют в физических регистрах). В нашем примере (рис. 9.32, *г*) это соответствует выборке операндов для команды MUL (значений регистров R0 и R1). Так как буфер переименования ассоциативный, то для получения значений операндов нужно произвести ассоциативный поиск последних значений регистров R0 и R1, то есть тех входов, где в поле Рг указаны искомые регистры, а в поле ПП содержится 1.

Если передаваемой далее команде требуется значение регистра, которое еще не вычислено (недостоверно), вместо значения выдается идентификатор (номер) соответствующего входа буфера и ставится пометка, что это не значение, а номер входа. Рисунок 9.32, *д* показывает такую ситуацию для команды ADD. Первое, что делается при переименовании этой команды, — выделение свободного входа для регистра R3, конкретно — входа 4. Далее должны быть извлечены значения регистров R1 и R2. Поскольку последнее переименование регистра R1 достоверно, выборка содержащегося в регистре значения может быть произведена так, как это было описано выше. Однако значение R2 еще не вычислено, поэтому вместо него в исполнительную часть процессора, где будет выполняться команда ADD, пересылается номер соответствующего входа буфера переименования (в нашем примере это 3).

Теперь в буфере переименования имеется несколько входов, ссылающихся на один и тот же регистр. Так, на рис. 9.32, *е* показана ситуация, когда команда SUB выдана до завершения команды MUL. В этом случае под регистр R2 выделен еще один вход (вход 5), в котором установлены соответствующие значения в полях ЗД и ПП. Одновременно содержимое поля ПП входа 3, где хранилось предыдущее описание регистра R2, изменено на 0. Таким образом, с данного момента все последующие команды, ссылающиеся на регистр R2 как источник операнда, будут переадресовываться на вход 5. Это будет продолжаться до появления новой команды, с регистром результата R2.

По завершении команды регистр результата должен быть модифицирован так, чтобы последующие команды могли получить доступ к вычисленному результату. Модификация базируется на идентификаторе входа буфера переименования, выделенного для запрошенного регистра результата. В нашем примере предположим, что завершилась команда MUL и результат 0 должен быть занесен во вход 3 (рис. 9.32, *ж*). В поле ЗД этого входа помещается единица, показывающая, что значение регистра R2 уже доступно.

Последний момент — это освобождение входа буфера переименования (рис. 9.32, *з*). Условие освобождения входа будет рассмотрено позже.

Переупорядочивание команд

После декодирования команд и переименования регистров команды передаются на исполнение. Как уже отмечалось, выдача команд в функциональные блоки может производиться неупорядоченно, по мере готовности. Поскольку порядок выполнения команд может отличаться от предписанного программой, необходимо обеспечить корректность их операндов (частично решается путем переименования регистров) и правильную последовательность занесения результатов в физические регистры. Одним из наиболее распространенных приемов решения этих проблем служит *переупорядочивание команд*. В его основе лежат использование *окна команд* — буферной памяти, куда помещаются все команды, прошедшие этапы декодирования и переименования регистров (последняя операция выполняется только с теми командами, которые записывают свой результат в регистры). Окно команд обеспечивает отсрочку передачи команд на исполнение до момента готовности операндов, а также нужную очередность завершения команд и загрузки их результатов в физические регистры. Эта техника известна также под названием *шелвинг* (shelving). Ниже рассматриваются два варианта окна команд — централизованное и распределенное.

Централизованное окно команд (scoreboard) впервые было предложено в 1964 году фирмой Cray и реализовано в модели CDC 6600. Окно команд представляет собой буферное запоминающее устройство, в котором хранится некоторое количество последних по времени извлеченных из памяти и декодированных команд, а также текущая информация о доступности ресурсов, привлекаемых для их исполнения. Функциями окна являются оперативное выявление команд, для исполнения которых уже доступны все необходимые операнды и ресурсы, и выдача таких команд на исполнение в соответствующие функциональные блоки.

Все извлеченные из памяти команды сразу же после их декодирования и, если это необходимо, переименования регистров заносятся в окно команд, причем с соблюдением порядка их следования в программе. Физически окно команд — это ассоциативное запоминающее устройство, где каждой команде выделяется одна ячейка, состоящая из нескольких полей:

- поля операции, где хранится код операции;
- двух полей операндов, предназначенных для хранения значений операндов, если они известны, либо информации о том, откуда эти операнды должны быть получены;
- поля результата, указывающего регистр, куда должен быть помещен результат выполнения данной команды;
- поля битов достоверности.

В окне также хранится текущая информация о доступности устройств обработки (функциональных блоков).

Функционирование окна команд тесно увязано с работой буфера переименования и может быть описано следующим образом. Каждая команда после декодирования и переименования регистров заносится в очередную свободную ячейку окна. Код операции помещается в поле операции. Если команда предполагает загрузку результата в регистр, то на этот регистр имеется ссылка в буфере переименования, и в поле результата заносится номер входа БП, в котором хранится последняя ссылка на данный регистр. Далее делается попытка заполнить поля операндов значениями операндов. Сначала производится поиск нужного значения в физических регистрах. Если бит ЗД регистра операнда установлен в 0 (значение недостоверно), это означает, что операндом является результат предыдущей операции и дальше следует искать в буфере переименования. Выполняется ассоциативный поиск ссылки на регистр в буфере переименования. При удачном исходе (в найденной ячейке БП биты ЗД и ПП установлены в единицу) значение операнда берется из буфера переименования. В любом варианте, при обнаружении достоверного значения операнда поле операнда ячейки окна заполняется найденным значением, а соответствующий этому полю бит достоверности (ЗД) устанавливается в единицу. Если же значение операнда еще не вычислено, то в поле операнда заносится идентификатор входа буфера переименования, где находится последняя ссылка на искомый регистр, при этом бит достоверности такого поля сбрасывается в 0.

Обновление информации о готовности операндов и доступности функциональных устройств выполняется в каждом цикле процессора.

Команда может быть считана из окна команд и выдана на исполнение после того, как будут занесены значения всех операндов, и лишь при условии, что нужный для исполнения этой команды функциональный блок свободен. После завершения исполнения операции, указанной в команде, производится запись полученного результата (если эта команда предполагает данное действие) в ту ячейку буфера переименования, на которую указывает поле результата. Одновременно производится ассоциативный доступ ко всем хранящимся в окне командам и в тех из них, где в полях операндов указан идентификатор обновленного входа БП, этот

идентификатор заменяется занесенным в регистр новым значением, с соответствующей коррекцией битов достоверности. Далее завершенная команда покидает окно команд. Удаление команды является основанием для перезаписи значения результата данной команды в физический регистр и удаления соответствующей записи из буфера переименования.

Отметим, что рассматриваемая технология предполагает схему распределения готовых команд по требуемым для их исполнения функциональным блокам, с одновременной проверкой их доступности. Эта функция названа *диспетчеризацией*.

В примере, приведенном на рис. 9.33, для команд I1, I2, I3 и I5 известны значения одного из операндов, и они вынуждены ожидать значения второго операнда. Команде I4 известны оба операнда, и при условии доступности ФБ, требуемого для ее исполнения, она вправе быть выдана из окна команд.

Команда	Поле операции	Поле результата	Поле операнда 1	Поле операнда 2	Поле битов достоверности	
					ЗД1	ЗД2
I1	ОП	ИД	ЗН	ИД	1	0
I2	ОП	ИД	ЗН	ИД	1	0
I3	ОП	ИД	ИД	ЗН	0	1
I4	ОП	ИД	ЗН	ЗН	1	1
I5	ОП	ИД	ИД	ЗН	0	1
...						

ОП - код операции; ИД - идентификатор регистра; ЗН - значение операнда;
ЗДi – бит достоверности значения i-го операнда (1–достоверно, 0–недостоверно)

Рис. 9.33. Содержимое окна команд

В каждом такте работы процессора готовыми к выдаче могут оказаться сразу несколько команд, и все готовые команды должны быть направлены в соответствующие функциональные блоки. Если имеется несколько однотипных блоков обработки, то в процессоре должна быть предусмотрена логика выбора одного из них.

После выдачи команды из окна ее позиция освобождается и может быть использована для загрузки новой команды. Вместе с тем, необходимо сохранить заданную программой последовательность команд. Эту задачу решают одним из двух методов. В первом из них, именуемом как *стек диспетчеризации*, после выдачи команд и освобождения позиций в окне последующие команды сдвигаются вниз, заполняя вновь доступные позиции и освобождая верхнюю часть окна. Новые команды всегда загружаются в верхнюю часть окна. В случае второго метода, с так называемым *блоком обновления регистров*, окно функционирует так же, как очередь типа FIFO, но производится общий сдвиг вниз, включая и освободившиеся позиции. Это упрощает логику работы централизованного окна.

Распределенное окно команд. В варианте распределенного окна команд на входе каждого функционального блока размещается буфер декодированных команд, называемый *накопителем команд* или *станцией резервирования* (reservation station).

Метод резервирования был разработан Р. Л. Томасуло в 1967 году и впервые воплощен в вычислительной системе IBM 360/91. После выборки и декодирования команды распределяются по станциям резервирования тех ФБ, где команда будет исполняться. В буфере команда запоминается и по готовности выдается в связанный с данным накопителем функциональный блок. Логика работы каждого накопителя аналогична централизованному окну команд. Выдача происходит только после того, как команда получит все необходимые операнды и при условии, что ФБ свободен. При обновлении содержимого буфера переименования производится доступ ко всем накопителям команд, и в них идентификаторы обновленных входов заменяются значениями операндов.

Отметим одну особенность рассматриваемой схемы: не требуется, чтобы операнд был обязательно занесен в отведенный для него регистр — он может быть ускоренно передан прямо в накопитель команд для немедленного использования или буферизирован там для последующего использования.

Число независимых команд, которые могут выполняться одновременно, варьируется от программы к программе, а также в пределах каждой программы. В среднем число таких команд равно 1–3, временами возрастая до 5–6. Механизм резервирования ориентирован на одновременную выдачу нескольких команд, что, как правило, легче реализовать с распределенным, а не централизованным окном команд, поскольку темп загрузки распределенных буферов обычно меньше, чем потенциальный темп выдачи команд. Пропускная способность линии связи между централизованным окном команд и функциональными блоками должна быть выше, чем в случае распределенного окна. Однако для централизованного окна характерно более эффективное использование емкости буфера.

Емкость накопителя команд в каждом функциональном блоке зависит от ожидаемого числа команд для этого блока. Типичный накопитель рассчитан на 1–3 команды. Если в одной из них одновременно готовы несколько команд, выдача их в ФБ производится в порядке занесения этих команд в накопитель.

В качестве инструмента для поддержания правильной последовательности исполнения команд (в случае нескольких параллельно работающих функциональных блоков) может быть использован *буфер восстановления последовательности*. Стратегия буфера восстановления последовательности (БВП) впервые была описана Смитом и Плескуном (Smith and Pleszkun) в 1988 году.

БВП представляет собой кольцевой буфер (рис. 9.34) с указателями головной и хвостовой части. *Указатель головной части* содержит адрес следующего свободного входа. Команды заносятся в БВП в порядке, определяемом программой. Каждая выданная команда помещается в следующую свободную ячейку буфера (говорят, что команде выделен очередной свободный вход БВП), причем выделение ячеек идет с соблюдением последовательности выдачи команд. Каждый занятый вход содержит также информацию о состоянии хранимой в нем команды: команда только выдана (i), находится в стадии исполнения (x) или уже завершена (f). Указатель хвостовой части показывает на команду, подлежащую удалению из БВП прежде других. Удаление команды разрешено, только если она завершена и все предшествующие ей команды уже удалены из буфера. Этот механизм гарантирует, что

команды покидают БВП строго по порядку. Очередность выполнения команд программы сохраняется благодаря тому, что заносить свои результаты в память или регистры разрешается лишь тем командам, которые покинули БВП.

Число входов в БВП в разных процессорах составляет от 5 (PowerPC 603) до 64 (SPARC64).

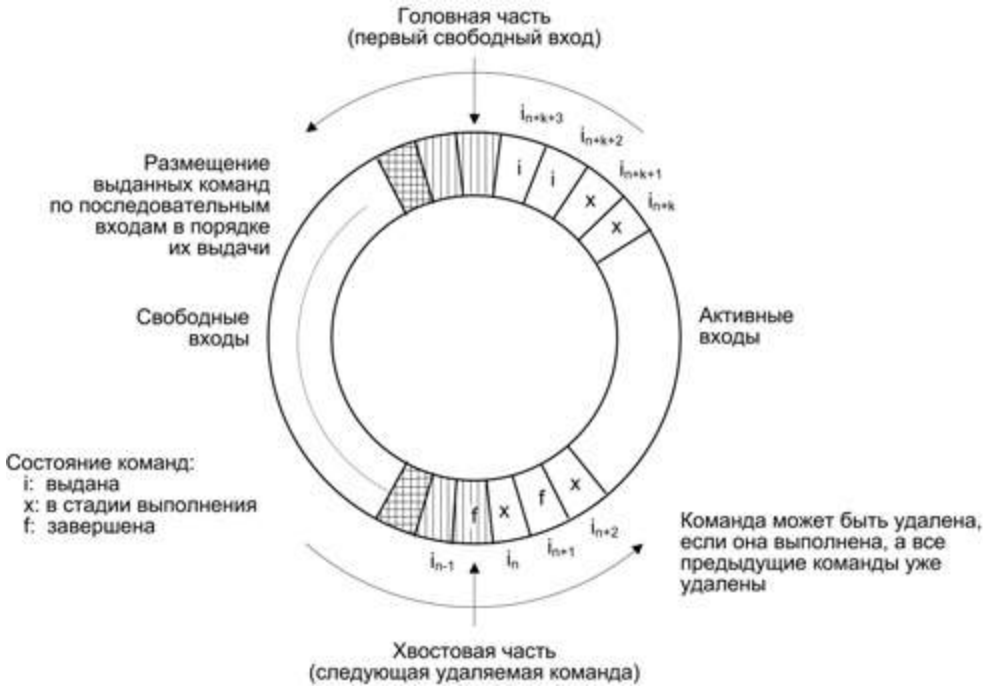


Рис. 9.34. Принципы организации буфера восстановления последовательности

Название буфера подчеркивает его основную задачу — поддержание строгой последовательности завершения команд путем переупорядочивания тех из них, которые исполнялись с нарушением этой последовательности. Однако БВП более универсален — с равным успехом он годится и для переименования регистров и для распределения декодированных команд по накопителям (станциям резервирования). Так, по своему основному назначению БВП применен в микропроцессорах PowerPC 603, PowerPC 604, R10000. В микропроцессорах Am29000, AMD K5, Pentium Pro буфер используется также для переименования регистров. Наконец, в системе Lightning БВП реализует все три из вышеперечисленных функций.

Гиперпотоковая обработка

В основе *гиперпотоковой технологии* (HTT — Hyper-Threading Technology), разработанной фирмой Intel и впервые реализованной в микропроцессоре Intel Xeon MP, лежит тот факт, что современные процессоры в большинстве своем являются суперскалярными и многоконвейерными, то есть выполнение команд в них идет

параллельно, по этапам, и на нескольких конвейерах сразу. Гиперпотоковая обработка призвана раскрыть этот потенциал таким образом, чтобы все функциональные блоки процессора были бы максимально загружены. Поставленная цель достигается за счет сочетания соответствующих аппаратных и программных средств.

Выполняемая программа разбивается на два параллельных *потока* или *нити* (threads). Задача компилятора (на стадии подготовки программы) и операционной системы (на этапе выполнения программы) заключается в формировании таких последовательностей независимых команд, которые процессор мог бы обрабатывать параллельно, по возможности заполняя функциональные блоки, не занятые одним из потоков, подходящими командами из другого, независимого потока.

Операционная система, поддерживающая гиперпотоковую технологию, воспринимает физический суперскалярный процессор как два логических процессора и организует поступление на эти два процессора двух независимых потоков команд.

Процессор с поддержкой HTT эмулирует работу двух одинаковых логических процессоров, принимая команды, направленные для каждого из них. Это не означает, что в процессоре имеются два вычислительных ядра — оба логических процессора конкурируют за ресурсы единственного вычислительного ядра. Следствием конкуренции является более эффективная загрузка всех ресурсов процессора.

В процессе вычислений физический процессор рассматривает оба потока команд и по очереди запускает на выполнение команды то из одного, то из другого или сразу их двух, если есть свободные вычислительные ресурсы. Ни один из потоков не считается приоритетным. При остановке одного из потоков (в ожидании какого-либо события или в результате зацикливания) процессор полностью переключается на второй поток. Возможность чередования команд из разных потоков принципиально отличает гиперпотоковую обработку от макропотоковой обработки (рассматриваемой позже).

Сказанное проиллюстрируем на примере суперскалярного процессора с тремя функциональными блоками, каждый из которых способен выполнять только свой тип операций (рис. 9.35). На рис. 9.35, *а* показан обычный суперскалярный процессор, способный обрабатывать потоки команд лишь последовательно. В двухпроцессорном варианте (рис. 9.35, *б*) параллельная обработка нитей обеспечивается тем, что каждый из двух аналогичных процессоров обрабатывает свой поток независимо от другого. При таком подходе в каждом из процессоров в определенные моменты времени остаются незанятые функциональные блоки, которые, однако, не могут быть использованы для обработки команд из другой нити. В варианте с гиперпотоковой обработкой (рис. 9.35, *в*) единственный процессор параллельно обрабатывает два потока, загружая функциональные блоки, незадействованные одним потоком, подходящими командами из другого потока. Очевидно, что функциональные блоки гиперпотокового процессора используются с максимальной эффективностью.

Наличие только одного вычислительного ядра не позволяет достичь удвоенной производительности, однако за счет большей отдачи от всех внутренних ресурсов общая скорость вычислений существенно возрастает. Это особенно ощущается, когда потоки содержат команды разных типов, тогда замедление обработки в одном из них компенсируется большим объемом работ, выполненных в другом потоке.

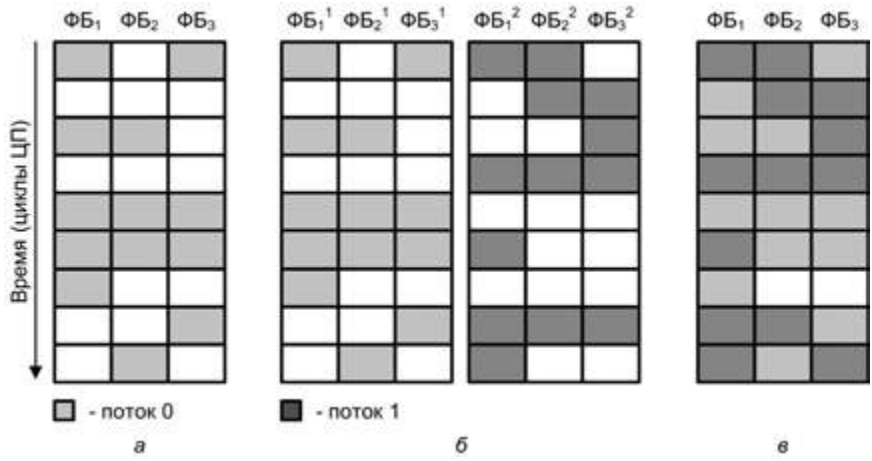


Рис. 9.35. Пример обработки потоков команд: *а* — стандартная суперскалярная архитектура; *б* — двухпроцессорная суперскалярная архитектура; *в* — гиперпотоковая архитектура

Следует учитывать, что эффективность HTT зависит от работы операционной системы, поскольку разделение команд на потоки осуществляет именно она.

Для иллюстрации рассмотрим некоторые особенности реализации гиперпотоковой технологии в процессоре Pentium 4 Xeon. Процессор способен параллельно обрабатывать два потока в двух логических процессорах. Чтобы выглядеть для операционной системы и пользователя как два логических процессора, физический процессор должен поддерживать информацию одновременно для двух отдельных и независимых потоков, распределяя между ними свои ресурсы. В зависимости от вида ресурса применяются три подхода: дублирование, разделение и совместное использование.

Дублированные ресурсы. Для поддержания двух полностью независимых потоков на каждом из логических процессоров некоторые ресурсы процессора необходимо дублировать. Прежде всего, это относится к счетчику команд, позволяющему каждому из логических процессоров отслеживать адрес очередной команды потока. Кроме того, каждому потоку выделяется свой набор регистров общего назначения (РОН) и регистров с плавающей запятой (РПЗ). С этой целью в процессоре имеются две таблицы распределения регистров (RAT, Register Allocation Table), каждая из которых обеспечивает отображение восьми регистров общего назначения (РОН) и восьми регистров с плавающей запятой (РПЗ), относящихся к одному логическому процессору, на совместно используемый регистровый файл из 128 РОН и 128 РПЗ. Таким образом, RAT — это дублированный ресурс, управляющий совместно используемым ресурсом (регистровым файлом).

Разделенные ресурсы. В качестве одного из видов разделенных ресурсов в Xeon выступают очереди (буферная память, организованная по принципу FIFO), расположенные между основными ступенями конвейера. Применяемое здесь разделение ресурсов можно условно назвать статическим: каждая буферная память (очередь)

разбивается пополам, и за каждым логическим процессором закрепляется своя половина очереди.

Применительно к другому виду очередей — очередям диспетчеризации команд (их в процессоре три) — можно говорить о динамическом разделении. Вместо фиксированного назначения двенадцати входов каждой очереди (например, входы 0–5 логическому процессору (ЛП) 0, а входы 6–11 — логическому процессору 1) каждому ЛП разрешается использовать любые входы очереди, лишь бы их общее число не превысило шести.

С позиций логического процессора и потока между статическим и динамическим разделением нет никакой разницы — в обоих случаях каждому ЛП выделяется своя половина ресурса. Различие становится существенным, если в качестве отправной точки взять физический процессор. Отсутствие привязки потоков к конкретным входам очереди позволяет не принимать во внимание, что имеются два потока, и расценивать обе половины как единую очередь. Очередь диспетчеризации команд просто просматривает каждую команду в общей очереди, оценивает зависимости между командами, проверяет доступность ресурсов, необходимых для выполнения команды, и планирует команду к исполнению. Таким образом, выдача команд на исполнение не зависит от того, какому потоку они принадлежат. Динамическое разделение очередей диспетчеризации команд предотвращает монополизацию очередей каким-либо одним из логических процессоров.

Отметим, что если процессор Xeon обрабатывает только один поток, то для обеспечения максимальной производительности этому потоку предоставляются все ресурсы процессора. В динамически разделяемых очередях снимаются ограничения на количество входов, доступных одному потоку, а в статических разделяемых очередях отменяется их разбиение на две половины.

Совместно используемые ресурсы. Этот вид ресурсов в гиперпотоковой технологии считается определяющим. Чем больше ресурсов могут совместно использовать логические процессоры, тем большую вычислительную мощность можно «снять» с единицы площади кристалла процессора. Первую группу общих ресурсов образуют функциональные (исполнительные) блоки: целочисленные операционные устройства, блоки операций с плавающей запятой и блоки обращения (чтения/записи) к памяти. Эти ресурсы «не знают», из какого ЛП поступила команда. То же самое можно сказать и о регистровом файле — второй разновидности совместно используемых ресурсов.

Сила гиперпотоковой технологии — общие ресурсы — одновременно является и ее слабостью. Проблема возникает, когда один поток монополизирует ключевой ресурс (такой, например, как блок операций с плавающей запятой), чем блокирует другой поток, вызывая его остановку. Задача предотвращения таких ситуаций возлагается на компилятор и операционную систему, которые должны образовать потоки, состоящие из команд с максимально различающимися требованиями к совместно используемым ресурсам. Так, один поток может содержать команды, нуждающиеся, главным образом, в блоке для операций с плавающей запятой, а другой — состоять преимущественно из команд целочисленной арифметики и обращения к памяти.

Третьей разновидностью общих ресурсов является кэш-память. Процессор Xeon предполагает работу с кэш-памятью трех уровней (L1, L2 и L3) и так называемой кэш-памятью трассировки. Оба логических процессора совместно используют одну и ту же кэш-память и хранящиеся в ней данные. Если поток, обрабатываемый логическим процессором 0, хочет прочитать некоторые данные, кэшированные логическим процессором 1, он может взять их из общего кэша. Из-за того, что в гиперпотоковом процессоре одну и ту же кэш-память используют сразу два логических процессора, вероятность конфликтов и, следовательно, вероятность снижения производительности возрастают.

Любой вид кэш-памяти одинаково трактует все обращения для чтения или записи, вне зависимости от того, какой из логических процессоров данное обращение производит. Это позволяет любому потоку монополизировать любую кэш-память, причем никакой защиты от монополизации процессор не имеет. Иными словами, физический процессор не в состоянии заставить логические процессоры сотрудничать при их обращении к кэш-памяти.

В целом, кэш-память — наиболее критичный из совместно используемых ресурсов, и конфликты за обладание ею сказываются на общей производительности процессора наиболее ощутимо.

По информации Intel, поддержка гиперпотоковой технологии в процессоре Pentium 4 Xeon потребовала 5% дополнительной площади на кристалле, в то же время прирост скорости вычислений может достигать 25–35%. В приложениях, ориентированных на многозадачность, программы ускоряются на 15–20%. Возможны, однако, ситуации, когда прироста в быстродействии не будет или он станет отрицательным. Таким образом, эффективность технологии находится в прямой зависимости от характера исполняемого программного приложения. Максимальная отдача достигается при работе серверных приложений (за счет разнообразия процессорных операций). Отметим, что в случае отключении режима HTT (такая возможность предусмотрена во всех гиперпотоковых процессорах) производительность процессора при обработке одиночного потока не ухудшается.

Несмотря на некоторое охлаждение интереса к гиперпотоковой технологии, связанное с ее зависимостью от характера программных приложений, фирма Intel продолжает использовать HTT в своих многоядерных процессорах. Так, в процессорах класса Core 2 каждое ядро, по-прежнему, одновременно поддерживает два потока команд, но общее число параллельно обрабатываемых потоков возрастает пропорционально количеству ядер.

Программная поддержка технологии предусмотрена во всех версиях операционных систем Windows, начиная с Windows 2000.

Архитектура процессоров

Архитектура процессоров непосредственно вытекает их архитектуры заложенной в них системы команд. Ранее уже упоминались наиболее распространенные архитектуры системы команд: CISC, RISC, VLIW. К перечисленным архитектурам

следует добавить EPIC, являющейся разновидностью архитектуры VLIW. Основными отличительными характеристиками каждой отдельной архитектуры являются:

- размер и функции регистров процессора (регистрового файла);
- способ и ограничения при обращении к памяти для чтения и записи;
- число операций, выполняемых одной командой;
- длина команд (переменная или фиксированная);
- число типов данных.

Обсудим их влияние на организацию процессоров.

Процессоры с архитектурой CISC

Основную идею CISC-архитектуры отражает ее название — «полный набор команд». В данной архитектуре стремятся иметь отдельную машинную команду для каждого возможного (типового) действия по обработке данных.

Исторически CISC-архитектура была одной из первых. Совершенствование процессоров шло по пути создания ВМ, способных выполнять как можно больше разных команд. Это упрощало работу программистов, которые писали программы на языке ассемблера (то есть практически на уровне машинных команд). Использование сложных команд позволяло сократить размер и время разработки программы.

В итоге сложились следующие черты организации CISC-процессоров:

- большое количество различных машинных команд (сотни), каждая из которых выполняется за несколько тактов центрального процессора;
- устройство управления с программируемой логикой;
- небольшое количество регистров общего назначения (РОН);
- различные форматы команд с разной длиной;
- преобладание двухадресной адресации;
- развитый механизм адресации операндов, включающий различные методы косвенной адресации.

CISC-подход, однако, привел к тому, что некоторые команды стало невозможно выполнять чисто аппаратными средствами (при разумной сложности таких средств). В результате в процессорах появились блоки, «на лету» заменяющие наиболее сложные команды последовательностями из более простых команд. Мало того, практика показала, что многие сложные команды при написании программ оказывались просто невостребованны. Наконец, из-за высокой сложности команд и их обилия устройство управления ВМ приходилось строить только на основе программируемой логики, то есть с применением «медленной» управляющей памяти. Последнее обстоятельство существенно ограничивало возможности наращивания тактовой частоты процессора. Все эти факторы привели к повороту в сторону RISC-архитектуры. В то же время целый ряд несомненных достоинств CISC-архитектуры сохраняют ее актуальность (прежде всего, в глазах разработчиков программных приложений). Именно поэтому ведущие фирмы-производители ВМ (Intel, AMD, IBM и др.) в своих последних разработках, по-прежнему, не отказываются от CISC-подхода.

Процессоры с архитектурой RISC

Особое внимание к RISC-архитектуре обусловлено тем, что в большинстве современных процессоров, относимых к классу CISC, сложные команды на стадии декодирования сводятся к набору простых RISC-команд, а ядро процессора реализуется как RISC-процессор.

Основные черты RISC-архитектуры

Главные усилия в архитектуре RISC направлены на построение максимально эффективного конвейера команд, то есть такого, где все команды извлекаются из памяти и поступают в ЦП на обработку в виде равномерного потока, причем ни одна команда не должна находиться в состоянии ожидания, а ЦП должен оставаться загруженным на протяжении всего времени. Кроме того, идеальным будет вариант, когда любой этап цикла команды выполняется в течение одного тактового периода.

Последнее условие относительно просто можно реализовать для этапа выборки. Необходимо лишь, чтобы все команды имели стандартную длину, равную ширине шины данных, соединяющей ЦП и память. Унификация времени исполнения для различных команд — значительно более сложная задача, поскольку наряду с регистровыми существуют также команды с обращением к памяти.

Помимо одинаковой длины команд важно иметь относительно простую подсистему декодирования и управления: сложное устройство управления (УУ) будет вносить дополнительные задержки в формирование сигналов управления. Очевидный путь существенного упрощения УУ — сокращение числа выполняемых команд, форматов команд и данных, а также способов адресации.

Очевидно, что в сокращенном списке команд должны оставаться лишь те, что используются наиболее часто. Исследования показали, что 80–90% времени выполнения типовых программ приходится на относительно малую часть команд (10–20%). К наиболее часто востребованным действиям относятся пересылка данных, арифметические и логические операции.

Основная причина, препятствующая сведению всех этапов цикла команды к одному тактовому периоду — потенциальная необходимость доступа к памяти для выборки операндов и/или записи результатов. По этой причине желательно максимально сократить число команд, имеющих доступ к памяти, что добавляет к ранее упомянутым принципам RISC еще два:

- доступ к памяти во время исполнения осуществляется только командами «Чтение» и «Запись»;
- все команды, кроме «Чтение» и «Запись», имеют формат «регистр-регистр».

Для упрощения выполнения большинства команд и приведения их к формату «регистр-регистр» требуется снабдить ЦП значительным числом регистров общего назначения. Это позволяет обеспечить временное хранение промежуточных результатов, используемых как операнды в последующих операциях, и ведет к уменьшению числа обращений к памяти, ускоряя выполнение операций. Минимальное число регистров, равное 32, принято как стандарт де-факто большинством производителей RISC-компьютеров.

Суммируя сказанное, концепцию RISC-процессора можно свести к следующим положениям:

- выполнение всех (или, по крайней мере, 75% команд) за один цикл;
- стандартная однословная длина всех команд, равная естественной длине слова и ширине шины данных и допускающая унифицированную конвейерную обработку всех команд;
- малое число команд (не более 128);
- малое количество форматов команд (не более 4);
- малое число способов адресации (не более 4);
- доступ к памяти только посредством команд «Чтение» и «Запись»;
- все команды, за исключением «Чтения» и «Записи», используют внутрипроцессорные межрегистровые пересылки;
- устройство управления с аппаратной логикой;
- относительно большой (не менее 32) процессорный файл регистров общего назначения (согласно [152] число РОН в современных RISC-микропроцессорах может превышать 500).

Регистры в RISC-процессорах

Отличительная черта RISC-архитектуры — большое число регистров общего назначения, что объясняется стремлением свести все пересылки к типу «регистр-регистр». Естественно, что увеличение числа РОН способно дать эффект лишь при разумном их использовании. Оптимизация использования регистров в RISC-процессорах обеспечивается как программными, так и аппаратными средствами.

Программная оптимизация выполняется на этапе компиляции программы, написанной на языке высокого уровня (ЯВУ). Компилятор стремится распределить регистры процессора таким образом, чтобы разместить в них те переменные, которые в течение заданного периода времени будут использоваться наиболее интенсивно.

На начальном этапе компилятор выделяет каждой переменной виртуальный регистр. Число виртуальных регистров в принципе не ограничено. Затем компилятор отображает виртуальные регистры на ограниченное количество физических регистров. Виртуальные регистры, использование которых не перекрывается, отображаются на один и тот же физический регистр. Если в определенном фрагменте программы физических регистров не хватает, то их роль для оставшихся виртуальных регистров выполняют ячейки памяти. В ходе вычислений содержимое каждой такой ячейки с помощью команды «Чтение» временно засылается в регистр, после чего командой «Запись» вновь возвращается в ячейку памяти.

Задача оптимизации состоит в определении того, каким переменным в данной точке программы выгоднее всего выделить физические регистры. Наиболее распространенный метод, применяемый для этой цели, известен как *раскраска графа*. В общем случае метод формулируется следующим образом. Имеется граф, состоящий из узлов и ребер. Необходимо раскрасить узлы так, чтобы соседние узлы имели разный цвет и чтобы при этом общее количество привлеченных цветов было минимальным. В нашем случае роль узлов выполняют виртуальные регистры. Если два

виртуальных регистра одновременно присутствуют в одном и том же фрагменте программы, они соединяются ребром. Делается попытка раскрасить граф в n цветов, где n — число физических регистров. Если такая попытка не увенчалась успехом, то узлам, которые не удалось раскрасить, вместо физических регистров выделяются ячейки в памяти.

На рис. 9.36 приведен пример раскраски графа [27], в котором шесть виртуальных регистров отображаются на три физических. Показаны временная последовательность активного вовлечения в выполнение каждого виртуального регистра (рис. 9.36, *а*) и раскрашенный граф (рис. 9.36, *б*).

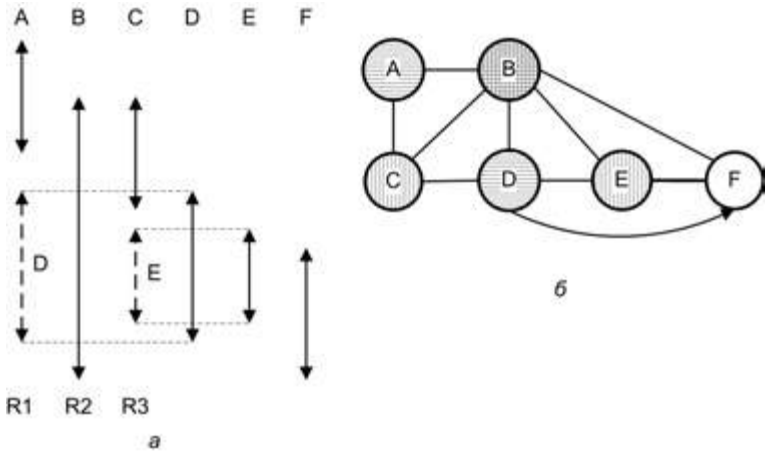


Рис. 9.36. Иллюстрация метода раскраски графа: *а* — временная последовательность активного использования виртуальных регистров; *б* — граф взаимного использования регистров

Как видно, не удалось раскрасить только виртуальный регистр F, его придется отображать на ячейку памяти.

Аппаратная оптимизация использования регистров в RISC-процессорах ориентирована на сокращение затрат времени при работе с процедурами. Наибольшее время в программах, написанных на языках высокого уровня, расходуется на вызовы процедур и возврат из них. Связано это с созданием и обработкой большого числа локальных переменных и констант. Одним из механизмов для борьбы с этим эффектом являются так называемые *регистровые окна*. Главная их задача — упростить и ускорить передачу параметров от вызывающей процедуры к вызываемой и обратно.

Регистровый файл разбивается на группы регистров, называемые окнами. Отдельное окно назначается глобальным переменным. Глобальные регистры доступны всем процедурам, выполняемым в системе в любое время. С другой стороны, каждой процедуре выделяется отдельное окно в регистровом файле. Все окна имеют одинаковый размер (обычно по 32 регистра) и состоят из трех полей. Левое поле каждого регистрового окна одновременно является и правым полем предшествующего ему

окна (рис. 9.37). Среднее поле служит для хранения локальных переменных и констант процедуры.

База окна (начальный адрес, номер окна) указывается полем, называемым указателем текущего окна (CWP, Current Window Pointer), обычно расположенным в регистре (слове) состояния ЦП. Если текущей процедуре назначено регистровое окно *j*, CWP содержит значение *j*.

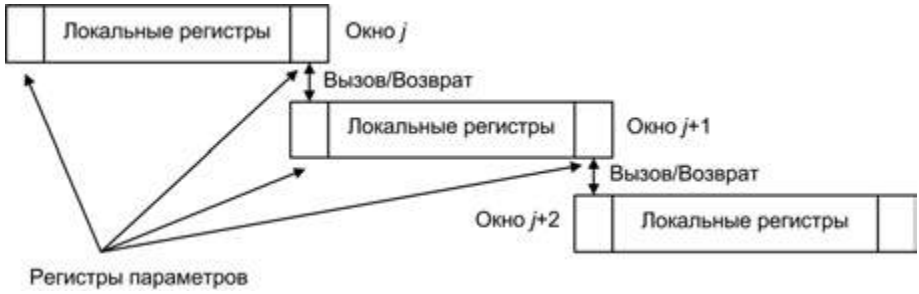


Рис. 9.37. Перекрытие регистровых окон

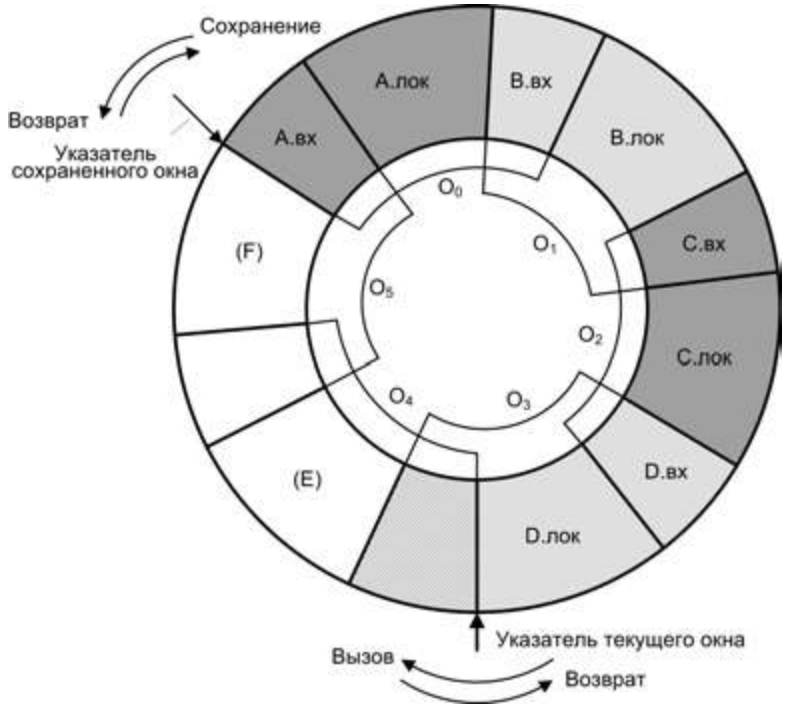


Рис. 9.38. Циклический буфер из пересекающихся регистровых окон

Каждой вновь вызванной процедуре выделяется регистровое окно, непосредственно следующее за окном вызвавшей ее процедуры. Последние *k* регистров окна *j*

одновременно являются первыми k регистрами окна $j + 1$. Если процедура, занимающая окно j, обращается к процедуре, которой в данной архитектуре должно быть назначено окно $j + 1$, она может передать в процессе вызова k аргументов. Упомянутые k регистров сразу же будут доступны вызванной процедуре без всяких пересылок. Естественно, вызов приведет к увеличению содержимого поля CPW на единицу.

Глубина вложения процедур одна в другую может быть весьма велика, и желательно, чтобы количество регистровых окон не было сдерживающим фактором. Это достигается за счет организации окон в виде циклического буфера.

На рис. 9.38 показан циклический буфер из шести окон, заполненный на глубину 4 (процедура A вызвала B, B вызвала C, C вызвала D). *Указатель текущего окна* (CWP) идентифицирует окно активной на данный момент процедуры — D, то есть окно O_3. При выполнении процедуры все ссылки на регистры в командах преобразуются в смещение относительно CWP. *Указатель сохраненного окна* (SWP, Saved Window Pointer) содержит номер последнего из окон, сохраненных в памяти по причине переполнения циклического буфера. Если процедура D теперь вызовет процедуру E, аргументы для нее она поместит в общее для обеих поле регистровых окон (пересечение окон O_3 и O_4), а значение CWP увеличится на единицу, то есть CWP будет показывать на окно O_4.

Если далее процедура E вызовет процедуру F, то этот вызов при существующем состоянии буфера не может быть выполнен, поскольку окно для F (O_5) перекрывается с окном процедуры A (O_0). Следовательно, при попытке F начать загружать правое поле своего окна будут потеряны параметры процедуры A ($A_{.вх}$). Поэтому когда CWP увеличивается на единицу (операция выполняется по модулю 6) и оказывается равным SWP, возникает прерывание, и окно процедуры A сохраняется в памяти (запоминаются только поля $A_{.вх}$ и $A_{.лок}$). Далее значение CWP инкрементируется и производится вызов процедуры F. Аналогичное прерывание происходит и при возврате, например когда выполнится возврат из B в A. CWP уменьшается на единицу (по модулю 6) и совпадет со значением SWP. Обработка прерывания приведет к восстановлению содержимого окна процедуры A из памяти.

Как видно из примера, регистровый файл из n окон способен поддерживать $n - 1$ вызов процедуры. Число n не должно быть большим. В [150] показано, что при 8 регистровых окнах сохранение и восстановление окон в памяти требуется лишь для 1% операций вызова процедур. В BM Pyramid, например, используется 16 окон по 32 регистра в каждом.

Теоретически такой прием не исключен и в CISC. Однако устройство управления CISC-процессора оккупирует на кристалле более 50% площади, оставляя мало места для других подсистем, в частности для большого файла регистров. УУ RISC-процессора занимает порядка 10% поверхности кристалла, предоставляя возможность иметь большой регистровый файл.

Другая часто встречаемая техника аппаратной оптимизации использования регистров — придание некоторым из них специального качества: такие регистры в состоянии принимать на себя имя любого РОН. Набор регистров, обладающих подобным свойством, называют *буфером переименования*. Прием оказывается очень

удобным при конвейеризации команд и позволяет предотвратить конфликты, когда одна команда хочет воспользоваться регистром, в данный момент занятым другой командой.

Преимущества и недостатки RISC

Сравнивая достоинства и недостатки CISC и RISC, невозможно сделать однозначный вывод о неоспоримом преимуществе одной архитектуры над другой. Для отдельных сфер использования ВМ лучшей оказывается та или иная. Тем не менее ниже приводится основная аргументация «за» и «против» RISC-архитектуры.

Для технологии RISC характерна сравнительно простая структура устройства управления. Площадь, выделяемая на кристалле микросхемы для реализации УУ, существенно меньше. Как следствие, появляется возможность разместить на кристалле большое число регистров ЦП. Кроме того, остается больше места для других узлов ЦП и для дополнительных устройств: кэш-памяти, блока арифметики с плавающей запятой, части основной памяти, блока управления памятью, портов ввода/вывода.

Унификация набора команд, ориентация на конвейерную обработку, унификация размера команд и длительности их выполнения, устранение периодов ожидания в конвейере — все эти факторы положительно сказываются на общем быстродействии. Простое устройство управления имеет немного элементов и, следовательно, короткие линии связи для прохождения сигналов управления. Малое число команд, форматов и режимов приводит к упрощению схемы декодирования, и оно происходит быстрее. Высокой производительности способствует и упрощение передачи параметров между процедурами. Таким образом, применение RISC ведет к сокращению времени выполнения программы или увеличению скорости за счет сокращения числа циклов на команду.

Многие современные CISC-машины имеют средства для прямой поддержки функций ЯВУ, наиболее частых в этих языках (управление процедурами, операции с массивами, проверка индексов массивов, защита информации, управление памятью и т. д.). RISC также обладает рядом средств для непосредственной поддержки ЯВУ и упрощения разработки компиляторов ЯВУ, благодаря чему эта архитектура в плане поддержки ЯВУ почти равна CISC.

Недостатки RISC прямо связаны с некоторыми преимуществами этой архитектуры. Принципиальный недостаток — сокращенное число команд: на выполнение ряда функций приходится тратить несколько команд вместо одной в CISC. Это удлиняет код программы, увеличивает загрузку памяти и трафик команд между памятью и ЦП. Исследования показали, что RISC-программа в среднем на 30% длиннее CISC-программы, реализующей те же функции.

Хотя большое число регистров дает существенные преимущества, само по себе оно усложняет схему декодирования номера регистра, тем самым увеличивается время доступа к регистрам.

УУ с аппаратной логикой, реализованное в большинстве RISC-систем, менее гибко, более склонно к ошибкам, затрудняет поиск и исправление ошибок, уступает при выполнении сложных команд.

Однословная команда исключает прямую адресацию для полноразрядного адреса, поэтому ряд производителей допускают небольшую часть команд двойной длины, например в Intel 80960.

Процессоры с архитектурой VLIW

Архитектура с командными словами сверхбольшой длины или *со сверхдлинными командами* (VLIW, Very Long Instruction Word) известна с начала 80-х из ряда университетских проектов. Идея VLIW базируется на том, что задача эффективного планирования параллельного выполнения команд возлагается на «разумный» компилятор. Такой компилятор вначале анализирует исходную программу. Цель анализа: обнаружить все команды, которые могут быть выполнены одновременно, причем так, чтобы между командами не возникали конфликты. В ходе анализа компилятор может даже частично имитировать выполнение рассматриваемой программы. На следующем этапе компилятор пытается объединить такие команды в пакеты (связки), каждый из которых рассматривается как одна сверхдлинная команда. Объединение нескольких простых команд в одну сверхдлинную производится по следующим правилам:

- количество простых команд, объединяемых в одну команду сверхбольшой длины, равно числу имеющихся в процессоре функциональных (исполнительных) блоков (ФБ);
- в сверхдлинную команду входят только такие простые команды, которые исполняются разными ФБ, то есть обеспечивается одновременное исполнение всех составляющих сверхдлинной команды.

Длина сверхдлинной команды обычно составляет от 256 до 1024 битов. Такая *метакоманда* содержит несколько полей (по числу образующих ее простых команд), каждое из которых описывает операцию для конкретного функционального блока. Сказанное иллюстрирует рис. 9.39, где показан возможный формат сверхдлинной команды и взаимосвязь между ее полями и ФБ, реализующими отдельные операции.

Как видно из рисунка, каждое поле сверхдлинной команды отображается на свой функциональный блок, что позволяет получить максимальную отдачу от аппаратуры блока исполнения команд.

В качестве простых команд, образующих сверхдлинную, обычно используются команды RISC-типа, поэтому архитектуру VLIW иногда называют постRISC-архитектурой. Максимальное число полей в сверхдлинной команде равно числу вычислительных устройств и обычно колеблется в диапазоне от 3 до 20. Все вычислительные устройства имеют доступ к данным, хранящимся в едином многопортовом регистровом файле.

VLIW-архитектуру можно рассматривать как статическую суперскалярную архитектуру. Имеется в виду, что распараллеливание кода производится на этапе компиляции, а не динамически во время исполнения. То, что в выполняемой

сверхдлинной команде исключена возможность конфликтов, позволяет предельно упростить аппаратуру VLIW-процессора и, как следствие, добиться более высокого быстродействия. Подавляющее большинство цифровых сигнальных процессоров и мультимедийных процессоров с производительностью более 1 млрд операций/с базируется на VLIW-архитектуре.

Двумя проблемами VLIW-архитектуры являются:

- усложнение регистрового файла и, прежде всего, связей этого файла с вычислительными устройствами;
- трудности создания компиляторов, способных найти в программе независимые команды, связать такие команды в длинные строки и обеспечить их параллельное выполнение.

В России VLIW-концепция была реализована в суперкомпьютере «Эльбрус 3-1» и получила дальнейшее развитие в его последователе — «Эльбрус-2000» (E2k).

Рис. 9.39. Формат сверхдлинной команды и взаимосвязь полей команды с составляющими блока исполнения

Процессоры с архитектурой EPIC

Дальнейшим развитием идеи VLIW стала архитектура EPIC (Explicitly Parallel Instruction Computing) — *вычисления с явным параллелизмом команд*. В архитектуре, возникшей как совместная разработка фирм Intel и Hewlett–Packard, реализован новый подход, являющийся усовершенствованным вариантом технологии VLIW. Первым представителем данной стратегии стал микропроцессор Itanium компании Intel. Корпорация Hewlett–Packard также реализует данный подход в своих разработках.

В архитектуре EPIC, которая в изделиях Intel получила название IA-64 (Intel Architecture – 64), предполагается наличие в процессоре ста двадцати восьми 64-разрядных регистров общего назначения (РОН) и ста двадцати восьми 80-разрядных регистров с плавающей запятой. Кроме того, процессор IA-64 содержит 64 однобитовых регистра предикатов.

Формат команд в архитектуре IA-64 показан на рис. 9.40.

Рис. 9.40. Формат сверхдлинной команды в архитектуре IA-64

Команды упаковываются (группируются) компилятором в сверхдлинную команду — *связку* (bundle) длиною в 128 разрядов. Связка содержит три команды и шаблон, в котором указываются зависимости между командами (можно ли с командой I_0 запустить параллельно I_1, или же I_1 должна выполниться только после I_0), а также между другими связками (можно ли с командой I_2 из связки S_0 запустить параллельно команду I_3 из связки S_1).

Перечислим все варианты составления связки из трех команд:

- $I_0 \parallel I_1 \parallel I_2$ — все команды исполняются параллельно;
- I_0 & $I_1 \parallel I_2$ — сначала I_0, затем исполняются параллельно I_1 и I_2;
- $I_0 \parallel I_1$ & I_2 — параллельно обрабатываются I_0 и I_1, после них — I_2;
- I_0 & I_1 & I_2 — команды исполняются в последовательности I_0, I_1, I_2.

Одна связка, состоящая из трех команд, соответствует набору из трех функциональных блоков процессора. Процессоры IA-64 могут содержать разное количество таких блоков, оставаясь при этом совместимыми по коду. Благодаря тому, что в шаблоне указана зависимость и между связками, процессору с N одинаковыми блоками из трех ФБ будет соответствовать сверхдлинная команда из $N \times 3$ команд (N связок). Тем самым обеспечивается масштабируемость IA-64.

Поле каждой из трех команд в связке, в свою очередь, состоит из пяти полей:

- 13-разрядного поля кода операции;
- 6-разрядного поля предикатов, хранящего номер одного из 64 регистров предиката;
- 7-разрядного поля первого операнда (первого источника), где указывается номер регистра общего назначения или регистра с плавающей запятой, в котором содержится первый операнд;

- 7-разрядного поля второго операнда (второго источника), где указывается номер регистра общего назначения или регистра с плавающей запятой, в котором содержится второй операнд;
- 7-разрядного поля результата (приемника), где указывается номер регистра общего назначения или регистра с плавающей запятой, куда должен быть занесен результат выполнения команды.

Следует пояснить роль поля предикатов. *Предикация* — это способ обработки условных ветвлений. Суть в том, что еще компилятор указывает, что обе ветви выполняются на процессоре параллельно, ведь EPIC-процессоры должны иметь много функциональных блоков.

Если в исходной программе встречается условное ветвление (по статистике — через каждые шесть команд), то команды из разных ветвей помечаются разными регистрами предиката (команды имеют для этого соответствующие поля), далее эти команды выполняются совместно, но их результаты не записываются, пока значения регистров предиката (РП) не определены. Когда, наконец, вычисляется условие ветвления, РП, соответствующий «правильной» ветви, устанавливается в 1, а другой — в 0. Перед записью результатов процессор проверяет поле предиката и записывает результаты только тех команд, поле предиката которых указывает на РП с единичным значением.

Предикаты формируются как результат сравнения значений, хранящихся в двух регистрах. Результат сравнения («Истина» или «Ложь») заносится в один из РП, но одновременно с этим во второй РП записывается инверсное значение полученного результата. Такой механизм позволяет процессору более эффективно выполнять конструкции типа IF-THEN-ELSE.

Логика выдачи команд на исполнение сложнее, чем в традиционных процессорах типа VLIW, но намного проще, чем у суперскалярных процессоров с неупорядоченной выдачей.

По мнению специалистов Intel и HP, концепция EPIC, сохраняя все достоинства архитектурной организации VLIW, свободна от большинства ее недостатков. Особенностями архитектуры EPIC являются:

- большое количество регистров;
- масштабируемость архитектуры до большого количества функциональных блоков. Это свойство представители компаний Intel и HP называют *наследственно масштабируемой системой команд* (Inherently Scaleable Instruction Set);
- явный параллелизм в машинном коде. Поиск зависимостей между командами осуществляет не процессор, а компилятор;
- предикация — команды из разных ветвей условного предложения снабжаются полями предикатов (полями условий) и запускаются параллельно;
- предварительная загрузка — данные из медленной основной памяти загружаются заранее.

Архитектура многоядерных процессоров

Многоядерный процессор — это центральный процессор, содержащий два и более вычислительных ядра на одном процессорном кристалле или в одном корпусе. Под ядром принято понимать процессор и кэш-память (обычно первого уровня — L1).

При наличии нескольких ядер необходимо обеспечить возможность их взаимодействия с основной памятью и между собой. Эта задача решается либо путем подключения ядер и памяти к общей шине, либо с помощью коммуникационной сети (рис. 9.41). Выбор варианта зависит от способа организации основной памяти. Если она является совместно используемой, то применяется вариант с шиной. В случае распределенной основной памяти коммуникации может обеспечить лишь сеть. Шинная организация обусловливает ограничение на число ядер. Пропускная способность шины ограничивает число ядер величиной 32, поскольку дальнейшее увеличение количества ядер ведет к снижению производительности многоядерного процессора.

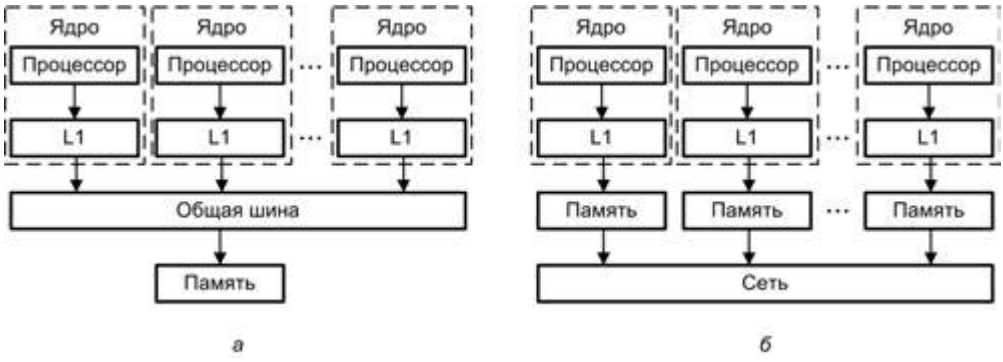


Рис. 9.41. Варианты многоядерных процессоров: *а* — с совместно используемой памятью; *б* — с разделенной памятью

В плане реализации многоядерные архитектуры различных производителей существенно различаются (рис. 9.42). Это касается не только системы коммуникаций, но и организации иерархии кэш-памяти. Во всех вариантах каждый процессор обладает кэш-памятью первого уровня. Кэш-память второго уровня может быть либо индивидуальной для каждого ядра, либо общей для группы ядер (обычно одна память L2 на пару ядер). В новейших многоядерных процессорах присутствует кэш-память третьего уровня L3. В известных микросхемах L3 является общей для всех ядер. Кроме того, каждый производитель отдает предпочтение своим архитектурным решениям [16]. Так, в процессорах фирмы Intel ядра поддерживают технологию Hyper-Threading. В процессорах фирмы AMD связь между ядрами организована непосредственно на процессорном кристалле, а не через материнскую плату. Это решение существенно повышает эффективность взаимодействия ядер.

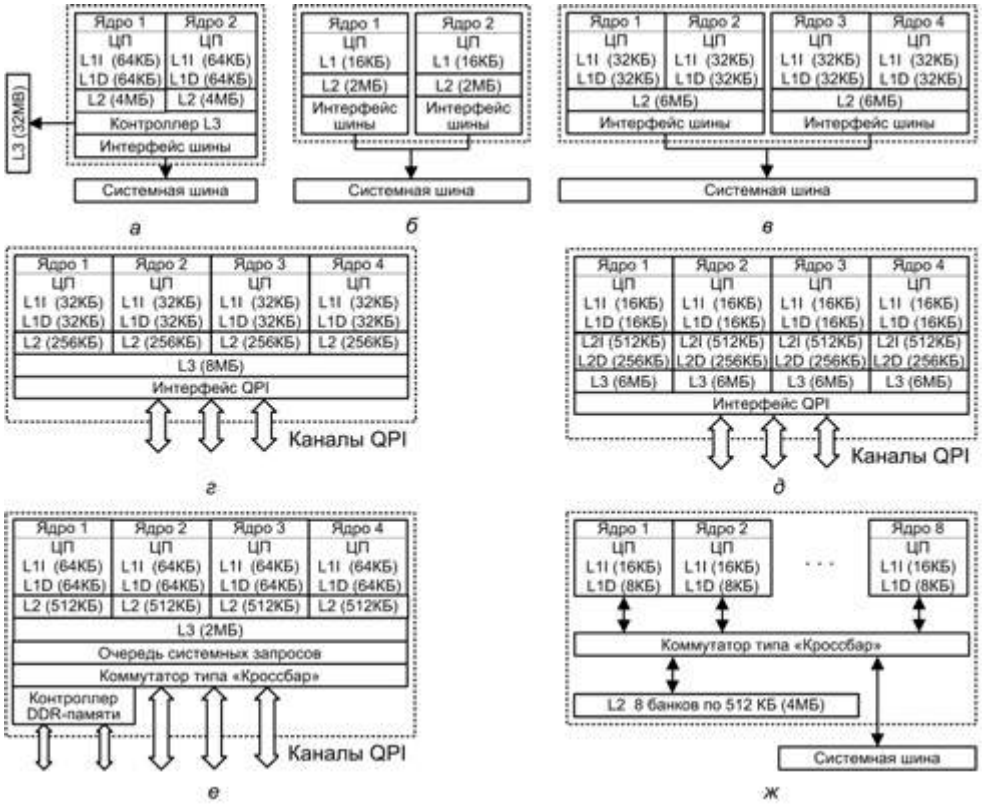


Рис. 9.42. Структура многоядерных процессоров различных производителей: *а* — IBM Power 6; *б* — Intel Pentium D; *в* — Intel Core 2 Quad; *г* — Intel Nehalem; *д* — Intel Itanium 3 Tukwila; *е* — AMD Phenom X4; *ж* — Sun UltraSPARC T2

Побудительной причиной для создания многоядерных процессоров стала необходимость повышения производительности ВМ. Возможности повышения тактовой частоты уже практически исчерпаны. Проблема может быть решена за счет максимального распараллеливания вычислений. В рамках одного ядра такое распараллеливание достигается, например, за счет гиперпотоковой обработки (ядром одновременно обслуживаются два потока). Так, 4 ядра позволяют обслуживать 8 программных потоков. Такая возможность обеспечивается, главным образом, операционной системой, поскольку в плане аппаратуры ядра достаточно самостоятельны. Особые средства для взаимодействия ядер, помимо системы коммуникаций и, возможно, общей кэш-памяти третьего уровня, обычно отсутствуют.

Применение многоядерной технологии позволяет увеличить производительность ВМ и одновременно избежать роста потребления энергии, которое накладывает ограничения на развитие одноядерных процессоров. Кроме того, чем выше частота процессора, тем больше он простаивает при обращении к памяти.

Поскольку частота памяти растет медленнее, чем частота процессоров, прирост производительности за счет нескольких ядер более предпочтителен.

Контрольные вопросы

1. В чем суть идеи конвейеризации? В каких случаях в конвейер следует вводить буферные регистры? В каких случаях буферные регистры нужно заменять буферной памятью? Ответы обоснуйте.
2. Определите синхронный конвейер из 8 функциональных блоков, из которых половина блоков работает в полтора раза медленнее других. Задайте конкретные временные параметры элементов конвейера. Рассчитайте значения трех метрик эффективности конвейера.
3. Пусть алгоритм вычисления программного разворота летательного аппарата (ЛА) имеет вид: $teta_i = (a \times v_i + b) \times v_i + c$, $v_i = v_{i-1} + \Delta v_i$, где $teta_i$ — программное значение угла тангажа; v_i — текущее значение скорости ЛА; v_{i-1} — предыдущее значение скорости ЛА; Δv_i — текущее приращение скорости ЛА; a, b, c — константы. Запрограммируйте этот алгоритм с помощью системы команд семейства Pentium. Составьте диаграмму выполнения своей программы на конвейере с шестью ступенями. Определите случаи структурного риска и объясните возможные пути их разрешения. Зафиксируйте случаи риска по данным и поясните, как они распознаются. Предложите пути их устранения.
4. Модифицируем предыдущий алгоритм следующим образом: $teta_i = (a \times \times v_i + b) \times v_i + c$, если $\Delta v_i > d$, тогда $v_i = v_{i-1} + \Delta v_i$, иначе $v_i = v_{i-1}$. Для соответствующей программы опишите случай нарушения ритмичности работы конвейера с шестью ступенями. Охарактеризуйте возможные способы уменьшения задержки конвейера и выберите один из них. Выбор обоснуйте.
5. В чем суть статического предсказания переходов? Сформулируйте достоинства и недостатки известных способов статического предсказания переходов.
6. В чем заключается смысл динамического предсказания переходов? Дайте развернутую характеристику достоинств и недостатков известных способов динамического предсказания переходов. Из каких соображений следует выбирать конкретную схему динамического предсказания? От чего зависит выбор?
7. Поясните идею суперконвейера. В чем заключаются достоинства и недостатки суперконвейеризации?
8. Поясните достоинства и недостатки ВМ с полным набором команд. Какие исторические причины привели к их возникновению?
9. Какие исторические причины способствовали появлению ВМ с сокращенным набором команд?
10. Перечислите основные характеристики ВМ с сокращенным набором команд.
11. Опишите возможности совместного использования в одной ВМ CISC-архитектуры и RISC-архитектуры.
12. Для чего вводится механизм регистровых окон? Поясните структуру окна. Ради какой цели окна организуются в виде циклического буфера?
13. Обоснуйте основные недостатки ВМ с сокращенным набором команд.

14. Дайте развернутую характеристику назначения и структурной организации суперскалярного процессора. Какие уровни параллелизма здесь используются?
15. На конкретных примерах поясните суть проблемы неупорядоченности команд в суперскалярных процессорах.
16. Разработайте структуру конкретного суперскалярного процессора. Задайте несколько программных фрагментов и составьте диаграммы их выполнения для трех методик выдачи и завершения команд.
17. На примере конкретного программного фрагмента поясните суть метода переименования регистров. В чем состоят недостатки этого метода? Как их смягчить?
18. На примере конкретного программного фрагмента поясните смысл метода переупорядочивания команд. В чем заключается разница между централизованным и распределенным окном команд? Проведите сравнительный анализ организации этих окон.
19. Опишите назначение, организацию и порядок работы буфера восстановления последовательности. Для решения каких задач он применяется в современных процессорах?
20. Каким образом и при каких условиях гиперпотоковая обработка способствует повышению производительности процессора?
21. Сформулируйте правила объединения простых команд в командное слово сверхбольшой длины.
22. Чем ограничивается количество объединяемых команд в технологии EPIC?
23. Поясните назначение системы предикации и ее реализацию в архитектуре IA-64.

Глава 10

Параллельные вычисления

Фон-неймановская архитектура ориентирована на последовательное исполнение команд программы. В условиях постоянно возрастающих требований к производительности вычислительной техники все очевидней становятся ограничения такого подхода. Дальнейшее развитие компьютерных средств связано с переходом к параллельным вычислениям, как в рамках одной ВМ, так и путем создания многопроцессорных систем и сетей, объединяющих большое количество отдельных процессоров или отдельных вычислительных машин. Для такого подхода вместо термина «вычислительная машина» более подходит термин «вычислительная система» (ВС). Отличительной особенностью вычислительных систем является наличие в них средств, реализующих параллельную обработку, за счет построения параллельных ветвей в вычислениях, что не предусматривалось классической структурой ВМ.

Уровни параллелизма

Методы и средства реализации параллелизма зависят от того, на каком уровне он должен обеспечиваться (рис. 10.1). Обычно различают следующие *уровни параллелизма*:

- *Микроуровень*. Выполнение команды разделяется на фазы, а фазы нескольких соседних команд могут быть перекрыты за счет конвейеризации. Уровень достижим на ВС с одним процессором.
- *Уровень команд*. Выражается в параллельном выполнении нескольких команд и достигается посредством размещения в процессоре сразу нескольких конвейеров. Реализуется в суперскалярных процессорах.
- *Уровень потоков*. Задачи разбиваются на части, которые могут выполняться параллельно (потоки). Данный уровень достигается на параллельных ВС.
- *Уровень заданий*. Несколько независимых заданий одновременно выполняются на разных процессорах, практически не взаимодействуя друг с другом. Этот уровень реализуется на многопроцессорных и многомашинных ВС.

Рис. 10.1. Уровни параллелизма

К понятию уровня параллелизма тесно примыкает понятие *гранулярности*. Это мера отношения объема вычислений, выполненных в параллельной задаче, к объему коммуникаций (для обмена сообщениями). Степень гранулярности варьируется от мелкозернистой до крупнозернистой. Определим понятия крупнозернистого (coarse grained), среднезернистого (medium grained) и мелкозернистого (fine grained) параллелизма.

Крупнозернистый параллелизм: каждое параллельное вычисление достаточно независимо от остальных, причем требуется относительно редкий обмен информацией между отдельными вычислениями. Единицами распараллеливания являются большие и независимые программы, включающие тысячи команд. Этот уровень параллелизма обеспечивается операционной системой.

Среднезернистый параллелизм: единицами распараллеливания являются вызываемые процедуры, включающие в себя сотни команд. Обычно организуется как программистом, так и компилятором.

Мелкозернистый параллелизм: каждое параллельное вычисление достаточно мало и элементарно, составляется из десятков команд. Обычно распараллеливаемыми единицами являются элементы выражения или отдельные итерации цикла, имеющие небольшие зависимости по данным. Сам термин «мелкозернистый параллелизм» говорит о простоте и быстроте любого вычислительного действия. Характерная особенность мелкозернистого параллелизма заключается в приблизительном равенстве интенсивности вычислений и обмена данными. Этот уровень параллелизма часто используется распараллеливающим (векторизирующим) компилятором.

Эффективное параллельное исполнение требует искусного баланса между степенью гранулярности программ и величиной коммуникационной задержки, возникающей между разными гранулами. В частности, если коммуникационная задержка минимальна, то наилучшую производительность обещает мелкоструктурное разбиение программы. Это тот случай, когда действует параллелизм данных. Если коммуникационная задержка велика, предпочтительней крупнозернистое разбиение программ.

Метрики параллельных вычислений

Метрики параллельных вычислений — это система показателей, позволяющая оценить преимущества, получаемые при параллельном решении задачи на n процессорах, по сравнению с последовательным решением той же задачи на единственном процессоре. С другой стороны, они позволяют судить об обоснованности применения данного числа процессоров для решения конкретной задачи. Под *параллельными вычислениями* будем понимать последовательность шагов, где каждый шаг состоит из i операций, выполняемых одновременно набором из i параллельно работающих процессоров.

Базисом для определения упомянутых метрик являются следующие характеристики вычислений:

- n — количество процессоров, используемых для организации параллельных вычислений;
- $O(n)$ — объем вычислений, выраженный через количество операций, выполняемых n процессорами в ходе решения задачи;
- $T(n)$ — общее время вычислений (решения задачи) с использованием n процессоров.

Условимся, что время изменяется дискретно, а за один квант времени процессор выполняет любую операцию. Вследствие этого справедливы следующие соотношения для времени и объема вычислений: $T(1) = O(1)$, $T(n) \leqslant O(n)$. Последнее соотношение формулирует утверждение: *время вычислений можно сократить за счет распределения объема вычислений по нескольким процессорам*.

Профиль параллелизма программы

Число процессоров, параллельно выполняющих программу в каждый момент времени t, задает *степень параллелизма* $P(t)$ (Degree Of Parallelism). Графическое представление параметра P в функции времени называют *профилем параллелизма программы*. Изменения числа параллельно работающих процессоров (за время наблюдения) зависят от многих факторов (алгоритма, доступных ресурсов, степени оптимизации, обеспечиваемой компилятором и т. д.). Типичный профиль параллелизма для алгоритма декомпозиции (divide-and-conquer algorithm) показан на рис. 10.2.

Предположим, что система состоит из n гомогенных (однородных) процессоров. Производительность Δ одиночного процессора системы выразим как количество операций в единицу (квант) времени, не учитывая издержек, связанных с обращением к памяти и пересылкой данных. Если за наблюдаемый период (определенное количество квантов времени) загружены i процессоров, то $P = i$. Общий объем вычислений $O(n)$ за период от стартового момента $t_н$ до момента завершения $t_к$ пропорционален площади под кривой профиля параллелизма:

$$O(n) = \Delta \sum_{i=1}^{n} i t_i,$$

где t_i — интервал времени (общее количество квантов времени), в течение которого $P = i$, а $\sum_{i=1}^{n} t_i = t_к - t_н$ — общее время вычислений.

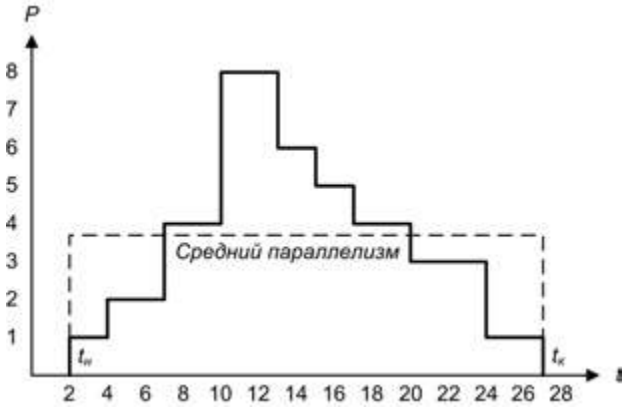


Рис. 10.2. Профиль параллелизма

Средний параллелизм A определяется как

$$A = \frac{\sum_{i=1}^{n} it_i}{\sum_{i=1}^{n} t_i}.$$

Профиль параллелизма на рисунке за время наблюдения ($t_н$, $t_к$) возрастает от 1 до пикового значения $n = 8$, а затем спадает до 0. Средний параллелизм $A = (1 \times 5 + 2 \times 3 + 3 \times 4 + 4 \times 6 + 5 \times 2 + 6 \times 2 + 8 \times 3) / (5 + 3 + 4 + 6 + 2 + 2 + 0 + 3) = 93/25 = 3{,}72$.

Основные метрики

По существу, можно выделить четыре группы метрик.

Первая группа характеризует скорость вычислений. Эта группа представлена парой метрик — *индексом параллелизма* и *ускорением*.

Индекс параллелизма (Parallel Index) характеризует среднюю скорость параллельных вычислений через количество выполненных операций:

$$PI(n) = \frac{O(n)}{T(n)}.$$

Ускорение (Speedup) за счет параллельного выполнения программы служит показателем эффективной скорости вычислений. Вычисляется ускорение как отношение времени, затрачиваемого на проведение вычислений на однопроцессорной ВС (в варианте наилучшего последовательного алгоритма), ко времени решения той же задачи на параллельной n-процессорной системе (при использовании наилучшего параллельного алгоритма):

$$S(n) = \frac{T(1)}{T(n)}.$$

Замечания относительно алгоритмов решения задачи должны подчеркнуть тот факт, что для последовательной и параллельной реализации лучшими могут оказаться разные алгоритмы, а при оценке ускорения необходимо исходить именно из наилучших алгоритмов.

Вторую группу образуют метрики *эффективность* и *утилизация*, дающие возможность судить об эффективности привлечения к решению задачи дополнительных процессоров.

Эффективность (Efficiency) характеризует целесообразность наращивания числа процессоров через ту долю ускорения, достигнутого за счет параллельных вычислений, которая приходится на один процессор:

$$E(n) = \frac{S(n)}{n} = \frac{T(1)}{nT(n)}.$$

Утилизация (Utilization) учитывает вклад каждого процессора при параллельном вычислении, но в виде количества операций, выполненных процессором в единицу времени.

$$U(n) = \frac{O(n)}{nT(n)}.$$

Третья группа метрик, — *избыточность* и *сжатие*, — характеризует эффективность параллельных вычислений путем сравнения объема вычислений, выполненного при параллельном и последовательном решении задачи.

Избыточность (Redundancy) — это отношение объема параллельных вычислений к объему эквивалентных последовательных вычислений:

$$R(n) = \frac{O(n)}{O(1)}.$$

Важность данной метрики в том, что она исходит не из относительных показателей ускорения и эффективности, полученных из времени вычислений, а из абсолютных показателей, базирующихся на объеме выполненной вычислительной работы. Избыточность отражает степень соответствия между программным и аппаратным параллелизмом.

Отметим, что утилизация может быть выражена через метрики избыточности и эффективности:

$$U(n) = \frac{O(n)}{nT(n)} = R(n)E(n).$$

Здесь учитывается соотношение $T(1) = O(1)$.

Сжатие (Compression) вычисляется как величина, обратная избыточности:

$$C(n) = \frac{O(1)}{O(n)}.$$

Наконец, четвертую группу образует единственная метрика — *качество*, объединяющая три рассмотренных группы метрик.

Качество (Quality) определяется как:

$$Q(n) = \frac{T^3(1)}{nT^2(n)O(n)} = S(n)E(n)C(n).$$

Так как эта метрика увязывает метрики ускорения, эффективности и сжатия, она является более объективным показателем улучшения производительности за счет параллельных вычислений.

Для примера определим численные значения метрик применительно к задаче, использованной для иллюстрации понятия профиля параллелизма (рис. 10.2). Полагая, что наилучший алгоритм для последовательного и параллельного вычисления совпадают, имеем: $n = 8$; $T(1) = O(1) = O(8) = 93$; $T(8) = 25$. Тогда:

$$PI(8) = \frac{O(8)}{T(8)} = \frac{93}{25} = 3,72; \qquad S(8) = \frac{T(1)}{T(8)} = \frac{93}{25} = 3,72;$$

$$E(8) = \frac{T(1)}{8T(8)} = \frac{93}{8 \times 25} = 0,465; \qquad U(8) = \frac{O(8)}{8T(8)} = \frac{93}{8 \times 25} = 0,465;$$

$$R(8) = \frac{O(8)}{O(1)} = \frac{93}{93} = 1; \qquad C(8) = \frac{O(1)}{O(8)} = \frac{93}{93} = 1;$$

$$Q(8) = S(8)E(8)C(8) = 3,72 \times 0,465 \times 1 = 1,73.$$

В завершение отметим, что для рассмотренных метрик справедливы следующие соотношения:

$$1 \le S(n) \le PI(n) \le n; \quad 1 \le R(n) \le \frac{1}{E(n)} \le n; \quad \frac{1}{n} \le E(n) \le C(n) \le 1;$$

$$\frac{1}{n} \le E(n) \le U(n) \le 1; \quad Q(n) \le S(n) \le PI(n) \le n.$$

Закономерности параллельных вычислений

Приобретая для решения своей задачи параллельную вычислительную систему, пользователь рассчитывает на значительное повышение скорости вычислений за счет распределения вычислительной нагрузки по множеству параллельно работающих процессоров. В идеальном случае система из n процессоров могла бы ускорить вычисления в n раз. В реальности достичь такого показателя не удается из-за невозможности полного распараллеливания ни одной из задач. Как правило, в каждой программе имеется фрагмент кода, который принципиально должен выполняться последовательно и только одним из процессоров. Это может быть часть программы, отвечающая за запуск задачи и распределение распараллеленного кода

по процессорам, либо фрагмент программы, обеспечивающий операции ввода/вывода. Распараллелена может быть лишь оставшаяся часть программы. Таким образом, программный код решаемой задачи состоит из двух частей: последовательной и распараллеливаемой. Обозначим долю операций, которые должны выполняться последовательно одним из процессоров, через f, где $0 \le f \le 1$ (здесь доля понимается не по числу строк кода, а по числу реально выполняемых операций). Доля, приходящаяся на распараллеливаемую часть программы, составит $1 - f$. Крайние случаи в значениях f соответствуют полностью распараллеливаемым ($f = 0$) и полностью последовательным ($f = 1$) программам. Данную ситуацию иллюстрирует рис. 10.3, в котором использованы следующие обозначения:

- t_s — время обработки последовательной части программы с использованием одного процессора;
- $t_p(1)$ — время обработки распараллеливаемой части программы с использованием одного процессора;
- $t_p(n)$ — время обработки распараллеливаемой части программы с использованием n процессоров.

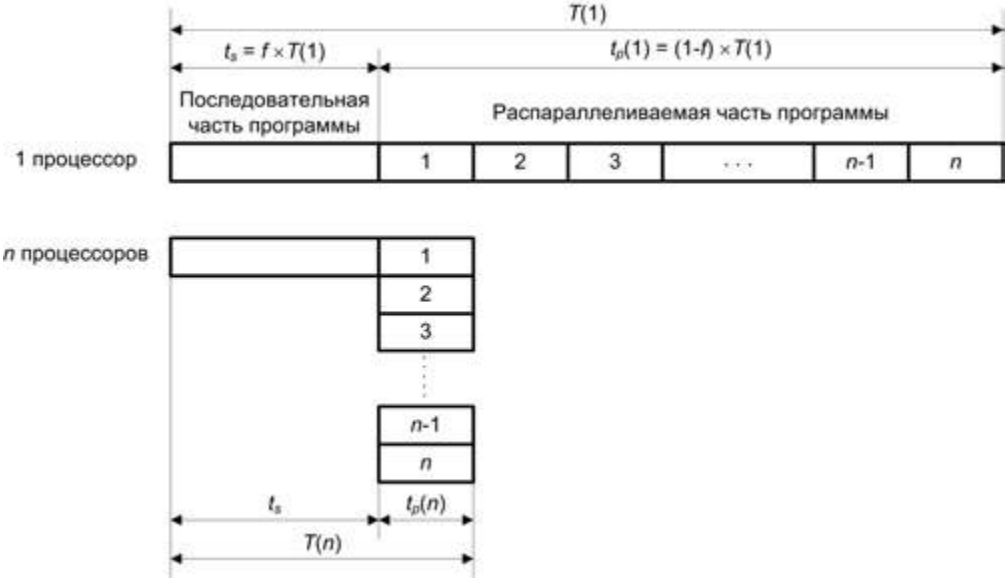


Рис. 10.3. Иллюстрация распараллеливания задачи в n-процессорной вычислительной системе

Для распараллеливаемой части задачи идеальным был бы вариант, когда параллельные ветви программы постоянно загружают все процессоры системы, причем так, чтобы нагрузка на каждый процессор была одинакова. К сожалению, оба этих условия на практике трудно реализуемы. Таким образом, ориентируясь на параллельную ВС, необходимо четко сознавать, что добиться увеличения производительности прямо пропорционального числу процессоров не удастся, и, естественно,

встает вопрос о том, на какое реальное ускорение можно рассчитывать. Ответ зависит от того, каким образом пользователь собирается использовать вычислительные мощности ВС, возросшие в результате увеличения числа процессоров. Наиболее характерными являются три варианта:

1. Объем вычислений не изменяется, а главная цель расширения ВС — сократить время вычислений. Достижимое в этом случае ускорение определяется законом Амдала.
2. Время вычислений с расширением системы не меняется, но при этом увеличивается объем решаемой задачи. Цель такого подхода — за заданное время выполнить максимальный объем вычислений. Эту ситуацию характеризует закон Густафсона.
3. Данный вариант похож на предыдущий, но с одним условием: увеличение объема решаемой задачи ограничено емкостью доступной памяти. Ускорение в такой формулировке определяет закон Сана и Ная.

Закон Амдала

Джин Амдал (Gene Amdahl) — один из разработчиков всемирно известной системы IBM 360, в своей работе [43], опубликованной в 1967 году, предложил формулу, отражающую зависимость ускорения вычислений, достигаемого на многопроцессорной ВС, как от числа процессоров, так и от соотношения между последовательной и распараллеливаемой частями программы. Проблема рассматривалась Амдалом исходя из положения, что объем решаемой задачи (рабочая нагрузка — число выполняемых операций) с изменением числа процессоров, участвующих в ее решении, остается неизменным (рис. 10.4). Такая постановка характерна для случаев, когда основным требованием является скорость вычислений, например, для задач, выполняемых в реальном времени. На рисунке o_s — это объем вычислений, обусловленных последовательной частью программы; $o_p(n)$ — объем вычислений, приходящийся на распараллеливаемую часть программы, и распределяемый на n процессоров; $O(n)$ — общий объем вычислений, выполняемый n-процессорной системой.

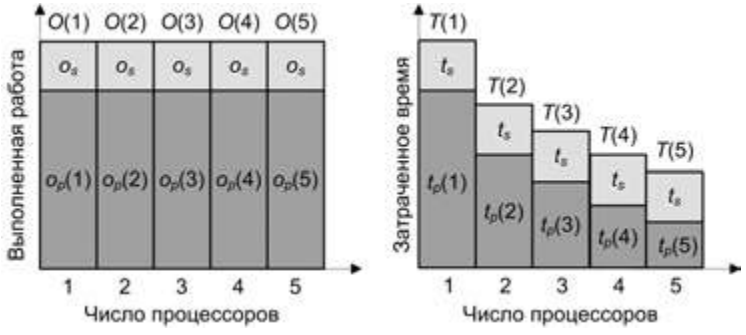


Рис. 10.4. Распределение рабочей нагрузки и времени вычислений в формулировке Амдала

С учетом «замораживания» объема задачи закон Амдала может быть представлен следующим выражением:

$$S(n) = \frac{T(1)}{T(n)} = \frac{T(1)}{fT(1) + \dfrac{(1-f)T(1)}{n}} = \frac{1}{f + \dfrac{1-f}{n}}.$$

При безграничном увеличении числа процессоров имеем:

$$\lim_{n\to\infty} S = \frac{1}{f}.$$

Это означает, что если в программе 25% последовательных операций (то есть f = = 0,25), то сколько бы процессоров не использовалось, ускорения работы программы более чем в четыре раза никак не получить, да и то и 4 — это теоретическая верхняя оценка самого лучшего случая, когда никаких других негативных факторов нет.

Характер зависимости ускорения от числа процессоров и доли последовательной части программы показан на рис. 10.5.

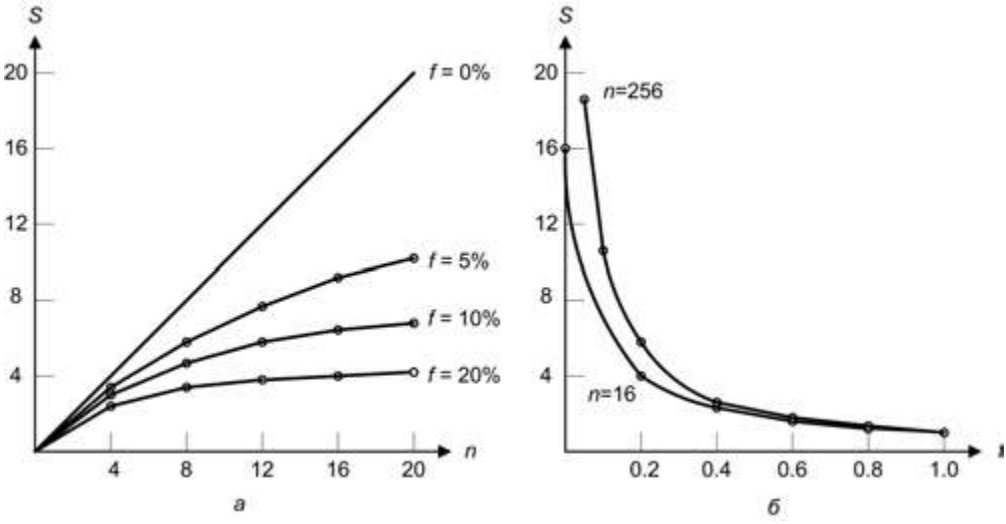


Рис. 10.5. Графики зависимости ускорения от: *а* — доли последовательных вычислений; *б* — числа процессоров

Следует отметить, что распараллеливание ведет к определенным издержкам, которых нет при последовательном выполнении программы. В качестве примера таких издержек можно упомянуть дополнительные операции, связанные с обменом информацией между процессорами. С учетом временных затрат, обусловленных такими издержками (T_o), формула Амдала приобретает вид:

$$S = \frac{T(1)}{T(n) + T_o} = \frac{T(1)}{fT(1) + \dfrac{(1-f)T(1)}{n} + T_o} = \frac{1}{f + \dfrac{1-f}{n} + \dfrac{T_o}{T(1)}}.$$

В этом случае предел ускорения при бесконечном увеличении количества процессоров определяется выражением:

$$\lim_{n\to\infty} S(n) = \frac{1}{f}.$$

Полученное выражение служит аргументом против вычислительных систем с очень большим числом процессоров, решающих общую задачу, поскольку в них издержки на обмен информацией между процессорами, реализующими параллельные части этой задачи, могут оказаться весьма существенными.

Закон Густафсона

Известную долю оптимизма в оценку, даваемую законом Амдала, вносят исследования, проведенные Джоном Густафсоном из NASA Ames Research [90]. Решая на вычислительной системе из 1024 процессоров три больших задачи, для которых доля последовательного кода f лежала в пределах от 0,4 до 0,8%, он получил значения ускорения по сравнению с однопроцессорным вариантом, равные соответственно 1021, 1020 и 1016. Согласно закону Амдала для данного числа процессоров и диапазона f, ускорение не должно было превысить величины порядка 201. Пытаясь объяснить это явление, Густафсон пришел к выводу, что причина кроется в исходной предпосылке, лежащей в основе закона Амдала: увеличение числа процессоров не сопровождается увеличением объема решаемой задачи. Реальное же поведение пользователей существенно отличается от такой гипотезы.

Обычно, получая в свое распоряжение более мощную систему, пользователь не стремится сократить время вычислений, а, сохраняя его практически неизменным, старается пропорционально мощности ВС увеличить объем решаемой задачи, например, чтобы повысить точность вычислений. И тут оказывается, что наращивание общего объема программы касается главным образом распараллеливаемой части программы. Это ведет к сокращению величины f. Примером может служить решение дифференциального уравнения в частных производных. Если доля последовательного кода составляет 10% для 1000 узловых точек, то для 100 000 точек доля последовательного кода снизится до 0,1%. Сказанное иллюстрирует рис. 10.6.

Таким образом, повышение ускорения обусловлено тем, что, оставаясь практически неизменной, последовательная часть в общем объеме увеличенной программы имеет уже меньший удельный вес. Чтобы оценить степень ускорения вычислений, когда объем последних увеличивается с ростом количества процессоров в системе (при постоянстве общего времени вычислений), Густафсон рекомендует использовать выражение, предложенное Е. Барсисом (Ed Barsis):

$$S(n) = f + (1 - f)n.$$

Данное выражение известно как закон масштабируемого ускорения или закон Густафсона (иногда его называют также законом Густафсона–Барсиса). Выражение констатирует, что ускорение является линейной функцией числа процессоров, если рабочая нагрузка масштабируется так, чтобы поддерживать неизменным время

вычислений. В заключение еще раз отметим, что закон Густафсона не противоречит закону Амдала. Различие состоит лишь в форме использования дополнительной мощности ВС, возникающей вследствие увеличения числа процессоров.

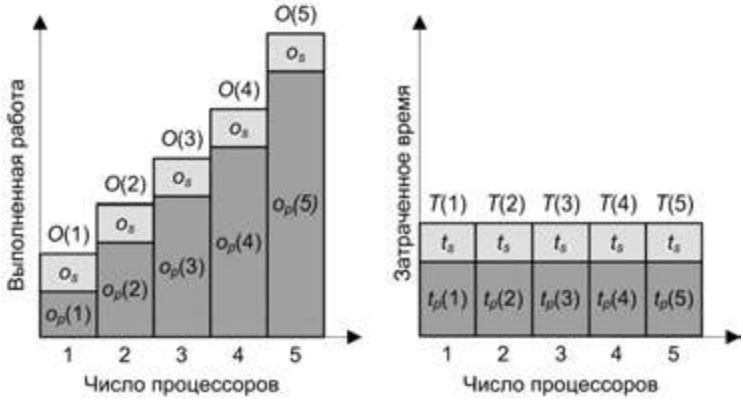


Рис. 10.6. Распределение рабочей нагрузки и времени вычислений в формулировке Густафсона

Закон Сана–Ная

В многопроцессорной параллельной ВС каждый процессор обычно имеет независимую локальную память сравнительно небольшой емкости. Общая память ВС образуется объединением локальной памяти каждого процессора ВС. При решении задача разделяется на подзадачи и распределяется по множеству процессоров. Подзадача размещается в локальной памяти процессора. Как и в постановке Густафсона, увеличение числа процессоров сопровождается возрастанием размера решаемой задачи, но до предела, обусловленного емкостью доступной памяти. Иными словами, объем задачи увеличивается так, чтобы каждая подзадача полностью занимала локальную память процессора. Такая постановка лежит в основе закона, сформулированного Ксиан-Хе Саном (Xian-He Sun) и Лайонелом Наем (Lionel M. Ni), и носит название закона ускорения, ограниченного памятью (рис. 10.7). На рисунке O^* обозначает рабочую нагрузку, ограниченную доступной памятью.

Пусть M — это емкость локальной памяти одного процессора. В этом случае суммарная память n-процессорной системы будет равна nM. В формулировке проблемы с ограничением, обусловленным памятью, предполагается, что память каждого процессора задействована полностью, а рабочая нагрузка на один процессор равна $O(1)$, где $O(1) = fO(1) + (1-f)O(1)$. Положим, что при использовании всех n процессоров распараллеливаемая часть задачи может масштабироваться в $G(n)$ раз. В этом случае масштабируемая рабочая нагрузка может быть описана выражением $O^* = fO + (1-f)G(n)O$. Здесь параметр $G(n)$ отражает возрастание рабо-

чей нагрузки с увеличением числа процессоров, а значит, и емкости памяти в n раз. В указанных рамках ускорение описывается выражением:

$$S(n) = \frac{f + (1-f)G(n)}{f + (1-f)\dfrac{G(n)}{n}}.$$

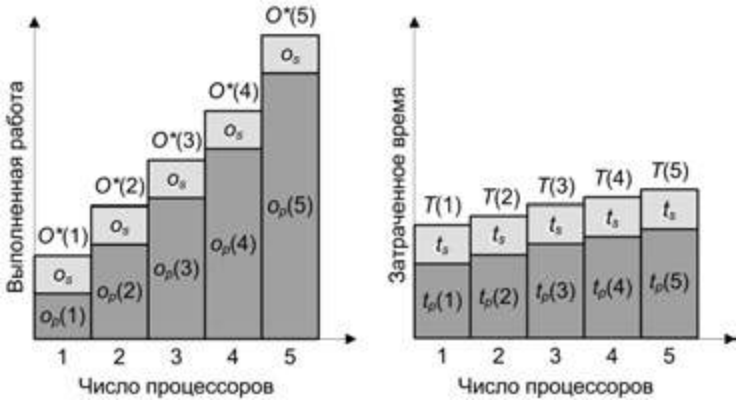


Рис. 10.7. Распределение рабочей нагрузки и времени вычислений в постановке Сана и Ная

Нетрудно показать, что полученное выражение представляет собой обобщение законов Амдала и Густафсона.

При $G(n) = 1$ размер задачи фиксирован, что соответствует постановке Амдала. В результате получаем формулу Амдала:

$$S(n) = \frac{f + (1-f)1}{f + (1-f)\dfrac{1}{n}} = \frac{1}{f + \dfrac{1-f}{n}}.$$

Вариант $G(n) = n$ соответствует случаю, когда с увеличением емкости памяти в n раз рабочая нагрузка также возрастает в n раз. Это идентично постановке Густафсона:

$$S(n) = \frac{f + (1-f)n}{f + (1-f)} = f + (1-f)n$$

В случае когда вычислительная нагрузка возрастает быстрее, чем требования к памяти ($G(n) > n$), модель с ограничением по памяти дает более оптимистичную оценку ускорения.

Рисунок 1.8 иллюстрирует оценки ускорения, получаемые по каждой из трех рассмотренных моделей ускорения.

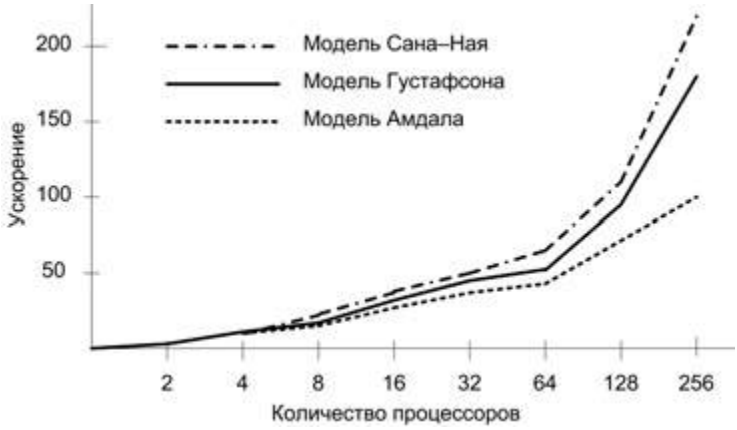


Рис. 10.8. Сравнение трех моделей ускорения

Метрика Карпа–Флэтта

В выражениях, представляющих законы Амдала, Густафсона и Сана–Ная, помимо числа процессоров, фигурирует доля последовательных вычислений f. Обычно она определяется путем анализа кода программы и выяснения того, какая по объему вычислений часть программы не может быть распараллелена. В то же время значения ускорения, получаемые на реальных системах, ниже, чем предсказывают соответствующие формулы. Вызвано это, главным образом, неучетом ранее упоминавшихся издержек на взаимодействие между параллельно работающими процессорами. Для оценки реальной возможности распараллеливания конкретного кода в параллельной ВС Алан Карп (Alan H. Karp) и Хорас Флэтт (Horace P. Flatt) предложили использовать эквивалент показателя f, известный как *метрика Карпа–Флэтта* и обозначаемый e. Этот показатель вычисляется исходя из экспериментально определенного ускорения на реальной ВС по следующей формуле:

$$e = \frac{\frac{1}{S(n)} - \frac{1}{n}}{1 - \frac{1}{n}}.$$

Чем меньше значение e, тем лучше может быть распараллелен код. Для задач фиксированного размера (как в постановке Амдала) эффективность параллельных вычислений с увеличением числа процессоров обычно уменьшается. С помощью показателя e, полученного на основании экспериментальных данных (метрики Карпа–Флэтта), можно оценить, чем именно обусловлено снижение эффективности — ограниченными возможностями распараллеливания или коммуникационными издержками параллельного вычисления (временем на обмен информации между параллельными ветвями).

Классификация параллельных вычислительных систем

Даже краткое перечисление типов современных параллельных вычислительных систем (ВС) дает понять, что для ориентации в этом многообразии необходима четкая система классификации. От ответа на главный вопрос — что заложить в основу классификации — зависит, насколько конкретная система классификации помогает разобраться с тем, что представляет собой архитектура ВС и насколько успешно данная архитектура позволяет решать определенный круг задач. Попытки систематизировать все множество архитектур параллельных вычислительных систем предпринимались достаточно давно и длятся по сей день, но к однозначным выводам пока не привели. Исчерпывающий обзор существующих систем классификации ВС приведен в [5].

Классификация Флинна

Среди всех рассматриваемых систем классификации ВС наибольшее признание получила классификация, предложенная в 1966 году М. Флинном [84, 85]. В ее основу положено понятие потока, под которым понимается последовательность элементов, команд или данных, обрабатываемая процессором. В зависимости от количества потоков команд и потоков данных Флинн выделяет четыре класса архитектур: SISD, MISD, SIMD, MIMD.

SISD

SISD (Single Instruction Stream/Single Data Stream) — одиночный поток команд и одиночный поток данных (рис. 10.9, *а*). Представителями этого класса являются, прежде всего, классические фон-неймановские ВМ, где имеется только один поток команд, команды обрабатываются последовательно и каждая команда инициирует одну операцию с одним потоком данных. То, что для увеличения скорости обработки команд и скорости выполнения арифметических операций может применяться конвейерная обработка, не имеет значения, поэтому в класс SISD одновременно попадают как ВМ CDC 6600 со скалярными функциональными устройствами, так и CDC 7600 с конвейерными. Некоторые специалисты считают, что к SISD-системам можно причислить и векторно-конвейерные ВС, если рассматривать вектор как неделимый элемент данных для соответствующей команды.

MISD

MISD (Multiple Instruction Stream/Single Data Stream) — множественный поток команд и одиночный поток данных (рис. 10.9, *б*). Из определения следует, что в архитектуре ВС присутствует множество процессоров, обрабатывающих один и тот же поток данных. Примером могла бы служить ВС, на процессоры которой подается искаженный сигнал, а каждый из процессоров обрабатывает этот сигнал с помощью своего алгоритма фильтрации. Тем не менее ни Флинн, ни другие специалисты в области архитектуры компьютеров до сих пор не сумели представить убедительный пример реально существующей вычислительной системы, построенной

на данном принципе. Ряд исследователей [92, 95, 156] относят к данному классу конвейерные системы, однако это не нашло окончательного признания. Отсюда принято считать, что пока данный класс пуст. Наличие пустого класса не следует считать недостатком классификации Флинна. Такие классы, по мнению некоторых исследователей [73, 137], могут стать чрезвычайно полезными для разработки принципиально новых концепций в теории и практике построения вычислительных систем.

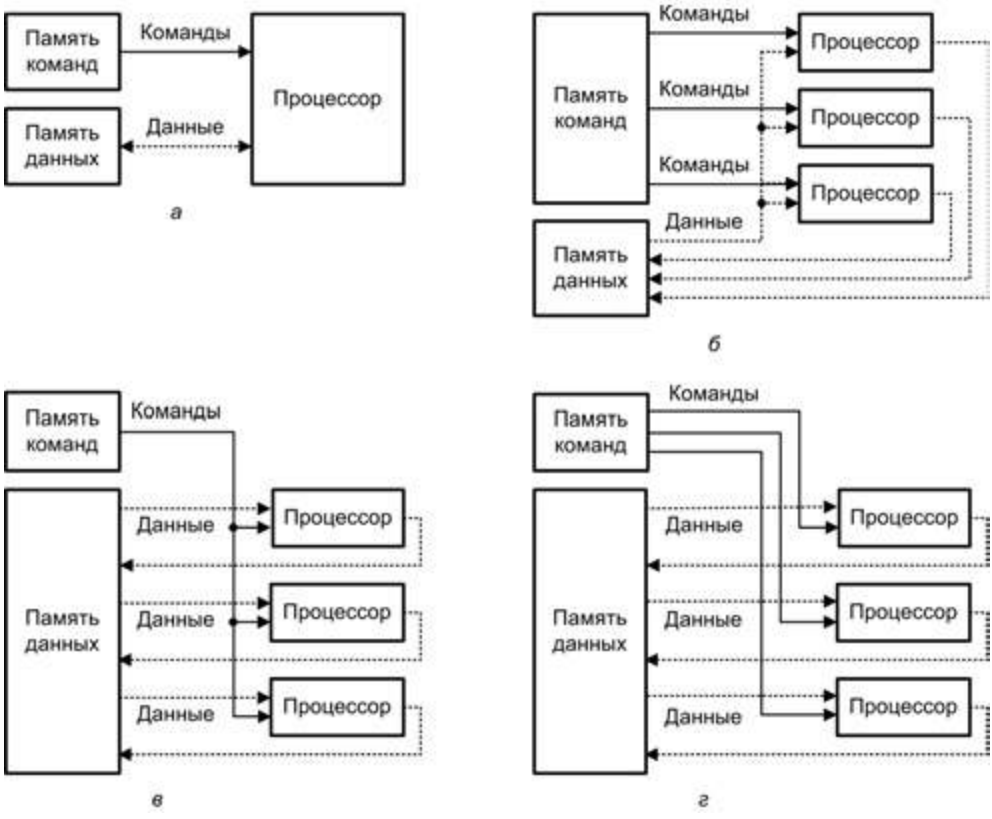


Рис. 10.9. Архитектура вычислительных систем по Флинну: *а* — SISD; *б* — MISD; *в* — SIMD; *г* — MIMD

SIMD

SIMD (Single Instruction Stream/Multiple Data Stream) — одиночный поток команд и множественный поток данных (рис. 10.9, *в*). ВМ данной архитектуры позволяют выполнять одну арифметическую операцию сразу над многими данными — элементами вектора. Бесспорными представителями класса SIMD считаются матрицы процессоров, где единое управляющее устройство контролирует множество процессорных элементов. Все процессорные элементы получают от устройства управления одинаковую команду и выполняют ее над своими локальными данными.

В принципе, в этот класс можно включить и векторно-конвейерные ВС, если каждый элемент вектора рассматривать как отдельный элемент потока данных.

MIMD

MIMD (Multiple Instruction Stream/Multiple Data Stream) — множественный поток команд и множественный поток данных (рис. 10.9, *г*). Класс предполагает наличие в вычислительной системе множества устройств обработки команд, объединенных в единый комплекс и работающих каждое со своим потоком команд и данных. Класс MIMD чрезвычайно широк, поскольку включает в себя всевозможные мультипроцессорные системы. Кроме того, приобщение к классу MIMD зависит от трактовки. Так, ранее упоминавшиеся векторно-конвейерные ВС можно вполне отнести и к классу MIMD, если конвейерную обработку рассматривать как выполнение множества команд (операций ступеней конвейера) над множественным скалярным потоком.

Схема классификации Флинна вплоть до настоящего времени является наиболее распространенной при первоначальной оценке той или иной ВС, поскольку позволяет сразу оценить базовый принцип работы системы, чего часто бывает достаточно. Однако у классификации Флинна имеются и очевидные недостатки, например неспособность однозначно отнести некоторые архитектуры к тому или иному классу. Другая слабость — это чрезмерная насыщенность класса MIMD. Все это породило множественные попытки либо модифицировать классификацию Флинна, либо предложить иную систему классификации.

Контрольные вопросы

1. Сравните схемы классификации параллелизма по уровню и гранулярности. Каковы, на ваш взгляд, достоинства, недостатки и области применения этих схем классификации?
2. В чем состоят основные идеи выделения четырех групп метрик параллельных вычислений?
3. Докажите интегральный характер метрики «качество».
4. Для заданной программы и конфигурации параллельной вычислительной системы рассчитайте значения метрик параллельных вычислений.
5. Поясните суть закона Амдала, приведите примеры, поясняющие его ограничения.
6. Какую проблему закона Амдала решает закон Густафсона? Как он это делает? Сформулируйте области применения этих двух законов.
7. Почему закон Сана–Ная называют обобщением законов Амдала и Густафсона? Докажите это утверждение.
8. Какую задачу решает метрика Карпа–Флэтта и каким именно образом?
9. Укажите достоинства и недостатки схемы классификации Флинна.

Глава 11

Память вычислительных систем

В вычислительных системах, объединяющих множество параллельно работающих процессоров или машин, задача правильной организации памяти является одной из важнейших. Различие между быстродействием процессора и памяти всегда было камнем преткновения в однопроцессорных ВМ. Многопроцессорность ВС приводит еще к одной проблеме — проблеме одновременного доступа к памяти со стороны нескольких процессоров.

В зависимости от того, каким образом организована память многопроцессорных (многомашинных) систем, различают вычислительные системы с разделяемой памятью (shared memory) и ВС с распределенной памятью (distributed memory).

В *системах с разделяемой памятью* (ее часто называют также совместно используемой или общей памятью) память ВС рассматривается как общий ресурс, и каждый из процессоров имеет полный доступ ко всему адресному пространству. Системы с разделяемой памятью называют *сильно связанными* (closely coupled systems). Подобное построение вычислительных систем имеет место как в классе SIMD, так и в классе MIMD. Иногда, чтобы подчеркнуть это обстоятельство, вводят специальные подклассы, используя для их обозначения аббревиатуры SM-SIMD (Shared Memory SIMD) и SM-MIMD (Shared Memory MIMD).

В варианте *с распределенной памятью* каждому из процессоров придается собственная память. Процессоры объединяются в сеть и могут при необходимости обмениваться данными, хранящимися в их памяти, передавая друг другу так называемые *сообщения*. Такой вид ВС называют *слабо связанными* (loosely coupled systems). Слабо связанные системы также встречаются как в классе SIMD, так и в классе MIMD, и иной раз, чтобы подчеркнуть данную особенность, вводят подклассы DM-SIMD (Distributed Memory SIMD) и DM-MIMD (Distributed Memory MIMD).

Иногда вычислительные системы с разделяемой памятью называют *мультипроцессорами*, а системы с распределенной памятью — *мультикомпьютерами*.

Различие между разделяемой и распределенной памятью заключается в структуре виртуальной памяти, то есть в том, как память выглядит со стороны процессора. Физически почти каждая система памяти разделена на автономные компоненты,

доступ к которым может производиться независимо. Разделяемую память от распределенной отличает то, каким образом подсистема памяти интерпретирует поступивший от процессора адрес ячейки. Для примера положим, что процессор выполняет команду load R0, i, означающую «Загрузить регистр R0 содержимым ячейки i». В случае разделяемой памяти i — это глобальный адрес, и для любого процессора указывает на одну и ту же ячейку. В распределенной системе памяти i — это локальный адрес. Если два процессора выполняют команду load R0, i, то каждый из них обращается к *i*-й ячейке в своей локальной памяти, то есть к разным ячейкам, и в регистры R0 могут быть загружены неодинаковые значения.

Различие между двумя системами памяти должно учитываться программистом, поскольку оно определяет способ взаимодействия частей распараллеленной программы. В варианте с разделяемой памятью достаточно создать в памяти структуру данных и передавать в параллельно используемые подпрограммы ссылки на эту структуру. В системе с распределенной памятью необходимо в каждой локальной памяти иметь копию совместно используемых данных. Эти копии создаются путем вкладывания разделяемых данных в сообщения, посылаемые другим процессорам.

Архитектура памяти вычислительных систем

В рамках как разделяемой, так и распределенной памяти реализуются несколько архитектур системы памяти.

Рис. 11.1. Классификация моделей архитектур памяти вычислительных систем

На рис. 11.1 приведена классификация таких моделей.

Физически разделяемая память

В вычислительных системах с физически разделяемой памятью все процессоры имеют равные возможности по доступу к единому адресному пространству — доступ любого процессора к памяти производится единообразно и занимает одинаковое время. Такие ВС называют *системами с однородным доступом к памяти* и обозначают аббревиатурой UMA (Uniform Memory Access). Это наиболее распространенная архитектура памяти параллельных ВС с разделяемой памятью [93].

Единая память может быть построена как одноблочная или по блочному принципу, но обычно практикуется второй вариант.

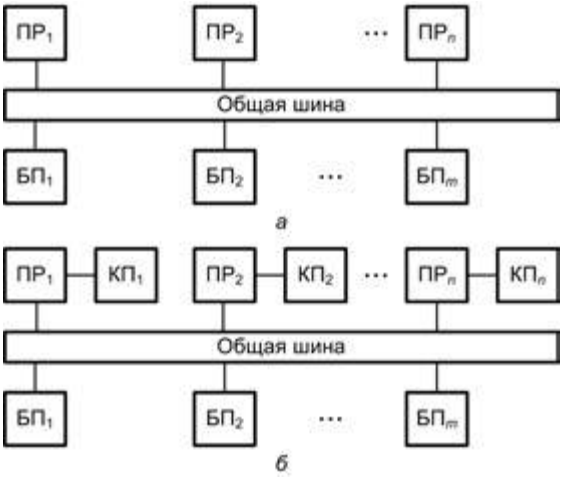


Рис. 11.2. Вычислительная система с однородным доступом к памяти (UMA): *а* — с процессорами без локальной кэш-памяти; *б* — с процессорами, оснащенными локальной кэш-памятью

Технически UMA-системы (рис. 11.2) предполагают наличие узла, соединяющего каждый из n процессоров с каждым из m банков памяти. Простейший путь построения таких ВС — объединение нескольких процессоров (ПР) с банками разделяемой памяти (БП) посредством общей шины (рис. 11.2, *а*). В этом случае, однако, в каждый момент времени обмен по шине может вести только один из процессоров, то есть процессоры должны соперничать за доступ к шине. Когда процессор ($ПР_i$) выбирает из памяти команду, остальные процессоры $ПР_j$ ($i \neq j$) должны ожидать, пока шина освободится. Если в систему входят только два процессора, они в состоянии работать с производительностью, близкой к максимальной, поскольку их доступ к шине можно чередовать: пока один процессор декодирует и выполняет команду, другой вправе использовать шину для выборки из памяти следующей команды. Однако когда добавляется третий процессор, производительность начинает падать. При наличии на шине десяти процессоров кривая быстродействия шины (рис. 11.3) становится горизонтальной, так что добавление 11-го процессора уже не дает повышения производительности. Нижняя кривая на этом рисунке иллюстрирует тот факт, что память и шина обладают фиксированной пропускной способностью, определяемой длительностью цикла памяти и протоколом шины. В многопроцессорной системе с общей шиной эта пропускная способность распределена между несколькими процессорами. Если длительность цикла процессора больше цикла памяти, к шине можно подключать много процессоров. Однако фактически процессор намного быстрее памяти, поэтому данная схема широкого применения не находит.

Ситуация может быть улучшена при наличии у каждого процессора локальной кэш-памяти ($КП_i$). Такая схема показана на рис. 11.2, *б*. Точка излома кривой производительности (верхняя кривая на рис. 11.3), в которой добавление процессоров

еще остается эффективным, теперь перемещается в область 20-ти процессоров, а точка, где кривая становится горизонтальной, — в область 30-ти процессоров.

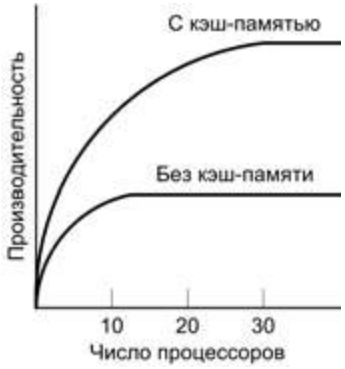


Рис. 11.3. Производительность UMA-системы на базе шины, как функция от числа процессоров на шине

Альтернативный способ построения многопроцессорной ВС с разделяемой памятью на основе UMA показан на рис. 11.4. Здесь шина заменена коммутатором, маршрутизирующим запросы процессора к одному из нескольких банков памяти. Несмотря на то что имеется несколько банков памяти, все они входят в единое адресное пространство. Преимущество такого подхода в том, что коммутатор в состоянии параллельно обслуживать несколько запросов. Каждый процессор может быть соединен со своим банком памяти и иметь доступ к нему на максимально допустимой скорости. Соперничество между процессорами может возникнуть при попытке одновременно обратиться к одному и тому же банку памяти. В этом случае доступ получает только один процессор, а прочие — блокируются.

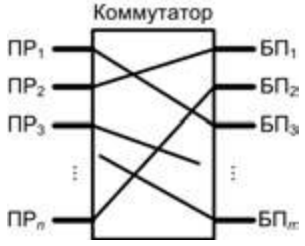


Рис. 11.4. UMA-система на базе коммутатора

Примерами ВС, где реализована модель UMA, могут служить суперЭВМ Cray T90, вычислительные системы Intel SHV, Sun E10000, IBM R60 и др.

К сожалению, архитектура UMA не очень хорошо масштабируется. Наиболее распространенные системы содержат 4–8 процессоров, значительно реже 32–64 процессора. Кроме того, подобные системы нельзя отнести к отказоустойчивым, так как отказ одного процессора или модуля памяти влечет отказ всей ВС.

При построении разделяемой памяти по блочному принципу следует учитывать одну особенность, свойственную именно многопроцессорным системам. Ранее отмечалось, что эффективность работы блочной памяти может быть повышена за счет расслоения памяти. В мультипроцессорах с физически разделяемой памятью все же используется поблочная адресация, когда ячейки со смежными адресами сосредоточены в одном банке, а не распределены между банками памяти по циклической схеме. Связано это с потерями времени на частое переключение процессоров с одного банка памяти на другой. В таких системах выгоднее соединить процессор с банком памяти и задействовать максимум находящейся в нем информации (прежде чем переключиться на другой банк).

Физически распределенная разделяемая память

Другим подходом к построению ВС с разделяемой памятью является физическое распределение памяти между процессорами. Каждый процессор имеет локальную память (ЛП), а совокупность ЛП всех процессоров образует общую память ВС. Здесь по-прежнему фигурирует единое адресное пространство и каждый из n процессоров имеет доступ к любой ячейке в пределах этого пространства. Варианты реализации этой идеи можно свести к трем группам, для обозначения которых применяют аббревиатуры NUMA, COMA и DSM.

Архитектура NUMA

Название архитектуры NUMA (Non-Uniform Memory Access) можно перевести как *неоднородный доступ к памяти*. В NUMA-системах каждый процессор имеет локальную память (ЛП), которая рассматривается как часть общей памяти ВС и которой выделен отдельный диапазон адресов в едином адресном пространстве. Помимо возможности обратиться к своей локальной памяти, каждый процессор имеет возможность физически «добраться» до локальной памяти остальных процессоров (удаленной памяти). Неоднородность обращения к памяти выражается в том, что доступ процессора к собственной локальной памяти производится напрямую (это намного быстрее, чем доступ к удаленной памяти через коммутатор или сеть). Обобщенная структура ВС с памятью типа NUMA показана на рис. 11.5.

Рис. 11.5. Вычислительная система с неоднородным доступом к памяти (NUMA)

Поскольку каждый процессор помимо локальной памяти имеет еще и локальную кэш-память (КП), в вычислительных системах с разделяемой памятью возникает проблема обеспечения непротиворечивости копий одного и того же блока основной памяти системы во всех локальных КП. Неоднозначность копий возникает,

если один из процессоров изменяет информацию в своей кэш-памяти. В этом случае аналогичные копии в кэш-памяти остальных процессоров должны быть либо соответствующим образом откорректированы, либо помечены как недостоверные. Эта проблема, известная как проблема *когерентности кэш-памяти*, детально будет рассмотрена позже. Именно способ решения этой проблемы обусловил то, что фактически концепция NUMA реализуется в одном из двух вариантов, обозначаемых аббревиатурами *ccNUMA* и *nccNUMA*.

Архитектура ccNUMA (Cache Coherent Non-Uniform Memory Architecture — *неоднородный доступ к памяти с обеспечением когерентности кэшей*) реализуется на базе шины. Аппаратные средства следят за процессами на шине, главным образом, относящимися к операциям записи в кэш-память. При попытке какого-то процессора модифицировать содержимое в одном из блоков своей кэш-памяти, эти аппаратные средства либо таким же образом обновляют содержимое аналогичных блоков в кэш-памяти других процессоров, либо помечают эти блоки как недостоверные. Примерами ВС с архитектурой системы памяти типа ccNUMA являются: SGI Origin 2000, Origin 3000, Cray T3E, HP Exemplar, IBM/Sequent NUMA-Q 2000.

Отличие архитектуры *nccNUMA* (Non-Cache Coherent Non-Uniform Memory Architecture — *неоднородный доступ к памяти без обеспечения когерентности кэшей*) от ccNUMA очевидно из названия. Архитектура nccNUMA не обеспечивает согласованности глобальных данных на аппаратном уровне. «Забота» об использовании таких данных полностью возлагается на программное обеспечение (приложения или компиляторы). Несмотря на этот недостаток, архитектура значительно упрощает масштабирование ВС, то есть увеличение числа процессоров. Примерами реализации концепции nccNUMA могут служить суперЭВМ Cray T3D и система IBM SP3.

Архитектура COMA

В архитектуре COMA (Cache Only Memory Architecture), что можно перевести как *архитектура только с кэш-памятью*, локальная память каждого процессора считается частью большой кэш-памяти [39]. Совокупность кэш-памяти всех процессоров рассматривается как глобальная память системы, причем собственно глобальная память отсутствует. Структура системы памяти с архитектурой COMA показана на рис. 11.6.

Рис. 11.6. Вычислительная система с архитектурой COMA

Принципиальная особенность концепции COMA выражается в динамике. Здесь данные не привязаны статически к определенному модулю памяти и не имеют

уникального адреса, остающегося неизменным в течение всего времени существования переменной. В архитектуре COMA данные переносятся в кэш-память того процессора, который последним их запросил, при этом расположение переменной в памяти не фиксировано уникальным адресом, и она в произвольный момент времени может размещаться в любой физической ячейке. Перенос данных из одной локальной кэш-памяти в другую не требует участия операционной системы, но подразумевает сложную и дорогостоящую аппаратуру управления памятью. Для организации такого режима используют так называемые *справочники* (СПР), хранящие информацию о текущем расположении всех переменных. Отметим также, что последняя копия элемента данных никогда из кэш-памяти не удаляется.

Поскольку в архитектуре COMA данные перемещаются в локальную кэш-память процессора, запросившего их последним, по производительности ВС с COMA-памятью существенно превосходят ВС с NUMA-памятью. С другой стороны, если единственная переменная или две различные переменные, хранящиеся в одном блоке одной и той же кэш-памяти, требуются двум процессорам, этот блок должен перемещаться между процессорами туда и обратно при каждом доступе к данным. Такие эффекты могут зависеть от деталей распределения памяти и приводить к непредсказуемым ситуациям. Архитектура COMA реализована в целом ряде ВС, в частности в системах KSR-1 и DDM.

Архитектура DSM

Интересный вариант системы с физически распределенной разделяемой памятью известен как DSM (Distribute Shared Memory — *распределенная разделяемая память*). Каждый процессор обладает локальной памятью, но в отличие от NUMA не имеет возможности физического доступа к локальной памяти других процессоров. Идея архитектуры заключается в том, что ВС представляется пользователю как система с общей памятью лишь благодаря операционной системе. Это означает, что операционная система предлагает пользователю единое адресное пространство, однако фактическое обращение к памяти «чужого» компьютера обеспечивается путем обмена сообщениями между процессорами. Невозможность физического доступа к удаленной памяти отличает данную архитектуру от nccNUMA. Поэтому используемое иногда другое название рассматриваемой архитектуры — *scNUMA* (Software-Coherent Non-Uniform Memory Architecture — *архитектура с неоднородным доступом к памяти и программным обеспечением когерентности*) представляется не вполне оправданным. DSM-подход находит применение в ВС класса MIMD, например в системах IBM SP2, DEC TruCluster, Intel TFLOPS.

Распределенная память

В системе с распределенной памятью каждый процессор обладает собственной памятью и способен адресовать только ее. Некоторые авторы называют этот тип систем *многомашинными ВС* или *мультикомпьютерами*, подчеркивая тот факт, что блоки, из которых строится система, сами по себе являются небольшими вычислительными системами с процессором и памятью. Модели архитектур с распределенной памятью принято обозначать как *архитектуры без прямого доступа к удаленной*

памяти (NORMA, No Remote Memory Access). Такое название подчеркивает, что каждый процессор имеет доступ только к своей локальной памяти. Доступ к удаленной памяти (локальной памяти другого процессора) возможен только путем обмена сообщениями с процессором, которому принадлежит адресуемая память.

Подобная организация имеет ряд достоинств. Во-первых, при доступе к данным не возникает конкуренции за шину или коммутаторы — каждый процессор может полностью использовать полосу пропускания тракта связи с собственной локальной памятью. Во-вторых, размер системы ограничивает только сеть, объединяющая процессоры. В-третьих, снимается проблема когерентности кэш-памяти. Каждый процессор вправе самостоятельно менять свои данные, не заботясь о согласовании копий данных в собственной локальной кэш-памяти с кэш-памятью других процессоров.

Основной недостаток ВС с распределенной памятью заключается в сложности обмена информацией между процессорами. Если какой-то из процессоров нуждается в данных из памяти другого процессора, он должен обменяться с этим процессором сообщениями. Это приводит к двум видам издержек:

- требуется время на формирование и пересылку сообщения от одного процессора к другому;
- для обеспечения реакции на сообщения от других процессоров принимающий процессор должен получить запрос прерывания и выполнить процедуру обработки этого прерывания.

Структура системы с распределенной памятью приведена на рис. 11.7. По отношению к каждому процессору все остальные процессоры можно рассматривать просто как периферийные устройства. Для посылки сообщения процессор формирует блок данных в своей локальной памяти и извещает свой локальный контроллер о необходимости передачи информации на другой процессор. По коммуникационной сети это сообщение пересылается на приемный контроллер принимающего процессора. Последний находит место для сообщения в собственной локальной памяти и уведомляет процессор-источник о получении сообщения.

Рис. 11.7. Вычислительная система с распределенной памятью

Мультипроцессорная когерентность кэш-памяти

Мультипроцессорная система с разделяемой памятью состоит из двух или более независимых процессоров, каждый из которых выполняет либо часть большой программы, либо независимую программу. Все процессоры обращаются к командам

и данным, хранящимся в разделяемой основной памяти. Поскольку память является общим ресурсом, при обращении к ней между процессорами возникает соперничество, в результате чего средняя задержка на доступ к памяти увеличивается. Для сокращения такой задержки каждому процессору придается локальная кэш-память, которая, обслуживая локальные обращения к памяти, во многих случаях предотвращает необходимость доступа к разделяемой основной памяти. В свою очередь, оснащение каждого процессора локальной кэш-памятью приводит к так называемой *проблеме когерентности* или обеспечения *согласованности кэш-памяти*. Согласно [79, 158], система является когерентной, если каждая операция чтения по какому-либо адресу, выполненная любым из процессоров, возвращает значение, занесенное в ходе последней операции записи по этому адресу, вне зависимости от того, какой из процессоров производил запись последним.

В простейшей форме проблему когерентности кэш-памяти можно пояснить следующим образом (рис 11.8). Пусть два процессора $ПР_1$ и $ПР_2$ связаны с общей памятью посредством шины. Сначала оба процессора читают переменную *x*. Копии блоков, содержащих эту переменную, пересылаются из основной памяти в локальные кэши обоих процессоров $КП_1$ и $КП_2$ (рис. 11.8, *а*). Далее процессор $ПР_1$ выполняет операцию увеличения значения переменной *x* на единицу. Так как копия переменной уже находится в кэш-памяти данного процессора, произойдет кэш-попадание и значение *x* будет изменено только в $КП_1$. Если теперь процессор $ПР_2$ вновь выполнит операцию чтения *x*, то также произойдет кэш-попадание и $ПР_2$ получит хранящееся в его кэш-памяти «старое» значение *x* (рис. 11.8, *б*).

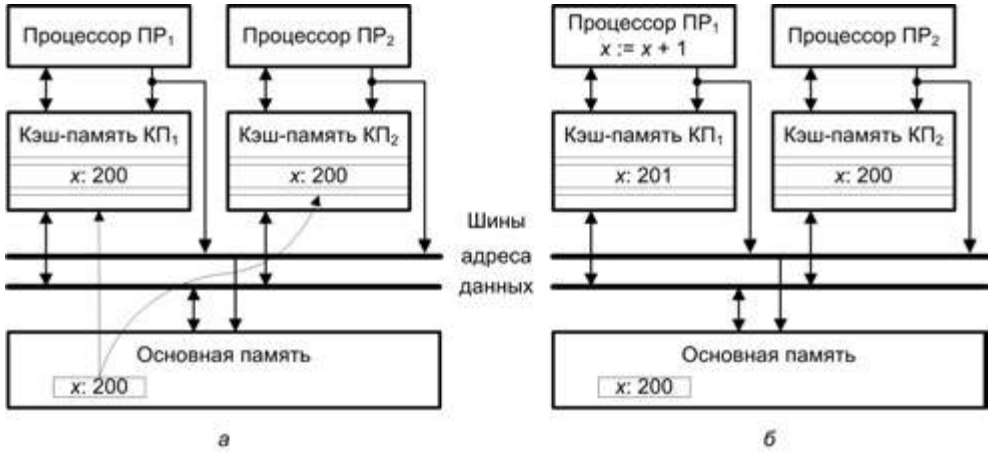


Рис. 11.8. Иллюстрация проблемы когерентности памяти: содержимое памяти: *а* — до изменения значения x; *б* — после изменения

Поддержание согласованности требует, чтобы при изменении элемента данных одним из процессоров соответствующие изменения были проведены в кэш-памяти остальных процессоров, где есть копия измененного элемента данных, а также в общей памяти. Схожая проблема возникает, кстати, и в однопроцессорных системах,

где присутствует несколько уровней кэш-памяти. Здесь требуется согласовать содержимое кэшей разных уровней.

В решении проблемы когерентности выделяются два подхода: программный и аппаратный. В некоторых системах применяют стратегии, совмещающие оба подхода.

Программные способы решения проблемы когерентности

Программные приемы решения проблемы когерентности позволяют обойтись без дополнительного оборудования или свести его к минимуму [65]. Задача возлагается на компилятор и операционную систему. Привлекательность такого подхода в возможности устранения некогерентности еще до этапа выполнения программы, однако принятые компилятором решения могут в целом отрицательно сказаться на эффективности кэш-памяти.

Компилятор анализирует программный код, определяет те совместно используемые данные, которые могут стать причиной некогерентности, и помечает их. В процессе выполнения программы операционная система или соответствующая аппаратура предотвращают кэширование (занесение в кэш-память) помеченных данных, и в дальнейшем для доступа к ним, как при чтении, так и при записи, приходится обращаться к «медленной» основной памяти. Учитывая, что некогерентность возникает только в результате операций записи, происходящих значительно реже, чем чтение, рассмотренный прием следует признать недостаточно удачным.

Более эффективными представляются способы, где в ходе анализа программы определяются безопасные периоды использования общих переменных и так называемые критические периоды, где может проявиться некогерентность. Затем компилятор вставляет в генерируемый код команды, позволяющие обеспечить когерентность всей кэш-памяти именно в такие критические периоды.

Аппаратные способы решения проблемы когерентности

Большинство из предложенных способов борьбы с некогерентностью ориентированы на динамическое (в процессе вычислений) распознавание и устранение несогласованности копий совместно используемых данных с помощью специальной аппаратуры. Аппаратные методы обеспечивают более высокую производительность, поскольку издержки, связанные с некогерентностью, имеют место только при возникновении ситуации некогерентности. Кроме того, этот подход прозрачен для программиста и пользователя [158]. Аппаратные механизмы преодоления проблемы когерентности принято называть *протоколами когерентности кэш-памяти*.

Как известно, для обеспечения идентичности копий данных в кэше и основной памяти в однопроцессорных системах применяется одна из двух стратегий: *сквозная запись* (write through) или *обратная запись* (write back). При сквозной записи новая информация одновременно заносится как в кэш, так и в основную память. При обратной записи все изменения производятся только в кэш-памяти, а обновление содержимого основной памяти происходит лишь при удалении блока из кэш-памяти путем пересылки его в соответствующее место основной памяти. В случае

мультипроцессорной системы, когда копии совместно используемых данных могут находиться сразу в нескольких кэшах, необходимо обеспечить когерентность всех копий. Ни сквозная, ни обратная запись не предусматривают такой ситуации, и для ее разрешения опираются на другие приемы, а именно: *запись с аннулированием* (write invalidate) и *запись с обновлением* (write update). Последняя известна также под названием *записи с трансляцией* (write broadcast).

В варианте записи с аннулированием, если какой-либо процессор производит изменения в одном из блоков своей кэш-памяти, все имеющиеся копии этого блока в других локальных кэшах аннулируются, то есть помечаются как недостоверные. Для этого бит достоверности измененного блока во всех прочих кэшах устанавливается в 0. Идею записи с аннулированием иллюстрирует рис. 11.9, где показано исходное состояние системы памяти, когда копия переменной x имеется во всех кэшах (рис. 11.9, *а*), а также ее состояние после записи нового значения x в $КП_2$ (рис 11.9, *б*).

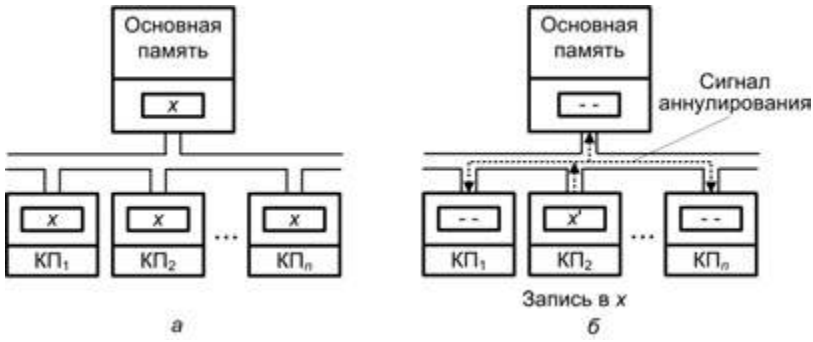


Рис. 11.9. Запись с аннулированием: *а* — исходное состояние; *б* — после изменения значения x в кэш-памяти $КП_2$

Если впоследствии другой процессор попытается прочитать данные из своей копии такого блока, произойдет кэш-промах. Следствием кэш-промаха должно быть занесение в локальную кэш-память читающего процессора корректной копии блока. Некоторые схемы когерентности позволяют получить корректную копию непосредственно из той локальной кэш-памяти, где блок подвергся модификации. Если такая возможность отсутствует, новая копия берется из основной памяти. В случае сквозной записи это может быть сделано сразу же, а при использовании обратной записи модифицированный блок предварительно должен быть переписан в основную память.

Запись с обновлением предполагает, что любая запись в локальный кэш немедленно дублируется и во всех остальных кэшах, содержащих копию измененного блока (немедленное обновление блока в основной памяти не является обязательным). Этот случай иллюстрирует рис. 11.10.

Стратегия записи с обновлением требует широковещательной передачи новых данных по коммуникационной сети, что осуществимо не при любой конфигурации последней.

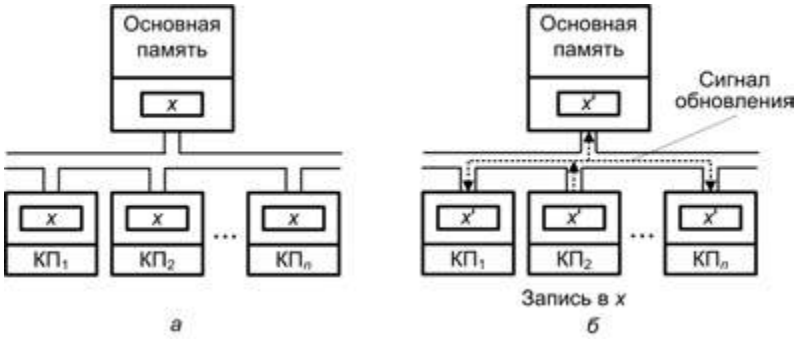


Рис. 11.10. Запись с обновлением: *а* — исходное состояние; *б* — после изменения значения *x* в кэш-памяти $КП_2$

В общем случае для поддержания когерентности в мультипроцессорных системах имеются следующие возможности:

- разделяемая кэш-память;
- некэшируемые данные;
- широковещательная запись;
- протоколы наблюдения;
- протоколы на основе справочника.

Разделяемая кэш-память. Первое и наиболее простое решение — вообще отказаться от локальных кэшей, и все обращения к памяти адресовать к одной общей кэш-памяти, связанной со всеми процессорами посредством какой-либо коммуникационной сети. Хотя данный прием обеспечивает когерентность копий данных и прозрачен для пользователя, количество конфликтов по доступу к памяти он не снижает, поскольку возможно одновременное обращение нескольких процессоров к одним и тем же данным в общей кэш-памяти. Кроме того, наличие разделяемой кэш-памяти нарушает важнейшее условие высокой производительности, согласно которому процессор и кэш-память должны располагаться как можно ближе друг к другу. Положение осложняется и тем, что каждый доступ к кэш-памяти связан с обращением к арбитру, который определяет, какой из процессоров получит доступ к кэш-памяти. Тем не менее общая задержка обращения к памяти в целом уменьшается.

Некэшируемые данные. Проблема когерентности имеет отношение к тем данным, которые в ходе выполнения программы могут быть изменены. Одно из вероятных решений — это запрет кэширования таких данных. Технически запрет на кэширование отдельных байтов и слов достаточно трудно реализуем. Несколько проще сделать некэшируемым определенный блок данных. При обращении процессора к такому блоку складывается ситуация кэш-промаха, производится доступ к основной памяти, но копия блока в кэш-память не заносится. Для реализации подобного приема каждому блоку в основной памяти должен быть придан признак, указывающий, является ли блок кэшируемым или нет.

Если кэш-система состоит из раздельных кэшей команд и данных, сказанное относится главным образом к кэш-памяти данных, поскольку современные подходы

к программированию не рекомендуют модификацию команд программы. Следовательно, по отношению к информации в кэше команд применяется только операция чтения, что не влечет проблемы когерентности.

В отношении того, какие данные не должны кэшироваться, имеется несколько подходов.

В первом варианте запрещается занесение в кэш лишь той части совместно используемых данных, которая служит для управления критическими секциями программы, то есть теми частями программы, где процессоры могут изменять разделяемые ими данные. Принятие решения о том, какие данные могут кэшироваться, а какие — нет, возлагается на программиста, что делает этот способ непрозрачным для пользователя.

Во втором случае накладывается запрет на кэширование всех совместно используемых данных, которые в процессе выполнения программы могут быть изменены. Естественно, что для доступа к таким данным приходится обращаться к медленной основной памяти и производительность процессора падает. На первый взгляд, в варианте, где запрещается кэширование только управляющей информации, производительность процессора будет выше, однако, прежде чем сделать такой вывод, нужно учесть одно обстоятельство. Дело в том, что для сохранения согласованности данных, модифицируемых процессором в ходе выполнения критической секции программы, блоки с копиями этих данных в кэш-памяти при выходе из критической секции нужно аннулировать. Данная операция носит название *очистки кэш-памяти* (cache flush). Очистка необходима для того, чтобы к моменту очередного входа в критическую секцию в кэш-памяти не осталось «устаревших» данных. Регулярная очистка кэша при каждом выходе из критической секции снижает производительность процессора за счет увеличения времени, нужного для восстановления копий в кэш-памяти. Ситуацию можно несколько улучшить, если вместо очистки всей кэш-памяти помечать те блоки, к которым при выполнении критической секции было обращение, тогда при покидании критической секции достаточно очищать только эти помеченные блоки.

Широковещательная запись. При широковещательной записи каждый запрос на запись в конкретную кэш-память направляется также и всем остальным кэшам системы. Это заставляет контроллеры кэшей проверить, нет ли там копии изменяемого блока. Если такая копия найдена, то она аннулируется или обновляется, в зависимости от применяемой схемы. Метод широковещательной записи связан с дополнительными групповыми операциями с памятью (транзакциями), поэтому он реализован лишь в больших вычислительных системах.

На двух последних возможностях поддержания когерентности в мультипроцессорных системах остановимся более подробно.

Протоколы наблюдения

В *протоколах наблюдения* (snoopy protocols или просто snooping) ответственность за поддержание когерентности всех кэшей многопроцессорной системы возлагается на контроллеры кэшей. В системах, где реализованы протоколы наблюдения, контроллер каждой локальной кэш-памяти содержит *блок слежения за шиной*

(рис. 11.11), который следит за всеми транзакциями на общей шине и, в частности, контролирует все операции записи. Процессоры должны широковещательно передавать на шину любые запросы на доступ к памяти, потенциально способные изменить состояние когерентности совместно используемых блоков данных. Локальный контроллер кэш-памяти каждого процессора затем определяет, присутствует ли в его кэш-памяти копия модифицируемого блока, и если это так, то такой блок аннулируется или обновляется.

Протоколы наблюдения характерны для мультипроцессорных систем на базе шины, поскольку общая шина достаточно просто обеспечивает как наблюдение, так и широковещательную передачу сообщений. Однако здесь необходимо принимать меры, чтобы повышенная нагрузка на шину, связанная с наблюдением и трансляцией сообщений, не «съела» преимуществ локальных кэшей.

Рис. 11.11. Структура системы, обеспечивающей когерентность кэшей с помощью протокола наблюдения

Ниже рассматриваются некоторые из наиболее распространенных протоколов наблюдения. Большинство из них описываются упрощенно, а их детальное изложение можно найти по ссылкам на литературные источники.

В большинстве протоколов стратегия обеспечения когерентности кэш-памяти расценивается как смена состояний в конечном автомате. При таком подходе предполагается, что любой блок в локальной кэш-памяти может находиться в одном из фиксированных состояний. Обычно число таких состояний не превышает четырех, поэтому в теге каждого блока кэш-памяти имеются два бита, называемые *битами состояния* (SB, Status Bit). Следует также учитывать, что некоторым идентичным по смыслу состояниям блока кэш-памяти разработчиками различных протоколов присвоены разные наименования. Например, состояние блока, в котором были произведены локальные изменения, в одних протоколах называют *Dirty* («грязный»), а в других — *Modified* («модифицированный» или «измененный»).

Протокол сквозной записи. Этот протокол представляет собой расширение стандартной процедуры сквозной записи, известной по однопроцессорным системам. В нем запись в кэш-память любого процессора сопровождается записью в основную память. В дополнение, все остальные кэши, содержащие копию измененного блока, должны объявить свою копию недействительной. Протокол считается наиболее простым, но при большом числе процессоров приводит к значительному трафику шины, поскольку требует повторной перезагрузки измененного блока в те кэши, где этот блок ранее был объявлен недействительным [155].

Протокол обратной записи. В основе протокола лежит стандартная схема обратной записи, за исключением того, что расширено условие перезаписи блока в основную память. Так, если копия блока данных в одном из локальных кэшей подверглась модификации, этот блок будет переписан в основную память при выполнении одного из двух условий:

- блок удаляется из той кэш-памяти, где он был изменен;
- другой процессор обратился к своей копии измененного блока.

Если содержимое блока в локальной кэш-памяти не модифицировалось, перезапись в основную память не производится. Доказано, что такой протокол по эффективности превосходит схему сквозной записи, поскольку необходимо переписывать только измененные блоки [155].

Протокол однократной записи. Протокол однократной записи (write-once), предложенный Гудменом [88], — первый из упоминающихся в публикациях протоколов обеспечения когерентности кэш-памяти. Он относится к схемам на основе наблюдения, действующим на принципе записи с аннулированием. Протокол предполагает, что первая запись в любой блок кэш-памяти производится по схеме сквозной записи, при этом контроллеры других кэшей объявляют свои копии измененного блока недействительными. С этого момента только процессор, произведший запись, обладает достоверной копией данных [103]. Последующие операции записи в рассматриваемый блок выполняются в соответствии с протоколом обратной записи [46].

Два бита состояния в теге каждого блока кэш-памяти позволяют представить четыре состояния, в которых может находиться блок: «недействительный» (I, Invalid), «достоверный» (V, Valid), «резервированный» (R, Reserved) и «измененный» (D, Dirty). В состоянии I блок кэш-памяти не содержит достоверных данных. В состоянии V блок кэша содержит данные, считанные из основной памяти и к данному моменту еще не измененные, то есть содержимое блоков кэш-памяти и основной памяти согласовано. Состояние R означает, что с момента считывания из основной памяти в блоке локальной кэш-памяти было произведено только одно изменение, причем оно учтено и в основной памяти. В состоянии R содержимое блока кэша и основной памяти также согласовано. Наконец, статус D показывает, что блок кэш-памяти модифицировался более одного раза и последние изменения еще не переписаны в основную память. В этом случае блок кэша и содержимое основной памяти не согласованы.

Диаграмма состояний протокола показана на рис. 11.12.

Протокол Synapse. Данный протокол, реализованный в отказоустойчивой мультипроцессорной системе Synapse $N + 1$, представляет собой версию протокола однократной записи, где вместо статуса R используется статус D. Кроме того, переход из состояния D в состояние V при промахе, возникшем в ходе чтения данных другим процессором, заменен достаточно громоздкой последовательностью. Связано это с тем, что при первом кэш-промахе чтения запросивший процессор не может получить достоверную копию непосредственно из той локальной кэш-памяти, где произошло изменение данных, и вынужден обратиться напрямую к основной памяти [46, 102].

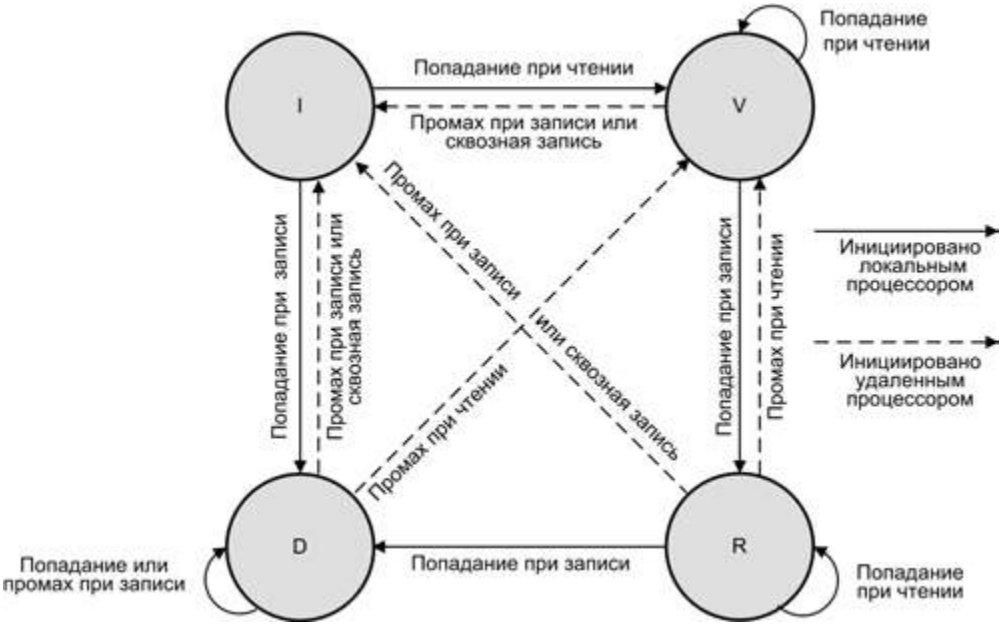


Рис. 11.12. Протокол однократной записи

Протокол Berkeley. Протокол Berkeley [103] был применен в мультипроцессорной системе Berkeley, построенной на базе RISC-процессоров.

Снижение издержек, возникающих в результате кэш-промахов, обеспечивается благодаря реализованной в этом протоколе идее прав владения на блок кэша. Обычно владельцем прав на все блоки данных считается основная память. Прежде чем модифицировать содержимое блока в своей кэш-памяти, процессор должен получить права владения на данный блок. Эти права приобретаются с помощью специальных операций чтения и записи. Если при доступе к блоку, собственником которого в данный момент не является основная память, происходит кэш-промах, процессор, являющийся владельцем блока, предотвращает чтение из основной памяти и сам снабжает запросивший процессор данными из своей локальной кэш-памяти.

Другое улучшение — введение состояния совместного использования (shared). Производя запись в один из блоков своей локальной кэш-памяти, процессор обычно формирует сигнал аннулирования копий изменяемого блока в других кэшах. В протоколе Berkeley сигнал аннулирования формируется только при условии, что в прочих кэшах имеются такие копии. Это позволяет существенно снизить непроизводительный трафик шины.

Диаграмма состояний протокола Berkeley показана на рис. 11.13.

Сравнивая протоколы однократной записи и Berkeley, можно отметить следующее. Оба протокола используют стратегию обратной записи, при которой измененные блоки удерживаются в кэш-памяти как можно дольше. Основная память обновляется только при удалении блока из кэша. Однако протокол Berkeley пересылает

строки непосредственно между кэшами, в то время как протокол однократной записи передает блок из исходного кэша в основную память, а затем из ОП в запросившие кэши, что имеет следствием общую задержку системы памяти [103].

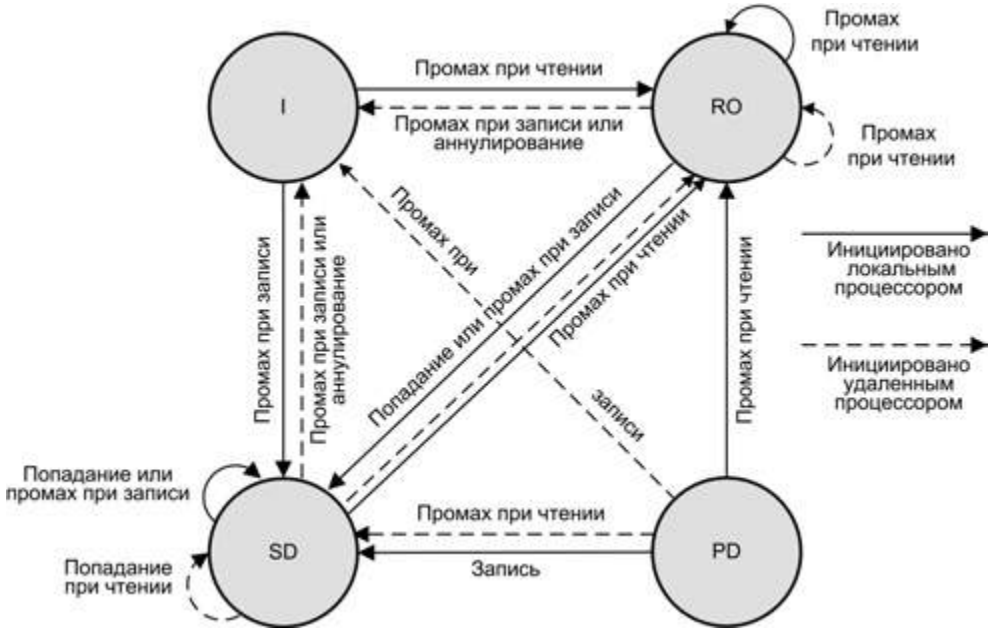


Рис. 11.13. Протокол Berkeley

Протокол Firefly. Протокол был предложен Такером и др. [155] и реализован в мультипроцессорной системе Firefly Multiprocessor Workstation, разработанной в исследовательском центре Digital Equipment Corporation.

В протоколе Firefly [117], диаграмма переходов для которого приведена на рис. 11.14, используется запись с обновлением. Стратегия обратной записи применяется только к тем блокам, которые находятся в состоянии PD или E, в то время как применительно к блокам в состоянии S действует сквозная запись.

Протокол имеет преимущества перед ранее описанными в том, что стратегия сквозной записи привлекается лишь при логической необходимости.

Протокол Dragon. Протокол применен в мультипроцессорной системе Xerox Dragon и представляет собой независимую версию протокола Firefly.

В протоколе реализована процедура записи с обновлением. Строка кэша может иметь одно из пяти состояний [117]:

- *Invalid* (I) — копия, хранящаяся в кэше, недействительна;
- *Read Private* (RP) — существует лишь одна копия блока, и она совпадает с содержимым основной памяти;
- *Private Dirty* (PD) — существует лишь одна копия блока, и она не совпадает с содержимым основной памяти;

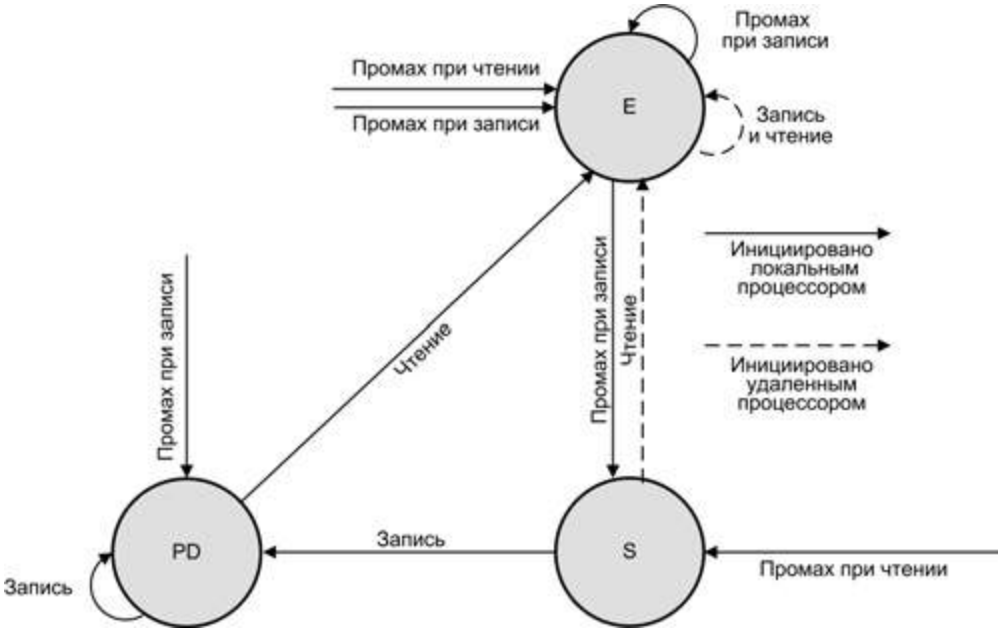


Рис. 11.14. Протокол Firefly

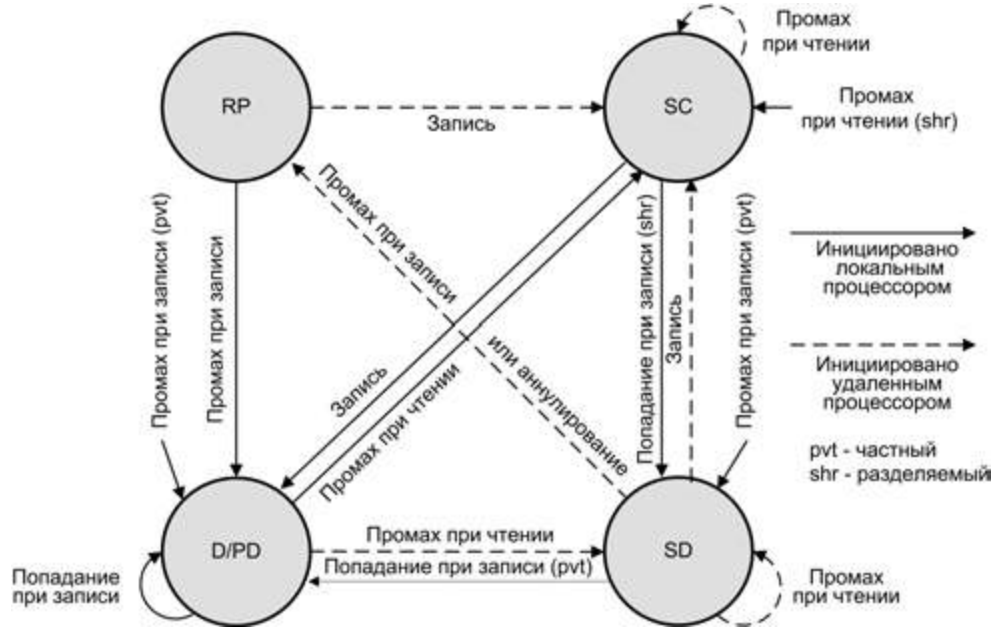


Рис. 11.15. Протокол Dragon

- *Shared Clean* (SC) — имеется несколько копий блока, и все они идентичны содержимому основной памяти;
- *Shared Dirty* (SD) — имеется несколько копий блока, не совпадающих с содержимым основной памяти.

Дополнительное состояние SD предназначено для предотвращения записи в основную память. Диаграмма состояний для данного протокола приведена на рис. 11.15.

Протокол MESI. Безусловно, среди известных протоколов наблюдения самым популярным является протокол MESI (Modified/Exclusive/Shared/Invalid). Протокол MESI широко распространен в коммерческих микропроцессорных системах, например на базе микропроцессоров Pentium и PowerPC.

Протокол был разработан для кэш-памяти с обратной записью. Одной из основных задач протокола MESI является откладывание на максимальный срок операции обратной записи кэшированных данных в основную память ВС. Это позволяет улучшить производительность системы за счет минимизации ненужных пересылок информации между кэшами и основной памятью.

Протокол MESI приписывает каждому блоку кэш-памяти одно из четырех состояний, которые контролируются двумя битами состояния MESI в теге данного блока. Статус кэш-блока может быть изменен как процессором, для которого эта кэш-память является локальной, так и другими процессорами мультипроцессорной системы. Управление состоянием кэш-блоков может быть возложено и на внешние логические устройства. Одна из версий протокола предусматривает использование ранее рассмотренной схемы однократной записи.

Согласно протоколу MESI, каждый блок бывает в одном из четырех возможных состояний (в дальнейшем будем ссылаться на эти состояния с помощью букв M, E, S и I):

- *Модифицированный* (M, Modified) — данные в блоке кэш-памяти, помеченном как M, были модифицированы, но измененная информация пока не переписана в основную память. Это означает, что информация, содержащаяся в рассматриваемом блоке, достоверна только в данном кэше, а в основной памяти и остальных кэшах — недостоверна.
- *Эксклюзивный* (E, Exclusive) — данный блок в кэш-памяти не подвергался изменению посредством запроса на запись, совпадает с аналогичным блоком в основной памяти, но отсутствует в любом другом локальном кэше. Иными словами, информация достоверна в этой локальной кэш-памяти и недостоверна в любой другой.
- *Разделяемый* (S, Shared) — блок в кэш-памяти совпадает с аналогичным блоком в основной памяти (данные достоверны) и может присутствовать в одном или нескольких из прочих кэшей.
- *Недействительный* (I, Invalid) — блок кэша, помеченный как недействительный, не содержит достоверных данных и становится логически недоступным.

Порядок перехода блока кэш-памяти из одного состояния в другое зависит от: текущего статуса блока, выполняемой операции (чтение или запись), результата обращения в кэш (попадание или промах) и, наконец, от того, является ли блок

разделяемым или нет. На рис. 11.16 приведена диаграмма переходов без учета режима однократной записи.

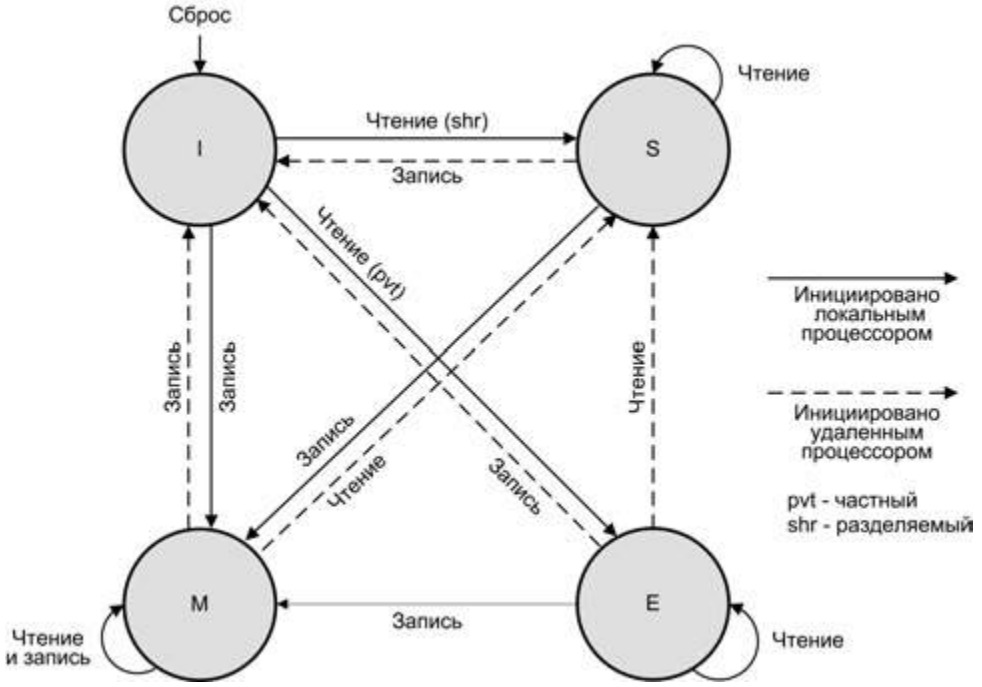


Рис. 11.16. Протокол MESI — диаграмма переходов без учета однократной записи

Предположим, что один из процессоров делает запрос на чтение из блока, которого в текущий момент нет в локальной кэш-памяти (промах при чтении). Запрос будет широковещательно передан по шине. Если ни в одном из кэшей не нашлось копии нужного блока, то ответной реакции от контроллеров наблюдения других процессоров не последует, блок будет считан в кэш запросившего процессора из основной памяти, а копии будет присвоен статус E. Если в каком-либо из локальных кэшей имеется искомая копия, от соответствующего контроллера слежения поступит отклик, означающий доступ к разделяемому блоку. Все копии рассматриваемого блока во всех кэшах будут переведены в состояние S, вне зависимости от того, в каком состоянии они были до этого (M, E или S).

Когда процессор делает запрос на запись в блок, отсутствующий в локальной кэш-памяти (промах при записи), перед загрузкой в кэш-память блок должен быть считан из основной памяти (ОП) и модифицирован. Прежде чем процессор сможет загрузить блок, он должен убедиться, что в основной памяти действительно находится достоверная версия данных, то есть что в других кэшах отсутствует модифицированная копия данного блока. Формируемая в этом случае последовательность операций носит название *чтения с намерением модификации* (RWITM, Read With Intent To Modify).

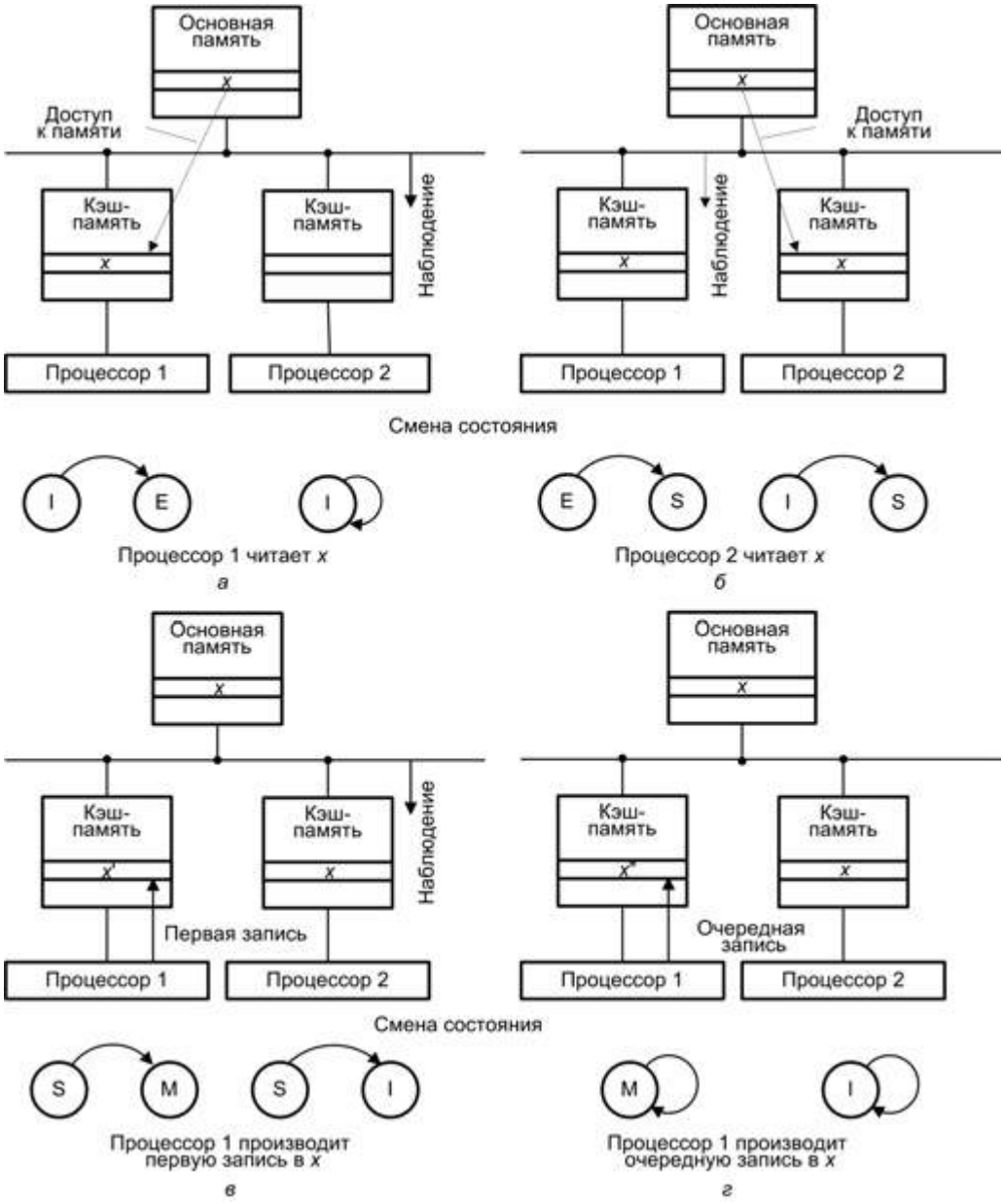


Рис. 11.17. Последовательность смены состояний в протоколе MESI: *а* — процессор 1 читает *x*; *б* — процессор 2 читает *x*; *в* — процессор 1 производит первую запись в *x*; *г* — процессор 1 производит очередную запись в *x*

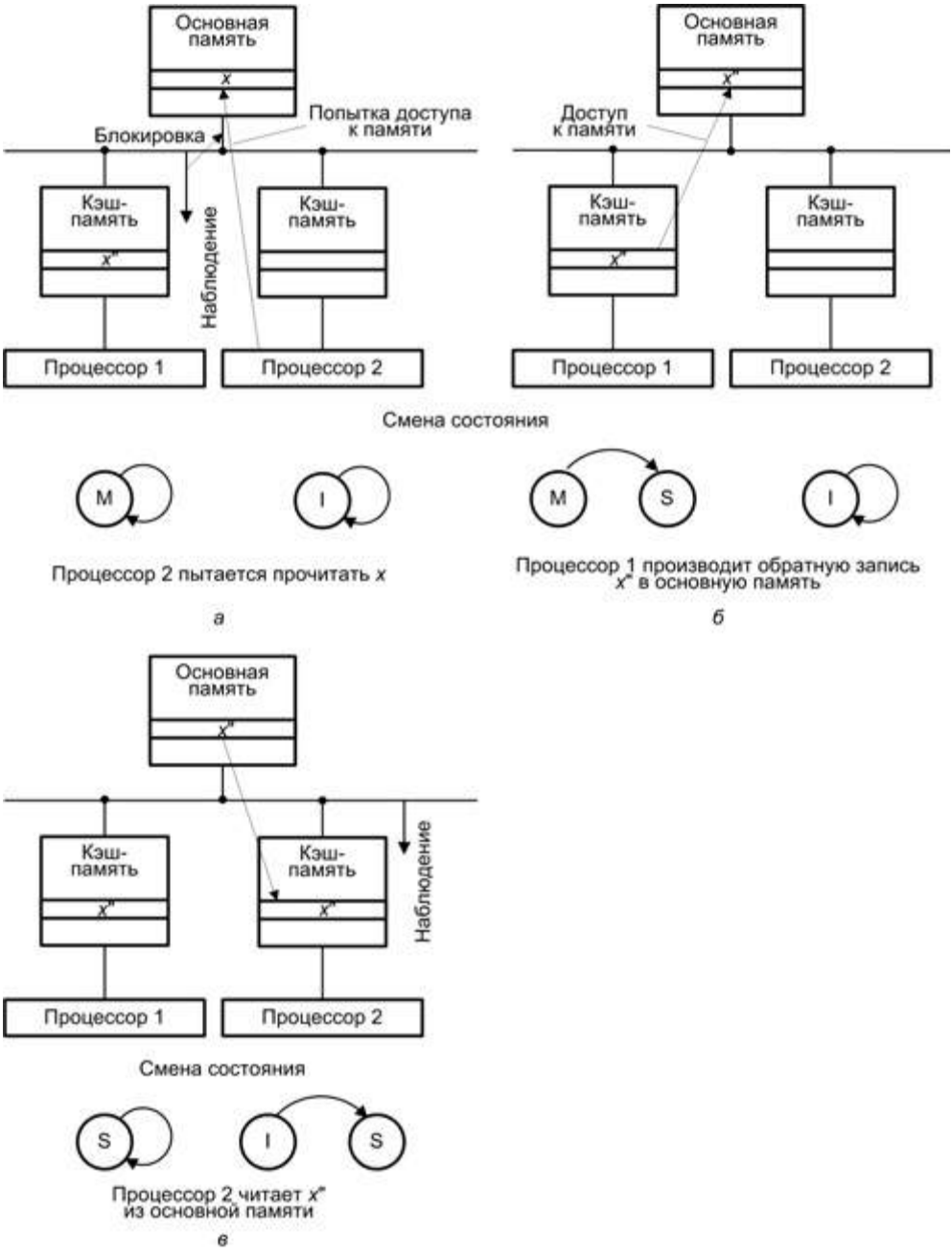


Рис. 11.18. Переход из состояния E в состояние S в протоколе MESI: *а* — процессор 2 пытается прочитать *x*; *б* — процессор 1 производит обратную запись *x*'' в основную память; *в* — процессор 2 читает *x*'' из основной памяти

Если в одном из кэшей обнаружилась копия нужного блока, причем в состоянии M, то процессор, обладающий этой копией, прерывает RWITM-последовательность и переписывает блок в основную память, после чего меняет состояние блока в своем кэше на I. Затем RWITM-последовательность возобновляется и делается повторное обращение к основной памяти для считывания обновленного блока. Окончательным состоянием блока будет M, при котором ни в ОП, ни в других кэшах нет еще одной достоверной его копии. Если копия блока существовала в другом кэше и не имела состояния M, то такая копия аннулируется и доступ к основной памяти производится немедленно.

Кэш-попадание при чтении не изменяет статуса читаемого блока. Если процессор выполняет доступ для записи в существующий блок, находящийся в состоянии S, он передает на шину широковещательный запрос, с тем чтобы информировать другие кэши, обновляет блок в своем кэше и присваивает ему статус M. Все остальные копии блока переводятся в состояние I. Если процессор производит доступ по записи в блок, находящийся в состоянии E, единственное, что он должен сделать, — это произвести запись в блок и изменить его состояние на M, поскольку другие копии блока в системе отсутствуют.

На рис. 11.17 показана типичная последовательность событий в системе из двух процессоров, запрашивающих доступ к ячейке x. Обращение к любой ячейке блока кэш-памяти рассматривается как доступ ко всему блоку.

Проиллюстрируем этапы, когда процессор 2 пытается прочитать содержимое ячейки x" (рис. 11.18). Сначала наблюдается кэш-промах по чтению и процессор пытается обратиться к основной памяти. Процессор 1 следит за шиной, обнаруживает обращение к ячейке, копия которой есть в его кэш-памяти и находится в состоянии M, поэтому он блокирует операцию чтения от процессора 2. Затем процессор 1 переписывает блок, содержащий x", в ОП и освобождает процессор 2, чтобы тот мог повторить доступ к основной памяти. Теперь процессор 2 получает блок, содержащий x", и загружает его в свою кэш-память. Обе копии помечаются как S.

До сих пор рассматривалась версия протокола MESI без однократной записи. С учетом однократной записи диаграмма состояний, изображенная на рис. 11.16, немного видоизменяется. Все кэш-промахи при чтении вызывают переход в состояние S. Первое попадание при записи сопровождается переходом в состояние E (так называемый переход однократной записи). Следующее попадание при записи влечет за собой изменение статуса строки на M.

Протоколы на основе справочника

Протоколы обеспечения когерентности на основе справочника характерны для сложных мультипроцессорных систем с совместно используемой памятью, где процессоры объединены многоступенчатой иерархической коммуникационной сетью. Сложность конфигурации сети приводит к тому, что применение протоколов наблюдения с их механизмом широковещания становится дорогостоящим и неэффективным.

Протоколы на основе справочника предполагают сбор и отслеживание информации о содержимом всех локальных кэшей. Такие протоколы обычно реализуются

с помощью централизованного контроллера, физически представляющего собой часть контроллера основной памяти. Собственно справочник хранится в основной памяти. Когда контроллер локальной кэш-памяти делает запрос, контроллер справочника обнаруживает такой запрос и формирует команды, необходимые для пересылки данных из основной памяти либо из другой локальной кэш-памяти, содержащей последнюю версию запрошенных данных. Центральный контроллер отвечает за обновление информации о состоянии локальных кэшей, поэтому он должен быть извещен о любом локальном действии, способном повлиять на состояние блока данных.

Справочник содержит множество записей, описывающих каждый кэшируемый блок основной памяти (ОП), который может быть совместно использован процессорами системы. Обращение к справочнику производится всякий раз, когда один из процессоров изменяет копию такого блока в своей локальной памяти. В этом случае информация из справочника нужна для того, чтобы аннулировать или обновить копии измененного блока в прочих локальных кэшах, где такие копии имеются.

Для разделяемого блока, копия которого может быть помещена в кэш-память, в справочнике выделяется одна запись, хранящая указатели на копии данного блока. Кроме того, в каждой записи выделен один бит модификации (D), показывающий, является ли копия «грязной» (D = 1 — dirty) или «чистой» (D = 0 — clean), то есть изменялось ли содержимое блока в кэш-памяти после того, как он был туда загружен. Этот бит указывает, имеет ли право процессор производить запись в данный блок.

В настоящее время известны три способа реализации протоколов обеспечения когерентности кэш-памяти на основе справочника: полный справочник, ограниченные справочники и сцепленные справочники.

В протоколе *полного справочника* единый централизованный справочник поддерживает информацию обо всех кэшах. Справочник хранится в основной памяти.

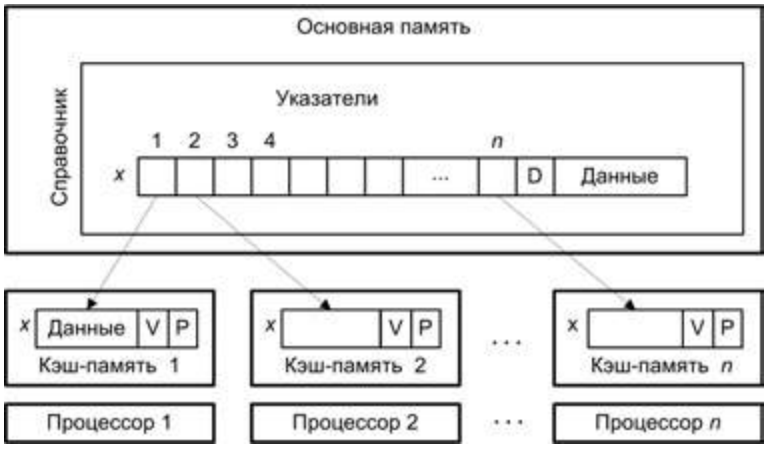


Рис. 11.19. Протокол обеспечения когерентности кэш-памяти с полным справочником

В системе из n процессоров каждая запись справочника будет содержать n однобитовых указателей. Если в соответствующей локальной кэш-памяти присутствует копия данных, бит-указатель устанавливается в 1, иначе — в 0. Схема с полным справочником показана на рис. 11.19. Здесь предполагается, что копия блока имеется в каждом кэше. Каждому блоку придаются два индикатора состояния: бит достоверности (V, Valid) и бит владения (P, Private). Если информация в блоке корректна, V-бит этого блока устанавливается в 1. Единичное значение P-бита указывает, что данному процессору предоставлено право на запись в соответствующий блок своей локальной кэш-памяти.

Предположим, что процессор 2 производит запись в ячейку x. В исходный момент процессор не получил еще разрешения на такую запись. Он формирует запрос к контроллеру справочника и ждет разрешения на продолжение операции. В ответ на запрос во все кэши, где есть копии блока, содержащего ячейку x, выдается сигнал аннулирования имеющихся копий. Каждый кэш, получивший этот сигнал, сбрасывает бит достоверности аннулируемого блока (V-бит) в 0 и возвращает контроллеру справочника сигнал подтверждения. После приема всех сигналов подтверждения контроллер справочника устанавливает в единицу бит модификации (D-бит) соответствующей записи справочника и посылает процессору 2 сигнал, разрешающий запись в ячейку x. С этого момента процессор 2 может продолжить запись в собственную копию ячейки x, а также в основную память, если в кэше реализована схема сквозной записи.

Основные проблемы протокола полного справочника связаны с большим количеством записей. Для каждой ячейки в справочнике системы из n процессоров требуется $n + 1$ бит, то есть с увеличением числа процессоров коэффициент сложности возрастает линейно. Протокол полного справочника допускает одновременное присутствие копии одного и того же блока во всех локальных кэшах. На практике такая возможность далеко не всегда остается востребованной — в каждый конкретный момент обычно актуальны лишь одна или несколько копий. В *протоколе с ограниченными справочниками* копии отдельного блока вправе находиться только в ограниченном числе кэшей — одновременно может быть не более чем m копий, при этом число указателей в записях справочника уменьшается до m ($m < n$). Чтобы однозначно идентифицировать кэш-память, хранящую копию, указатель вместо одного бита должен состоять из $\log_2 n$ битов, а общая длина указателей в каждой записи справочника вместо n битов будет равна $m \log_2 n$ битов. При постоянном значении m темпы роста коэффициента сложности ограниченного справочника по мере увеличения размера системы ниже, чем в случае линейной зависимости.

Когда одновременно требуется более чем m копий, контроллер принимает решение, какие из копий сохранить, а какие аннулировать, после чего производятся соответствующие изменения в указателях записей справочника.

Метод *сцепленных справочников* также ориентирован на сокращение объема справочника. Для хранения записей привлекается связный список, который может быть реализован как односвязный (однонаправленный) и двусвязный (двунаправленный).

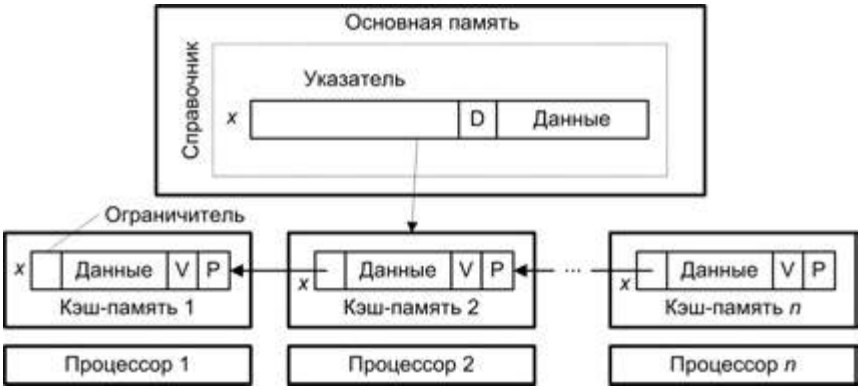


Рис. 11.20. Протокол обеспечения когерентности кэш-памяти со сцепленным справочником

В односвязном списке (рис. 11.20) каждая запись справочника содержит указатель на копию блока в одном из локальных кэшей. Копии одноименных блоков в разных кэшах системы образуют однонаправленную цепочку. Для этого в их тегах предусмотрено специальное поле, куда заносится указатель на кэш-память, содержащую следующую копию цепочки. В тег последней копии цепочки помещается специальный символ-ограничитель. Сцепленный справочник допускает цепочки длиной в n, то есть поддерживает n копий ячейки. При создании еще одной копии цепочку нужно разрушить, а вместо нее сформировать новую. Пусть, например, в процессоре 5 нет копии ячейки x, и он обращается за ней к основной памяти. Указатель в справочнике изменяется так, чтобы указывать на кэш с номером 5, а указатель в кэше 5 — таким образом, чтобы указывать на кэш 2. Для этого контроллер основной памяти наряду с затребованными данными должен передать в кэш-память 5 также и указатель на кэш-память с номером 2. Лишь после того как будет сформирована вся структура цепочки, процессор 5 получит разрешение на доступ к ячейке x. Если процессор производит запись в ячейку, то вниз по тракту, определяемому соответствующей цепочкой указателей, посылается сигнал аннулирования. Цепочка должна обновляться и при удалении копии из какой-либо кэш-памяти.

Двусвязный список поддерживает указатели как в прямом, так и в обратном направлениях. Это позволяет более эффективно вставлять в цепочку новые указатели или удалять из нее уже ненужные, но требует хранения большего числа указателей.

Схемы на основе справочника «страдают» от «заторов» в централизованном контроллере, а также от коммуникационных издержек в трактах между контроллерами локальных кэшей и центральным контроллером. Тем не менее они оказываются весьма эффективными в мультипроцессорных системах со сложной конфигурацией связей между процессорами, где невозможно реализовать протоколы наблюдения.

Ниже дана краткая характеристика актуальных (на настоящее время) протоколов обеспечения когерентности кэш-памяти на основе справочника. Для детального ознакомления с этими протоколами приведены ссылки на соответствующие литературные источники.

Протокол Tang. Здесь присутствует централизованный глобальный справочник, содержащий полную копию всей информации из каталогов каждого локального кэша. Это приводит к проблеме узких мест, а также требует поиска соответствующих входов.

Протокол Censier. В схеме справочника Censier для указания процессоров, содержащих локальную копию данного блока памяти, используется битовый вектор указателей. Такой вектор имеется для каждого блока памяти. Недостатками метода является его неэффективность при большом числе процессоров и, кроме того, для обновления блоков кэша требуется доступ к основной памяти [111].

Протокол Archibald. Схема справочника Archibald — это пара замысловатых схем для иерархически организованных сетей процессоров. С детальным описанием протокола можно ознакомиться в [47].

Протокол Stenstrom. Справочник Stenstrom для каждого блока данных предусматривает шесть допустимых состояний. Этот протокол относительно прост и подходит для любых топологий связей между процессорами. Справочник хранится в основной памяти. В случае кэш-промаха при чтении происходит обращение к основной памяти, которая посылает сообщение кэш-памяти, являющейся владельцем блока, если такой находится. Получив это сообщение, кэш-владелец посылает затребованные данные, а также направляет сообщение всем остальным процессорам, совместно использующим эти данные, для того чтобы они обновили свои битовые векторы. Схема не очень эффективна при большом числе процессоров, однако в настоящее время это наиболее проработанный и широко распространенный протокол на основе справочника [111].

Контрольные вопросы

1. Проанализируйте влияние особенностей ВС с общей памятью и ВС с распределенной памятью на разработку программного обеспечения. Почему эти ВС называют соответственно сильно связанными и слабо связанными?
2. Поясните идею с чередованием адресов памяти. Из каких соображений выбирается механизм распределения адресов? Как он связан с классом архитектуры ВС?
3. Дайте сравнительную характеристику однородного и неоднородного доступов к памяти.
4. В чем заключаются преимущества архитектуры COMA?
5. Проведите сравнительный анализ моделей с кэш-когерентным и кэш-некогерентным доступом к неоднородной памяти.
6. Сформулируйте достоинства и недостатки архитектуры без прямого доступа к удаленной памяти.
7. Объясните смысл распределенной и совместно используемой памяти.
8. Разработайте свой пример, иллюстрирующий проблему когерентности кэш-памяти.
9. Охарактеризуйте особенности программных способов решения проблемы когерентности, выделите их преимущества и слабые стороны.

10. Сравните методики записи в память с аннулированием и записи в память с трансляцией, акцентируя их достоинства и недостатки.
11. Дайте сравнительную характеристику методов для поддержания когерентности в мультипроцессорных системах.
12. Выполните сравнительный анализ известных вам протоколов наблюдения.
13. Какой из протоколов наблюдения наиболее популярен? Обоснуйте причины повышенного к нему интереса.
14. Дайте развернутую характеристику протоколов когерентности на основе справочника и способов их реализации. В чем суть отличий этих протоколов от протоколов наблюдения?

Глава 12

Топология вычислительных систем

Сама идея многопроцессорной вычислительной системы предполагает обмен данными между компонентами этой ВС. Коммуникационная система ВС представляет собой сеть, *узлы* которой связаны трактами передачи данных — *каналами*. В роли узлов сети могут выступать процессоры, банки памяти, устройства ввода/вывода, коммутаторы либо несколько перечисленных элементов, объединенных в группу. Организация внутренних коммуникаций вычислительной системы называется *топологией*.

Топологию сети определяет множество узлов N, объединенных множеством каналов C. Связь между узлами обычно реализуется по двухточечной схеме (point-to-point). Любые два узла, связанные каналом связи, называют смежными узлами или соседями. Каждый канал $c = (x, y) \in C$ соединяет один узел-источник (source node) x с одним узлом-получателем (recipient node) y, где $x, y \in N$. Часто пары узлов соединяют два канала — по одному в каждом направлении. Канал характеризуют шириной w_c — числом сигнальных линий; частотой f_c — скоростью передачи битов по каждой сигнальной линии; задержкой t_c — временем пересылки бита из узла x в узел y. Для большинства каналов задержка находится в прямой зависимости от физической длины линии связи l_c и скорости распространения сигнала v: $l_c = v\, t_c$. Полоса пропускания канала b_c определяется выражением $b_c = w_c f_c$.

Классификация коммуникационных сетей

Классификацию сетей обычно производят по следующим признакам:

- стратегия синхронизации;
- стратегия коммутации;
- стратегия управления;
- топология.

Классификация по стратегии синхронизации

Две возможных стратегии синхронизации операций в сети — это синхронная и асинхронная. В синхронных сетях все действия жестко согласованы во времени,

что обеспечивается за счет единого генератора тактовых импульсов (ГТИ), сигналы которого одновременно транслируются во все узлы. В асинхронных сетях единого генератора нет, а функции синхронизации распределены по всей системе, причем в разных частях сети часто используются локальные ГТИ. Синхронные сети по сравнению с асинхронными обычно медленнее, но в них легче предотвращаются конфликтные ситуации.

Классификация по стратегии коммутации

По стратегии коммутации различают *сети с коммутацией соединений* и *сети с коммутацией пакетов*. Как в первом, так и во втором варианте информация пересылается в виде пакета. *Пакет* представляет собой группу битов, для обозначения которой применяют также термин *сообщение*.

В сетях с коммутацией соединений еще до начала пересылки сообщения формируется тракт от узла-источника до узла-получателя (путем соответствующей установки коммутирующих элементов сети). Этот тракт сохраняется вплоть до доставки пакета в пункт назначения. Пересылка сообщений между определенной парой узлов производится всегда по одному и тому же маршруту.

Сети с коммутацией пакетов предполагают, что сообщение самостоятельно находит свой путь к месту назначения. В отличие от сетей с коммутацией соединений, маршрут от исходного пункта к пункту назначения каждый раз может быть иным. Пакет последовательно проходит через узлы сети. Очередной узел запоминает принятый пакет в своем буфере временного хранения, анализирует его и решает, что с ним делать дальше. В зависимости от загруженности сети принимается решение о возможности немедленной пересылки пакета к следующему узлу и о дальнейшем маршруте следования пакета на пути к цели. Если все возможные тракты для перемещения пакета к очередному узлу заняты, в буфере узла формируется очередь пакетов, которая «рассасывается» по мере освобождения линий связи между узлами (при насыщении очереди, согласно одной из стратегий маршрутизации, может произойти так называемый «сброс хвоста» (tail drop) — отказ от вновь поступающих пакетов). Хотя сети с коммутацией пакетов по сравнению с сетями на базе коммутации соединений более эффективно используют ресурсы сети, время доставки сообщения в них сильно зависит от переменных задержек в перемещении пакетов.

Классификация по стратегии управления

Коммуникационные сети можно также классифицировать по тому, как в них организовано управление. В некоторых сетях, особенно с коммутацией соединений, принято *централизованное управление* (рис. 12.1). Процессоры посылают запрос на обслуживание в единый контроллер сети, который производит арбитраж запросов с учетом заданных приоритетов и устанавливает нужный маршрут. К данному типу следует отнести сети с шинной топологией. Процессорные матрицы также строятся как сети с централизованным управлением, которое осуществляется сигналами от центрального процессора. Приведенная схема применима и к сетям с коммутацией

пакетов. Здесь маршрут определяется адресом узла назначения, хранящимся в заголовке пакета. Большинство из серийно выпускаемых ВС имеют именно этот тип управления.

Рис. 12.1. Структура сети с централизованным управлением

В схемах с *децентрализованным управлением* функции управления распределены по узлам сети. Вариант с централизацией проще реализуется, но расширение сети в этом случае связано со значительными трудностями. Децентрализованные сети в плане подключения дополнительных узлов значительно гибче, однако взаимодействие узлов в таких сетях существенно сложнее.

В ряде сетей связь между узлами обеспечивается посредством множества коммутаторов, но существуют также сети с одним коммутатором. Наличие большого числа коммутаторов ведет к увеличению времени передачи сообщения, но позволяет использовать простые переключающие элементы. Подобные сети обычно строятся как многоступенчатые.

Классификация по топологии

Топологию коммуникационной сети (организацию внутренних коммуникаций многопроцессорной вычислительной системы) можно рассматривать как функцию отображения множества процессоров (ПР) и банков памяти (БП) на то же самое множество ПР и БП. Иными словами, топология описывает, каким образом процессоры и банки памяти могут быть соединены с другими процессорами и банками памяти.

В основу системы классификации обычно кладут разбиение всех возможных топологий на *статические* и *динамические*. В сетях со статическими топологиями структура связей фиксирована. В сетях с динамической топологией конфигурация соединений в процессе вычислений может быть оперативно изменена (с помощью программных средств).

Узел в сети может быть терминальным, то есть источником или приемником данных, коммутатором, пересылающим информацию с входного порта на выходной, или совмещать обе роли. В сетях *с непосредственными связями* (direct networks) каждый узел одновременно является как терминальным узлом, так и коммутатором, и сообщения пересылаются между терминальными узлами напрямую. В сетях *с косвенными связями* (indirect networks) узел может быть либо терминальным, либо коммутатором, но не одновременно, поэтому сообщения передаются опосредовано, с помощью выделенных коммутирующих узлов. (В дальнейшем условимся называть оба варианта «прямыми» и «косвенными» сетями, а терминальный узел будем называть «терминалом».) Существуют также такие топологии, которые нельзя однозначно причислить ни к прямым, ни к косвенным. Любую прямую сеть можно изобразить в виде косвенной, разделив каждый узел на два — терминальный узел и узел коммутации. Современные прямые сети реализуются именно таким образом — коммутатор отделяется от терминального узла и помещается в выделенный маршрутизатор. Основное преимущество прямых сетей в том, что коммутатор может использовать ресурсы терминальной части своего узла. Это становится существенным, если учесть, что, как правило, последний включает в себя вычислительную машину или процессор.

Схема классификации коммуникационных сетей на основе их топологии показана на рис. 12.2.

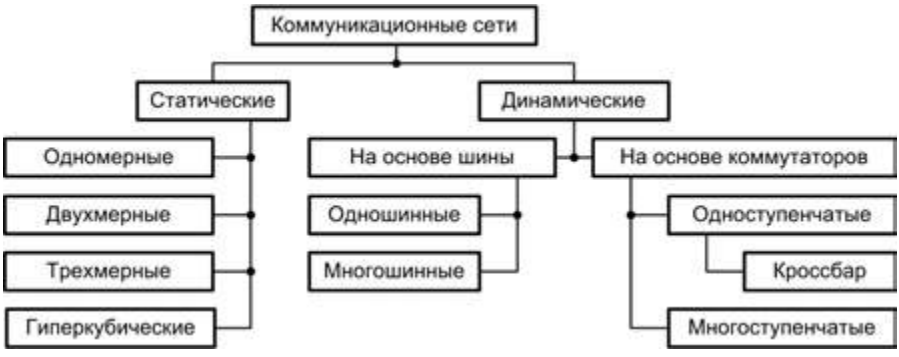


Рис. 12.2. Классификация коммуникационных сетей по топологии

Метрики сетевых соединений

Чтобы охарактеризовать сеть, обычно используют следующие параметры:

- размер сети;
- число связей;
- диаметр сети;
- степень узла;
- пропускная способность сети;
- задержка сети;

- связность сети;
- ширина бисекции сети;
- полоса бисекции сети.

Размер сети (N) численно равен количеству узлов, объединяемых сетью.

Число связей (I) — это суммарное количество каналов между всеми узлами сети. Иногда этот параметр называют *стоимостью сети*. В плане стоимости лучшей следует признать ту сеть, которая требует меньшего числа связей.

Диаметр сети (D), называемый также *коммуникационным расстоянием*, определяет минимальный путь, по которому проходит сообщение между двумя наиболее удаленными друг от друга узлами сети. *Путь* в сети — это упорядоченное множество каналов $P = \{c_1, c_2, ..., c_n\}$, по которым данные от узла-источника, последовательно переходя от одного промежуточного узла к другому, поступают на узел-получатель. Для обозначения отрезка пути между парой смежных узлов применяют термин *переход* (в живой речи также «транзит» и «хоп»). Минимальный путь от узла x до узла y — это путь с минимальным числом переходов. Если обозначить число переходов в минимальном пути от узла x до узла y через $H(x, y)$, то диаметр сети D — это наибольшее значение $H(x, y)$ среди всех возможных комбинаций x и y. Так, в цепочке из четырех узлов наибольшее число переходов будет между крайними узлами, и «диаметр» такой цепочки равен трем. С возрастанием диаметра сети увеличивается общее время прохождения сообщения, поэтому разработчики ВС стремятся по возможности обходиться меньшим диаметром.

Степень узла (d). Каждый узел сети x связан с прочими узлами множеством каналов $C_x = C_{Ix} \cup C_{Ox}$, где C_{Ix} — множество входных каналов, а C_{Ox} — множество выходных каналов. Степень узла x представляет собой сумму числа входных и выходных каналов узла, то есть она равна числу узлов сети, с которыми данный узел связан напрямую. Например, в сети, организованной в виде матрицы, где каждый узел связан только с ближайшими соседями (слева, справа, сверху и снизу), степень узла равна четырем. Увеличение степени узлов ведет к усложнению коммутационных устройств сети и, как следствие, к дополнительным задержкам в передаче сообщений. С другой стороны, повышение степени узлов позволяет реализовать топологии, имеющие меньший диаметр сети, и тем самым сократить время прохождения сообщения. Разработчики ВС обычно отдают предпочтение таким топологиям, где степень всех узлов одинакова, что позволяет строить сети по модульному принципу.

Пропускная способность сети (W) характеризуется количеством информации, которое может быть передано по сети в единицу времени. Обычно измеряется в мегабайтах в секунду или гигабайтах в секунду без учета издержек на передачу избыточной информации, например битов паритета.

Задержка сети (T) — это время, требуемое на прохождение сообщения через сеть. В сетях, где время передачи сообщений зависит от маршрута, говорят о минимальной, средней и максимальной задержках сети.

Связность сети (Q) можно определить как минимальное число параллельных трактов между любой парой узлов. Связность сети характеризует устойчивость

сети к повреждениям, то есть ее способность обеспечивать функционирование ВС при отказе компонентов сети.

Ширина бисекции сети (B). Для начала определим понятие *среза сети* $C(N_1, N_2)$ как множество каналов, разрыв которых разделяет множество узлов сети N на два непересекающихся подмножества узлов N_1 и N_2. Каждый элемент $C(N_1, N_2)$ — это канал, соединяющий узел из N_1 с узлом из N_2. Бисекция сети — это срез сети, разделяющий ее примерно пополам, то есть так, что $|N_2| \leqslant |N_1| \leqslant |N_2| + 1$. Ширину бисекции B характеризуют минимальным числом каналов, разрываемых при всех возможных бисекциях сети:

$$B = \min_{bisection} \left| C(N_1,\ N_2) \right|.$$

Ширина бисекции позволяет оценить число сообщений, которые могут быть переданы по сети одновременно, при условии что это не вызовет конфликтов из-за попытки использования одних и тех же узлов или линий связи.

Полоса бисекции сети (b) — это наименьшая полоса пропускания по всем возможным бисекциям сети. Она характеризует пропускную способность тех линий связи, которые разрываются при бисекции сети, и позволяет оценить наихудшую пропускную способность сети при попытке одномоментной передачи нескольких сообщений, если эти сообщения должны проходить из одной половины сети в другую. Полоса бисекции b определяется выражением $b = \min_{bisection} B(N_1,\ N_2)$. Для сетей с одинаковой шириной полосы b_c во всех каналах справедливо: $b = b_c \times B$. Малое значение полосы бисекции свидетельствует о возможности конфликтов при одновременной пересылке нескольких сообщений.

Функции маршрутизации данных

Важнейшим вопросом при выборе топологии является способ маршрутизации данных, то есть правило выбора очередного узла, которому пересылается сообщение. Основой маршрутизации служат адреса узлов. Каждому узлу в сети присваивается уникальный адрес. Исходя из этих адресов, а точнее их двоичных представлений, производится соединение узлов в статических топологиях или их коммутация в топологиях динамических. В сущности, принятая система соответствия между двоичными кодами адресов смежных узлов — *функция маршрутизации данных* — и определяет топологию сети. Последнюю можно описать как набор функций маршрутизации, задающий порядок выбора промежуточных узлов на пути от узла-источника к узлу-получателю. В некоторых топологиях используется единая для всей сети функция маршрутизации, в других — многоступенчатых — при переходе от одной ступени к другой может применяться иная функция маршрутизации.

Функция маршрутизации данных определяет алгоритм манипуляции битами адреса узла-источника для определения адреса узла-получателя. Ниже приводится формальное описание наиболее распространенных функций маршрутизации данных. Для всех функций предполагается, что размер сети равен N, а разрядность адре-

са — n, где $n = \log_2 N$. Биты адреса обозначены как x_i. В приводимых ниже примерах принято, что $N = 8$ ($n = 3$).

Кубическая перестановка

Функция *кубической перестановки* (cube permutation) отвечает следующему соотношению:

$$E_i(x_{n-1} \ldots x_{i+1}x_i x_{i-1} \ldots x_0) = x_{n-1} \ldots x_{i+1}\overline{x_i}\, x_{i-1} \ldots x_0, \quad 0 \le i \le n-1.$$

Двоичное представление адреса узла-получателя получается путем инвертирования i-го бита в адресе источника.

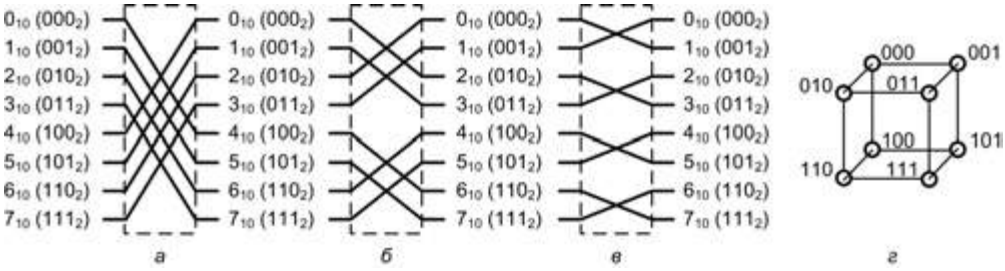


Рис. 12.3. Примеры топологий с кубической перестановкой: *а* — при $i = 2$; *б* — при $i = 1$; в — при $i = 0$; *г* — трехмерный гиперкуб

На рис. 12.3, *а*, *б*, *в* показана топология связей в сети, построенной в соответствии с функцией кубической перестановки, для трех значений i. Примером использования данной функции маршрутизации, где использованы все три возможных значения i, может служить топология трехмерного гиперкуба (рис. 12.3, *г*). Вариант функции для $i = 0$ известен также под названием *обменной перестановки* (exchange permutation).

Тасующая подстановка

Функция *тасующей подстановки* может быть реализована в одном из четырех вариантов, из которых наиболее распространены два: *совершенная тасующая подстановка* (perfect shuffle permutation) и *инверсная совершенная тасующая подстановка* (inverse perfect shuffle permutation). Ниже приведены формальные описания этих вариантов, а на рис. 12.4 — примеры соответствующих им топологий.

- *совершенная тасующая подстановка*:

$$S(x_{n-1}x_{n-2} \ldots x_1x_0) = x_{n-2} \ldots x_1x_0x_{n-1}.$$

  Из приведенной формулы видно, что адрес узла-получателя может быть получен из двоичного кода узла-источника циклическим сдвигом этого кода влево на одну позицию. Если использовать аналогию с картами, то тасующая подстановка эквивалентна разбиению колоды карт на две половины с последующим равномерным чередованием карт из каждой половины.

- *инверсная совершенная тасующая подстановка*:

$$U(x_{n-1}x_{n-2}\ \ldots\ x_1x_0) = x_0x_{n-1}\ \ldots\ \ x_2x_1.$$

Здесь также используется циклический сдвиг, но вправо.

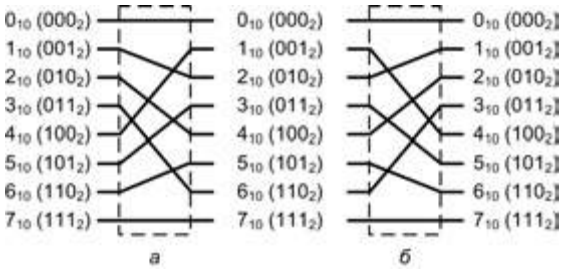


Рис. 12.4. Примеры топологий с тасующей подстановкой: *а* — совершенная; *б* — инверсная совершенная

Баттерфляй

Функция «*баттерфляй*» (butterfly) — «бабочка», была разработана в конце 60-х годов Рабинером и Гоулдом. Свое название она получила из-за того, что построенная в соответствии с ней сеть по конфигурации напоминает крылья бабочки (рис. 12.5).

Математически функция может быть записана в виде:

$$B_i(x_{n-1}\ \ldots\ x_{i+1}x_ix_{i-1}\ \ldots\ x_0) = x_{n-1}\ \ldots\ x_{i+1}x_0x_{i-1}\ \ldots\ x_1x_i,\quad 0 \le i \le n-1.$$

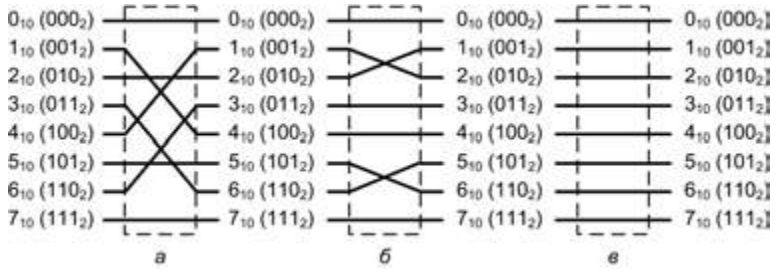


Рис. 12.5. Примеры топологии «баттерфляй»: *а* — при $i = 2$; *б* — при $i = 1$; *в* — при $i = 0$

Двоичное представление узла-получателя получается путем взаимной перестановки в адресе узла-источника битов с индексами i и 0. Хотя *баттерфляй*-функция используется в основном при объединении ступеней в сетях с динамической многоступенчатой топологией, известны также и «чистые» *баттерфляй*-сети.

Реверсирование битов

Как следует из названия, функция сводится к перестановке битов адреса в обратном порядке:

$$R(x_{n-1}x_{n-2}\ \ldots\ x_1x_0) = x_0x_1\ \ldots\ x_{n-2}x_{n-1}.$$

Соответствующая топология для $n = 4$ показана на рис. 12.6. Хотя для значений $n \leqslant 3$ топология реверсирования битов совпадает с топологией «баттерфляй», при больших значениях n различия становятся очевидными.

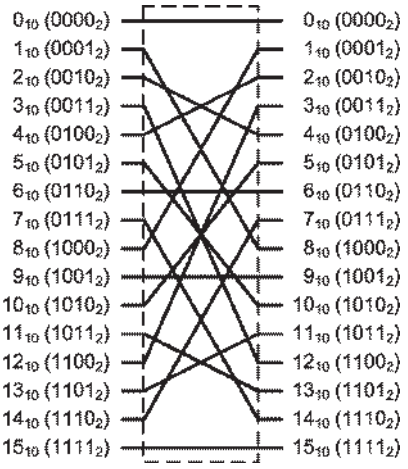


Рис. 12.6. Пример топологии на основе формулы реверсирования битов

Базисная линия

Функция маршрутизации типа *базисной линии* определяется соотношением:

$$L_i(x_{n-1} \ldots x_{i+1}x_ix_{i-1} \ldots x_1x_0) = x_{n-1} \ldots x_{i+1}x_0x_i \ldots x_1, \quad 0 \leq i \leq n-1.$$

и сводится к циклическому сдвигу $i + 1$ младших цифр адреса узла-источника на одну позицию вправо. Соответствующие топологии для различных значений i показаны на рис. 12.7.

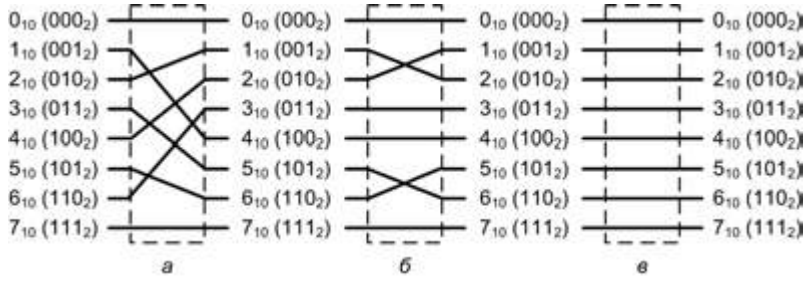


Рис. 12.7. Примеры топологии базисной линии: *а* — при $i = 2$; *б* — при $i = 1$; *в* — при $i = 0$

Статические топологии

К статическим топологиям относят такие, где между двумя узлами возможен только один прямой фиксированный путь, то есть статические топологии не пред-

полагают наличия в сети коммутирующих устройств. Если же такие устройства имеются, то используются они только перед выполнением конкретной задачи, а в процессе всего времени вычислений топология остается неизменной.

Из возможных показателей классификации статических сетей чаще всего выбирают их размерность. С этих позиций различают:

- одномерные топологии (линейный массив);
- двумерные топологии (кольцо, звезда, дерево, решетка, систолический массив);
- трехмерные топологии (полносвязная топология, хордальное кольцо);
- гиперкубическую топологию.

Ниже рассматриваются основные виды статических топологий.

Линейная топология

В *линейной топологии* узлы сети образуют одномерный массив и соединены в *цепочку* (рис. 12.8). Линейная топология характеризуется следующими параметрами: $D = N - 1$; $d = 1$ (для крайних узлов) и $d = 2$ (для всех остальных узлов); $I = N - 1$; $B = 1$.

Рис. 12.8. Линейная топология

Линейная топология не обладает свойством полной симметричности, поскольку узлы на концах цепочки имеют только одну коммуникационную линию, то есть их степень равна 1, в то время как степень остальных узлов равна 2. Время пересылки сообщения зависит от расстояния между узлами, а отказ одного из них способен привести к невозможности пересылки сообщения. По этой причине в линейных сетях используют отказоустойчивые узлы, которые при отказе изолируют себя от сети, позволяя сообщению миновать неисправный узел. Данный вид топологии наибольшее распространение нашел в системах класса SIMD.

Кольцевые топологии

Стандартная *кольцевая топология* представляет собой линейную цепочку, концы которой соединены между собой (рис. 12.9). В зависимости от числа каналов между соседними узлами (один или два) различают *однонаправленные* и *двунаправленные* кольца. Кольцевая топология характеризуется следующими параметрами: $D = \min\left[\frac{N}{2}\right]$ (для двунаправленного кольца) и $D = N - 1$ (для однонаправленного кольца); $d = 2$; $I = N$; $B = 2$. Введение одного дополнительного канала вдвое уменьшает диаметр и увеличивает ширину бисекции. Кольцевая топология, по сравнению с линейной, менее популярна, поскольку добавление или удаление узла требует демонтажа сети. Тем не менее она нашла применение в ряде ВС, например в вычислительных системах KSR-1 и SCI.

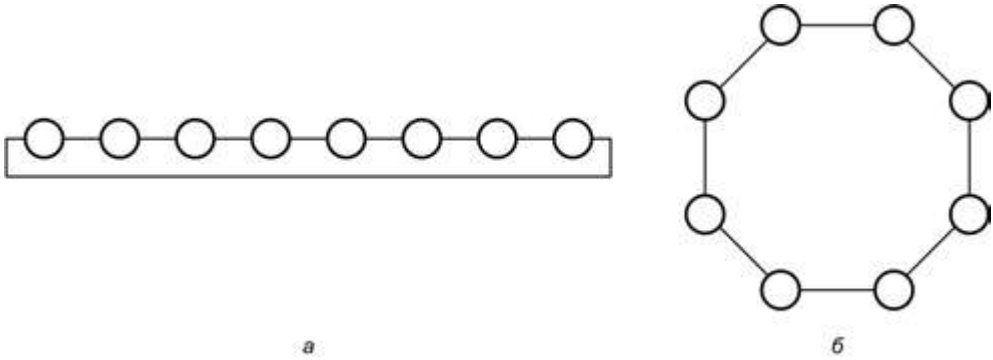


Рис. 12.9. Кольцевая топология: *а* — стандартное представление; *б* — альтернативное представление

Один из способов уменьшения диаметра и увеличения ширины бисекции кольцевой сети — добавление линий связи в виде хорд, соединяющих определенные узлы кольца. Подобная топология носит название *хордальной*. Если хорды соединяют узлы с шагом 1 или $\frac{N}{2}-1$, диаметр сети уменьшается вдвое. На рис. 12.10 показана *хордальная кольцевая сеть* с шагом 3.

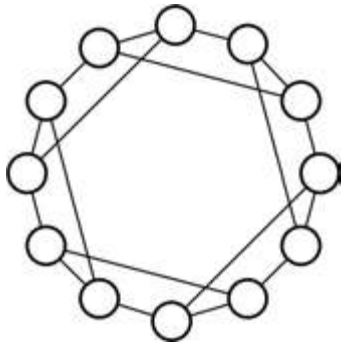


Рис. 12.10. Кольцевая хордальная топология

Введение хорд приводит к возрастанию степени узлов, а значит, и их сложности. В то же время появляется возможность отключения неисправного узла с сохранением работоспособности сети. Примером использования хордальной топологии может служить вычислительная система ASP.

Звездообразная топология

Звездообразная сеть объединяет множество узлов с $d = 1$ посредством специализированного центрального узла — концентратора (рис. 12.11). Топология характеризуется такими параметрами: $D = 2$; $d = 1$ (для краевых узлов) и $d = N - 1$ (для узла-концентратора); $I = N - 1$; $B = 1$.

Звездообразная организация узлов и соединений редко используется для объединения процессоров многопроцессорной ВС, но хорошо работает, когда поток информации идет от нескольких вторичных узлов, соединенных с одним первичным узлом, например, при подключении терминалов. Общая пропускная способность сети обычно ограничивается быстродействием концентратора, аналогично тому, как сдерживающим элементом в одношинной топологии выступает шина. По производительности эти топологии также идентичны. Основное преимущество звездообразной схемы в том, что конструктивное исполнение узлов на концах сети может быть очень простым.

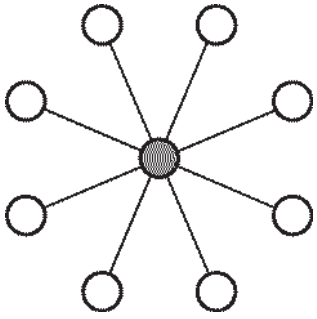


Рис. 12.11. Звездообразная топология

Древовидные топологии

Еще одним вариантом структуры сети является *древовидная топология* (рис. 12.12, *а*). Сеть строится по схеме так называемого строго двоичного дерева, где каждый узел более высокого уровня связан с двумя узлами следующего по порядку более низкого уровня. Узел, находящийся на более высоком уровне, принято называть *родительским*, а два подключенных к нему нижерасположенных узла — *дочерними*. В свою очередь, каждый дочерний узел выступает в качестве родительского для двух узлов следующего более низкого уровня. Отметим, что каждый узел связан только с двумя дочерними и одним родительским. Такую сеть можно охарактеризовать следующими параметрами: $D = 2\log_2\left(\frac{N+1}{2}\right)$; $d = 1$ (для краевых узлов), $d = 2$ (для корневого узла) и $d = 3$ (для остальных узлов); $I = N - 1$; $B = 1$. Диаметр для двоичного дерева возрастает пропорционально лишь логарифму числа узлов, в то время как степень узла остается постоянной. Сеть с такой топологией хорошо масштабируется. Основной недостаток топологии — малая ширина бисекции, что предполагает ограниченную полосу пропускания.

Топология двоичного дерева была использована в мультипроцессорной системе DADO из 1023 узлов, разработанной в Колумбийском университете.

При больших объемах пересылок между несмежными узлами древовидная топология оказывается недостаточно эффективной, поскольку сообщения должны

проходить через один или несколько промежуточных звеньев. Очевидно, что на более высоких уровнях сети вероятность затора из-за недостаточно высокой пропускной способности линий связи выше. Этот недостаток устраняют с помощью топологии, называемой «*толстым*» деревом (рис. 12.12, *б*).

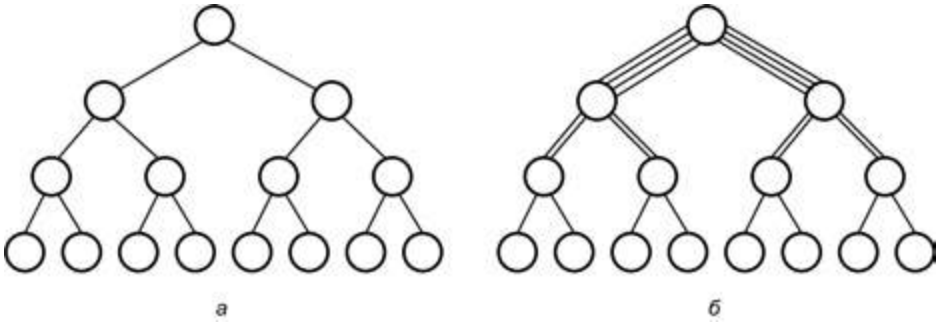


Рис. 12.12. Древовидная топология: *а* — стандартное дерево; *б* — «толстое» дерево

Идея «толстого» дерева состоит в увеличении пропускной способности коммуникационных линий на прикорневых уровнях сети. С этой целью на верхних уровнях сети родительские и дочерние узлы связывают не одним, а несколькими каналами, причем чем выше уровень, тем больше число каналов. На рисунке это отображено в виде множественных линий между узлами верхних уровней. Топология «толстого» дерева реализована в вычислительной системе CM-5.

Решетчатые топологии

Поскольку значительная часть научно-технических задач связана с обработкой массивов, желательно отражать эту специфику в топологии соответствующих ВС. Такие топологии относят к *решетчатым* (mesh), а их конфигурация определяется видом и размерностью массива.

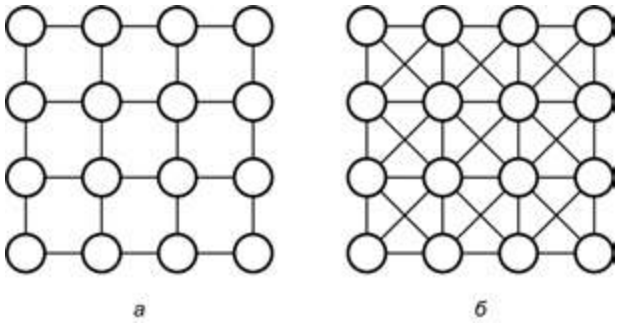


Рис. 12.13. Топологии плоской решетки: *а* — с 4 связями; *б* — с 8 связями

Простейшими примерами для одномерных массивов могут служить цепочка и кольцо. Для двухмерных массивов данных наиболее подходит топология плоской

прямоугольной матрицы узлов, каждый из которых соединен с ближайшим соседом (рис. 12.13, *а*). Такая сеть размерности $n \times n$ ($n = \sqrt{N}$) имеет следующие характеристики: $D = 2(n - 1)$; $d = 2$ (для угловых узлов), $d = 3$ (для краевых узлов) и $d = 4$ (для остальных узлов); $I = 2(N - n)$; $B = n$.

Возможны и иные формы плоской двухмерной решетчатой топологии (рис. 12.13, *б*). Этот вариант характеризуется следующими параметрами: $D = n$; $d = 3$ (для угловых узлов), $d = 5$ (для краевых узлов) и $d = 8$ (для остальных узлов); $I = 2(2N - 3n + 1)$; $B = 2n$. Введение диагональных связей позволяет сократить диаметр и увеличить ширину бисекции сети, но при этом возрастает степень узлов.

Двухмерные решетчатые топологии встречаются в вычислительных системах класса SIMD и транспьютерных ВС. Они были использованы в системах Intel Paragon и Intel Touchstone Delta.

Если провести операцию *свертывания* (wraparound) плоской матрицы, соединив информационными трактами одноименные узлы верхней и нижней строк плоской матрицы (рис. 12.14, *а*) или левого и правого столбцов (рис. 12.14, *б*), так чтобы столбцы (строки) образовывали кольцо, то из плоской конструкции получаем топологию типа цилиндра. В топологии цилиндра каждый ряд (или столбец) матрицы представляет собой кольцо. Если одновременно произвести свертывание плоской матрицы в обоих направлениях, получим тороидальную топологию сети (рис. 12.14, *в*). Двухмерный тор на базе решетки $n \times n$ обладает следующими параметрами: $D = 2\min\left[\frac{n}{2}\right]$; $d = 4$; $I = 2N$; $B = 2n$. Сеть с топологией тора используется в системах AP3000 компании Fujitsu.

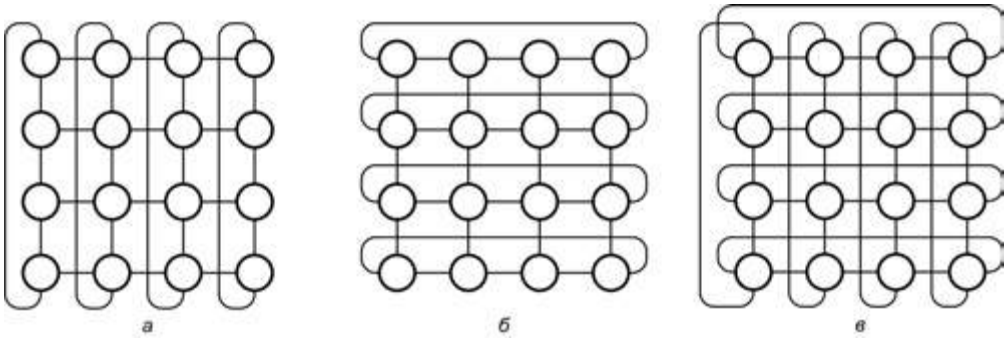


Рис. 12.14. Двухмерные решетчатые топологии с применением операции свертывания: *а* — по столбцам (цилиндр); *б* — по строкам (цилиндр); *в* — по столбцам и строкам (двухмерный тор)

Помимо свертывания к плоской решетке может быть применена операция *скручивания* (twisting). Суть этой операции состоит в том, что все узлы объединяются в разомкнутую или замкнутую спираль, то есть узлы, расположенные с противоположных краев плоской решетки, соединяются с некоторым сдвигом. Это приводит к топологиям *витого цилиндра* и *витого тора*. Если горизонтальные петли объединены

в виде спирали, образуется так называемая сеть типа ILLIAC. На рис. 12.15, *а* показана подобная конфигурация сети, соответствующая хордальной сети четвертого порядка и характеризуемая следующими значениями: $D = n - 1$; $d = 4$; $I = 2N$; $B = 2n$. Если же столбцы узлов также объединены в виде спирали, получаем топологию, показанную на рис. 12.15, *б*. В этом случае имеем два больших кольца.

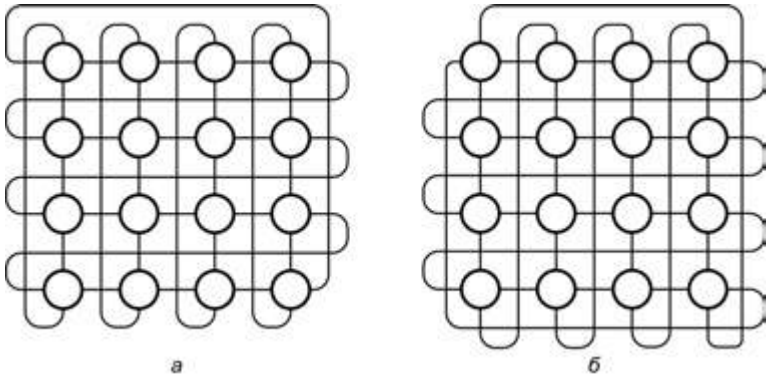


Рис. 12.15. Двухмерные решетчатые топологии с применением операции скручивания: *а* — ILLIAC; *б* — витой тор

Следует упомянуть и трехмерные сети. Один из вариантов, реализованный в архитектуре ВС Cray T3D, представляет собой трехмерный тор, образованный объединением процессоров в кольца по трем координатам: *x*, *y* и *z*.

Примерами ВС, где реализованы различные варианты решетчатых топологий, могут служить: ILLIAC IV, MPP, DAP, CM-2, Paragon и др.

Полносвязная топология

В *полносвязной топологии* (рис. 12.16), известной также под названием топологии «*максимальной группировки*» или «*топологии клика*» (clique — полный подграф), каждый узел напрямую соединен со всеми остальными узлами сети. Сеть, состоящая из N узлов, имеет следующие параметры: $D = 1$; $d = N - 1$; $I = \frac{N(N-1)}{2}$; $B = \frac{N^2}{4}$.

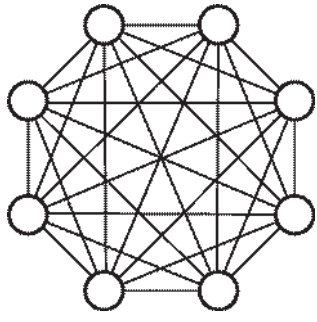


Рис. 12.16. Полносвязная топология

Если размер сети велик, топология становится дорогостоящей и трудно реализуемой. Более того, топология максимальной группировки не дает существенного улучшения производительности, поскольку каждая операция пересылки требует, чтобы узел проанализировал состояние всех своих $N - 1$ входов. Для ускорения этой операции необходимо, чтобы все входы анализировались параллельно, что, в свою очередь, усложняет конструкцию узлов.

Топология гиперкуба

При объединении параллельных процессоров весьма популярна топология гиперкуба, показанная на рис. 12.17. Линия, соединяющая два узла (рис. 12.17, *а*), определяет одномерный гиперкуб. Квадрат, образованный четырьмя узлами (рис. 12.17, *б*), — двумерный гиперкуб, а куб из 8 узлов (рис. 12.17, *в*) — трехмерный гиперкуб и т. д. Из этого ряда следует алгоритм получения n-мерного гиперкуба: начинаем с $(n - 1)$-мерного гиперкуба, делаем его идентичную копию, а затем добавляем связи между каждым узлом исходного гиперкуба и одноименным узлом копии. Гиперкуб размерности $n = \log_2 N$ имеет следующие характеристики: $D = n$; $d = n$; $I = \frac{nN}{2}$; $B = \frac{N}{2}$. Увеличение размерности гиперкуба на единицу ведет к удвоению числа его узлов, увеличению степени узлов и диаметра сети на единицу.

Обмен сообщениями в гиперкубе базируется на двоичном представлении номеров узлов. Нумерация узлов производится так, что для любой пары смежных узлов двоичное представление номеров этих узлов отличается только в одной позиции. В силу сказанного, узлы 0010 и 0110 — соседи, а узлы 0110 и 0101 таковыми не являются. Простейший способ нумерации узлов при создании n-мерного гиперкуба из двух $(n - 1)$-мерных показан на рис. 12.17, *г*. При копировании $(n - 1)$-мерного гиперкуба и соединении его с исходным $(n - 1)$-мерным гиперкубом необходимо, чтобы соединяемые узлы имели одинаковые номера. Далее к номерам узлов исходного гиперкуба слева добавляется бит, равный 0, а к номерам узлов копии — единичный бит.

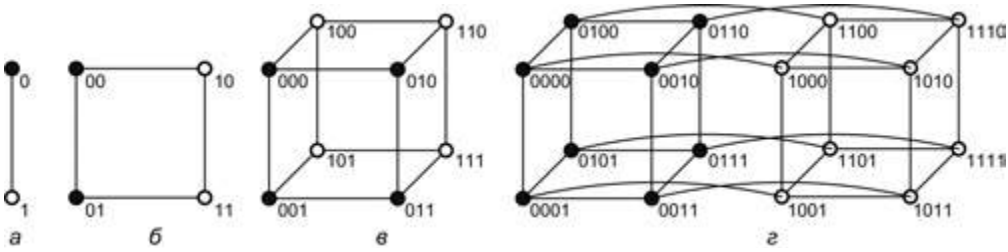


Рис. 12.17. Топология гиперкуба: *а* — одномерная; *б* — двумерная; *в* — трехмерная; *г* — четырехмерная

Номера узлов являются основой маршрутизации сообщений в гиперкубе. Такие номера в n-мерном гиперкубе состоят из n битов, а пересылка сообщения из узла A в узел B выполняется за n шагов. На каждом шаге узел может либо сохранить сообщение и не пересылать его дальше до следующего шага, либо отправить его дальше

по одной из линий. На шаге i узел, хранящий сообщение, сравнивает i-й бит своего собственного номера с i-м битом номера узла назначения. Если они совпадают, продвижение сообщения приостанавливается до следующего шага, в противном случае сообщение передается вдоль линии i-го измерения. Линией i-го измерения считается та, которая была добавлена на этапе построения i-мерного гиперкуба из двух $(i-1)$-мерных.

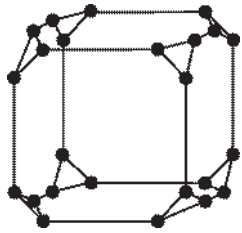


Рис. 12.18. Куб из циклически соединенных узлов третьего порядка

Создание гиперкуба при большом числе процессоров требует увеличения степени узлов, что сопряжено с большими техническими проблемами. Компромиссное решение, несколько увеличивающее диаметр сети при сохранении базовой структуры, представляет собой *куб из циклически соединенных узлов* (рис. 12.18). Здесь степень узла равна трем при любом размере сети.

Данные по рассмотренным статическим топологиям сведены в табл. 12.1.

Таблица 12.1. Характеристики сетей со статической топологией

Топология	Диаметр	Степень узла	Число связей	Ширина бисекции
Полносвязная	1	$N-1$	$\frac{N(N-1)}{2}$	$\frac{N^2}{4}$
Звезда	2	$N-1$	$N-1$	1
Двоичное дерево	$2\log_2\left(\frac{N+1}{2}\right)$	3	$N-1$	1
Линейный массив	$N-1$	2	$N-1$	1
Кольцо	$\left\lfloor \frac{N}{2} \right\rfloor$	2	N	2
Двухмерная решетка	$2(\sqrt{N}-1)$	4	$2(N-\sqrt{N})$	$\sqrt{N}$
Двухмерный тор	$2\left\lfloor \frac{\sqrt{N}}{2} \right\rfloor$	4	$2N$	$2\sqrt{N}$
Гиперкуб	$\log_2 N$	$\log_2 N$	$\frac{N\log_2 N}{2}$	$\frac{N}{2}$

Динамические топологии

В *динамической топологии сети* соединение узлов обеспечивается электронными ключами, варьируя установки которых можно менять топологию сети. В отличие от ранее рассмотренных топологий, где роль узлов играют сами объекты информационного обмена, в узлах динамических сетей располагаются коммутирующие элементы, а устройства, обменивающиеся сообщениями (терминалы), подключаются к входам и выходам этой сети. В роли терминалов могут выступать процессоры или процессоры и банки памяти.

Можно выделить два больших класса сетей с динамической топологией: *сети на основе шины* и *сети на основе коммутаторов*.

Одношинная топология

Сети на основе шины — наиболее простой и дешевый вид динамических сетей. Рисунок 12.19 иллюстрирует систему на базе *одношинной топологии*.

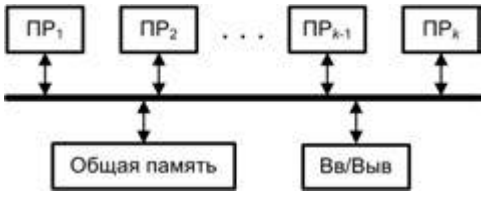


Рис. 12.19. Пример системы с одной шиной

В общем случае такая система состоит из k процессоров, каждый из которых подключен к общей совместно используемой шине. Все процессоры взаимодействуют через общую память. Узлы при одношинной топологии имеют степень 1 ($d = 1$). В каждый момент времени обмен сообщениями может вести только одна пара узлов, то есть на период передачи сообщения шину можно рассматривать как сеть, состоящую из двух узлов, в силу чего ее диаметр всегда равен единице ($D = 1$). Также единице равна и ширина бисекции (B), поскольку топология допускает одновременную передачу только одного сообщения. Одношинная конфигурация может быть полезной, когда число узлов невелико, то есть когда трафик шины мал по сравнению с ее пропускной способностью. Размер подобных систем изменяется в диапазоне от 2 до 50 процессоров. Одношинную архитектуру часто используют для объединения нескольких узлов в группу (кластер), после чего из таких кластеров образуют сеть на базе других видов топологии.

Многошинная топология

Естественным развитием идеи одношинной топологии является *многошинная топология*. В вычислительной системе с многошинной топологией для объединения множества процессоров и множества модулей памяти используются несколько параллельных шин. Процессоры ВС, как правило, подключаются к каждой из шин, а при подключении модулей памяти может использоваться одна из четырех схем:

- с подключением ко всем шинам;
- с подключением к одной из шин;
- с подключением к части шин;
- с подключением по классам.

Перечисленные схемы иллюстрирует рис. 12.20. В варианте (*а*) все банки памяти подключаются к каждой из шин. В схеме (*б*) каждый банк памяти подключен лишь к определенной шине. В случае (*в*) каждый банк памяти подключен к определенному подмножеству шин. Наконец, в варианте (*г*) банки памяти группируются в несколько классов, а каждый класс, в свою очередь, имеет соединение с определенным подмножеством шин.

Такая топология вполне пригодна для высокопроизводительных ВС. Диаметр сети по-прежнему равен 1, в то время как пропускная способность возрастает пропорционально числу шин. По сравнению с одношинной архитектурой управление сетью с несколькими шинами сложнее из-за необходимости предотвращения конфликтов, возникающих, когда в парах узлов, обменивающихся по разным шинам, присутствует общий узел. Кроме того, с увеличением степени узлов сложнее становится их техническая реализация.

Блокирующие, неблокирующие и реконфигурируемые топологии

В динамических сетях на основе коммутаторов тракты между терминальными узлами формируются с использованием коммутирующих устройств. Минимальным требованием к сети с коммутацией является поддержка соединения любого входа с любым выходом. Для этого в сети с n входами и n выходами система ключей обязана предоставить $n!$ вариантов коммутации входов и выходов. Проблема усложняется, когда сеть должна обеспечивать одновременную передачу данных между многими парами терминальных узлов, причем так, чтобы не возникали конфликты (блокировки) из-за передачи данных через одни и те же коммутирующие элементы в одно и то же время. Подобные топологии должны поддерживать n^n вариантов перестановок. С этих позиций все топологии сетей с коммутацией разделяются на три типа: блокирующие, неблокирующие и реконфигурируемые.

В *блокирующих сетях* установление соединения между свободным входом и свободным выходом возможно не всегда, поскольку это может вызвать конфликт с другим уже установленным соединением из-за наличия в этих соединениях общих коммутаторов. Обычно для минимизации общего числа коммутаторов в сети предполагают лишь единственный путь между каждой парой вход/выход. В то же время для уменьшения числа конфликтов и повышения отказоустойчивости в сети могут предусматриваться множественные пути. Такие блокирующие сети известны как *сети с обходными путями*.

В *неблокирующих сетях* любой входной порт может быть подключен к любому свободному выходу без влияния на уже существующие соединения. В рамках этой группы различают сети строго неблокирующие и неблокирующие в широком смысле.

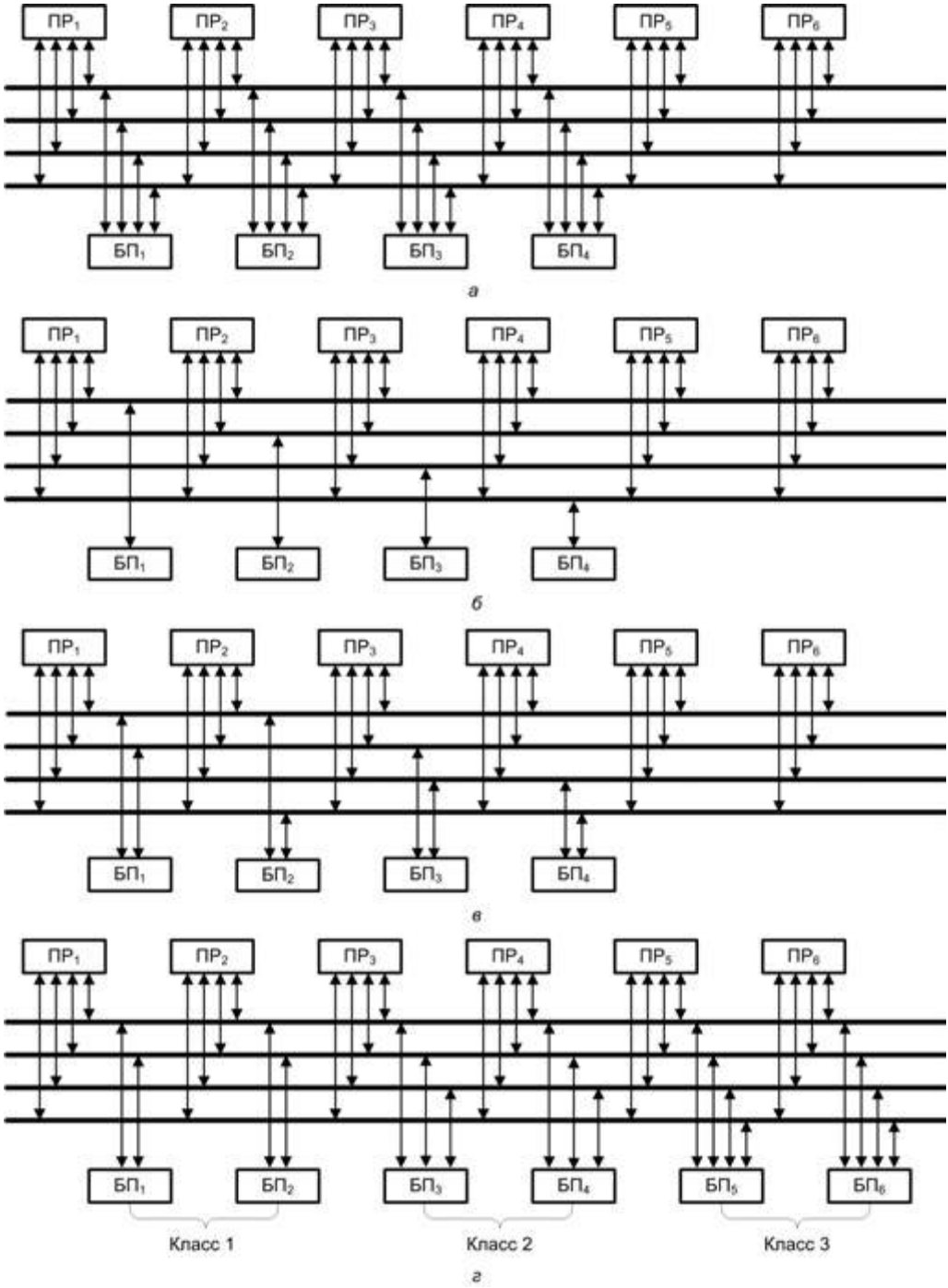


Рис. 12.20. Множественные шины с подключением банков памяти: *а* — ко всем шинам; *б* — к одной из шин; *в* — к части шин; *г* — по классам системы с одной шиной

В *строго неблокирующих сетях* возникновение блокировок принципиально невозможно в силу примененной топологии. *Неблокирующими в широком смысле* называют топологии, в которых конфликты при любых соединениях не возникают только при соблюдении определенного алгоритма маршрутизации. В любом случае такие сети, как правило, являются многоступенчатыми. В многоступенчатых динамических сетях коммутаторы группируются в так называемые *ступени коммутации*. Наличие более чем одной ступени коммутации позволяет обеспечить множественность путей между любыми парами входов и выходов.

В *реконфигурируемых сетях* возможна такая установка коммутаторов сети, при которой допустима одновременная передача сообщений между множеством пар входных и выходных узлов без взаимного блокирования сообщений. Здесь, однако, следует учитывать, что все тракты передачи сообщений должны формироваться одновременно. Если новый тракт формируется при уже функционирующих других трактах, неблокируемость без реконфигурации этих функционирующих трактов не гарантируется. В реконфигурируемых сетях обеспечивается множественность путей между каждой парой вход/выход, однако количество таких путей по сравнению с неблокирующими сетями меньше, но при этом меньше и стоимость сети. Сети данного типа были предложены для построения матричных ВС. В других типах многопроцессорных ВС реконфигурация оказывается более сложной, поскольку процессоры обращаются к сети асинхронно, и в этом случае рассматриваемые сети ведут себя как блокирующие.

Топология полносвязной коммутационной матрицы («кроссбар»)

Топология полносвязной коммутационной матрицы (кроссбар) представляет собой пример одноступенчатой динамической сети. Смысл, вкладываемый в понятие «полносвязная матрица», заключается в том, что любой входной порт может быть связан с любым выходным портом. Не совсем официальный термин «кроссбар», который будет применяться в дальнейшем для обозначения данной топологии, берет свое начало с самых первых телефонных коммутаторов, где с помощью штекеров входные линии замыкались на выходные, образуя соединение.

Коммутатор типа кроссбар представляет собой матрицу, строки которой образованы проводниками, связанными с входами, а столбцы — с выходами. На пересечении линий и столбцов матрицы находятся управляемые переключатели, которые могут либо замыкать соединение строки на столбец, либо, наоборот, размыкать его. Для управления переключателями, находящимися на пересечении строк и столбцов такой коммутационной матрицы, кроссбар должен иметь контроллер, который управляет состоянием переключателей на основе анализа информации об адресе назначения.

Кроссбар $n \times l$ способен соединить n входных и l выходных терминальных узлов (процессоров, банков памяти и т. п.), причем так, что обмен информацией одновременно могут вести $\min(n, l)$ пар терминальных узлов, и конфликты при этом не возникают. Новое соединение может быть установлено в любой момент при условии, что входной и выходной порты свободны.

Главное достоинство топологии состоит том, что сеть получается неблокирующей и обеспечивает меньшую задержку в передаче сообщений по сравнению с другими топологиями, поскольку любой путь содержит только один ключ. Тем не менее из-за значительного числа ключей в кроссбаре ($n \times l$) использование такой топологии в больших сетях становится непрактичным, хотя это достаточно хороший выбор для малых сетей.

Кроссбары могут использоваться в коммутаторах, ориентированных как на коммутацию пакетов, так и на коммутацию соединений. Они пригодны для применения в синхронных и асинхронных сетях.

Сети кроссбар традиционно используются в небольших ВС с разделяемой памятью, где все процессоры могут одновременно обращаться к банкам памяти при условии, что каждый процессор работает со своим банком памяти. В качестве примера рассмотрим сеть кроссбар, приведенную на рис. 12.21. Сеть содержит переключающий элемент в каждой из 64 точек пересечения горизонтальных и вертикальных линий. На рисунке показано состояние кроссбара при одновременном соединении каждого из 8 процессоров ($ПР_i$) со своим банком памяти ($БП_{8-i+1}$). Показаны также два возможных состояния переключающего элемента — это прямое соединение (см. 12.21, *а*) и диагональное соединение (см. 12.21, *б*).

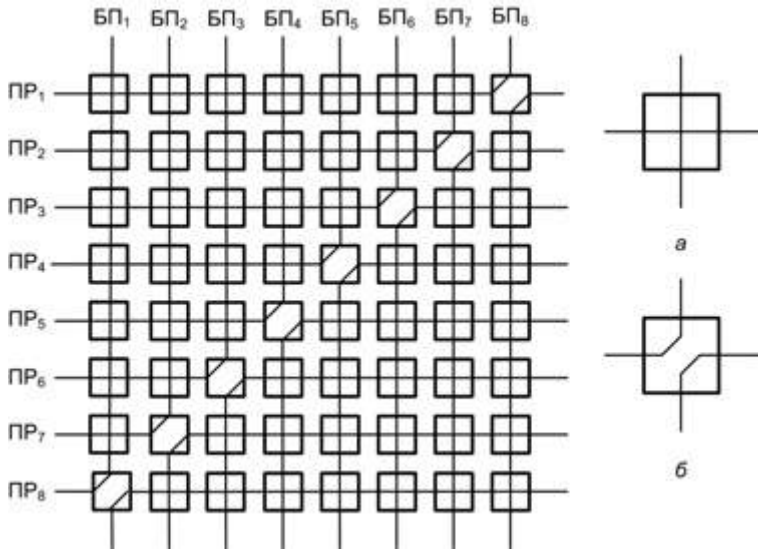


Рис. 12.21. Сеть кроссбар 8 × 8: *а* — прямое соединение в переключающем элементе; *б* — диагональное соединение в переключающем элементе

Когда два или более процессоров соперничают за один и тот же банк памяти, срабатывает схема арбитража, разрешающая доступ к банку лишь одному из процессоров, в то время как остальные процессоры вынуждены ожидать. В целом же схема арбитража в кроссбаре может быть менее сложной, чем в случае шины, поскольку конфликты в кроссбаре являются скорее исключением, чем правилом.

При $l = n$ кроссбар называют *полным*. Полный кроссбар на n входов и n выходов содержит n^2 ключей. Диаметр кроссбара равен 1, ширина бисекции — $\frac{n}{2}$. Большие кроссбары можно реализовать путем разбиения на меньшие. Так, полный кроссбар $n \times n$ можно построить из $\frac{n}{k} \times \frac{n}{k}$ кроссбаров размерности $k \times k$.

Топология используется для организации соединений в некоторых серийно выпускаемых вычислительных системах, например в вычислительной системе Earth Simulator фирмы NEC используется кроссбар 639×639.

Коммутирующие элементы сетей с динамической топологией

Поскольку последующие рассматриваемые топологии относятся к многоступенчатым, сначала необходимо определить типы коммутирующих элементов, применяемых в ступенях коммутации таких сетей. По этому признаку различают:

- сети на основе полносвязной коммутационной матрицы;
- сети на основе базового коммутирующего элемента.

В сетях, относящихся к первой группе, в качестве базового коммутирующего элемента используется кроссбар $n \times l$. Для второй категории роль коммутирующего элемента играет «полный кроссбар» 2×2. Потенциально такой коммутатор управляется четырехразрядным двоичным кодом и обеспечивает 16 вариантов коммутации, из которых полезными можно считать 12. На практике же обычно задействуют только четыре возможных состояния кроссбара 2×2, которые определяются двухразрядным управляющим кодом (рис. 12.22). Подобный кроссбар называют *базовым коммутирующим элементом* (БКЭ) или β-*элементом*. Первые два состояния БКЭ являются основными: в них входная информация может транслироваться на выходы прямо либо перекрестно. Два следующих состояния предназначены для широковещательного режима, когда сообщение от одного узла одновременно транслируется на все подключенные к нему прочие узлы. Широковещательный режим используется редко. Сигналы на переключение БКЭ в определенное состояние могут формироваться устройством управления сетью. В более сложном варианте β-элемента эти сигналы формируются внутри него, исходя из адресов узла-источника и узла-получателя, содержащихся в заголовке сообщения.

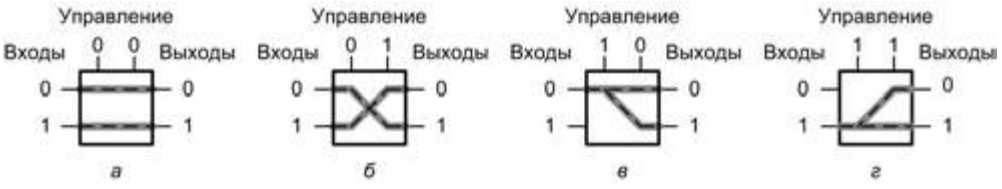


Рис. 12.22. Состояния β-элемента: *а* — прямое; *б* — перекрестное; *в* — широковещание с верхнего входа; *г* — широковещание с нижнего входа

Структура β-элемента показана на рис. 12.23.

Рис. 12.23. Структура β-элемента

Выбор в пользу того или иного варианта коммутации входных сообщений (пакетов) осуществляется управляющей логикой β-элемента. Назначение этой схемы состоит в анализе заголовков пакетов и в принятии решения о требуемом состоянии коммутационной матрицы. Результат решения фиксируется в регистре. В зависимости от принятого решения коммутационная матрица может обеспечивать либо прямое соединение между входами и выходами, либо перекрестное (кроссоверное). Элементы задержки служат для синхронизации процессов принятия решения и пересылки пакетов с входов на выходы.

Сложность β-элемента находится в зависимости от логики принятия решения. В ряде архитектур БКЭ их состояние определяется только битом активности пакета. В иных архитектурах используются адреса источника и получателя данных, хранящиеся в заголовке пакета, что может потребовать поддержания в БКЭ специальных таблиц. Тем не менее во всех своих вариантах β-элементы достаточно просты, что позволяет реализовать их на базе интегральных микросхем.

Многоступенчатые динамические сети

Многоступенчатые динамические сети (МДС) разрабатывались как средство для устранения ограничений, свойственных одноступенчатым сетям, при сохранении стоимости сети в допустимых пределах. В частности, наличие дополнительных ступеней позволяет одновременно иметь несколько альтернативных трактов между любой парой узлов.

Многоступенчатые динамические сети соединяют входные устройства с выходными посредством нескольких ступеней коммутации, построенных из базовых коммутирующих элементов, либо кроссбаров большей размерности. Количество ступеней и способ их соединения определяют возможности маршрутизации в сети.

Существует много способов соединения смежных ступеней МДС. На рис. 12.24, *а* показана обобщенная многоступенчатая сеть с n входами и l выходами. МДС состоит из g ступеней $G_0 - G_{g-1}$. Как показано на рис. 12.24, *б*, каждая ступень, скажем G_i, содержит w_i коммутаторов размерности $a_{i,j} \times b_{i,j}$, где $1 \le j \le w_i$. Таким образом, ступень G_i имеет p_i входов и q_i выходов, где

$$p_i = \sum_{j=1}^{w_i} a_{i,j} \quad \text{и} \quad q_i = \sum_{j=1}^{w_i} b_{i,j}.$$

Соединение между двумя смежными ступенями G_{i-1} и G_i, обозначенное C_i, определяется функцией маршрутизации для $p_i = q_{i-1}$ линий связи, где $p_0 = n$ и $q_{g-1} = l$. Таким образом, МДС может быть представлена в виде

$$C_0(N)G_0(w_0)C_1(p_1)G_1(w_1)...G_{g-1}(w_{g-1})C_g(M).$$

Функция маршрутизации $C_i(p_i)$ определяет, каким образом p_i линий должны быть использованы между $q_{i-1} = p_i$ выходами ступени G_{i-1} и p_i входами ступени G_i. Различные функции маршрутизации определяют различие в характеристиках и топологических свойствах МДС.

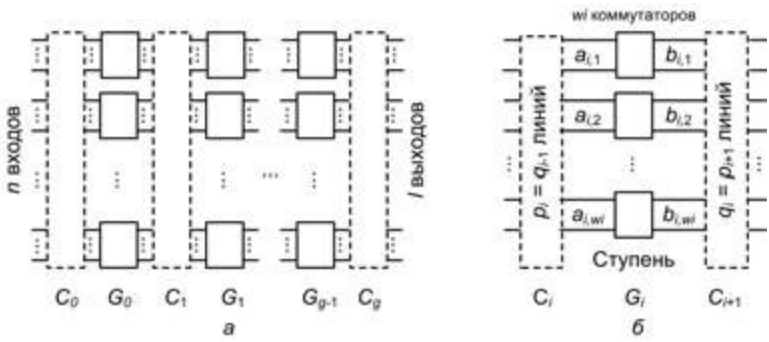


Рис. 12.24. Многоступенчатая сеть с *n* входами, *l* выходами и *g* ступенями: *а* — общая структура; *б* — детальное представление ступени G_i

На практике все коммутаторы должны быть идентичными, чтобы снизить стоимость разработки. Если входы и выходы сети коммутирующих элементов разделены, сеть называют *двухсторонней*. При совмещенных входах и выходах сеть является *односторонней*.

В зависимости от типа каналов связи и коммутаторов динамические многоступенчатые сети можно разделить на два класса: *однонаправленные* или *двунаправленные*. В однонаправленных сетях используются однонаправленные каналы и коммутаторы. В двунаправленных сетях каналы и коммутаторы — двунаправленные.

Блокирующие многоступенчатые сети

Среди блокирующих многоступенчатых сетей наибольшее распространение получили баньяноподобные сети [83]. *Баньян-сети* образуют обширное семейство МДС, в которых между любым входом и любым выходом возможен только единственный путь [112]. Название свое сети получили из-за того, что их конфигурация напоминает воздушные корни дерева баньян (индийской смоковницы).

Среди баньян-сетей наибольшее распространение получили так называемые сети «Дельта», предложенные Пателом в 1981 году [127]. Дельта-сеть на *n* входов и *n* выходов имеет $\log_2 n$ ступеней коммутации, каждая из которых состоит из $\frac{n}{2}$ базовых коммутирующих элементов. На рис. 12.25 показаны несколько вариантов дельта-сетей.

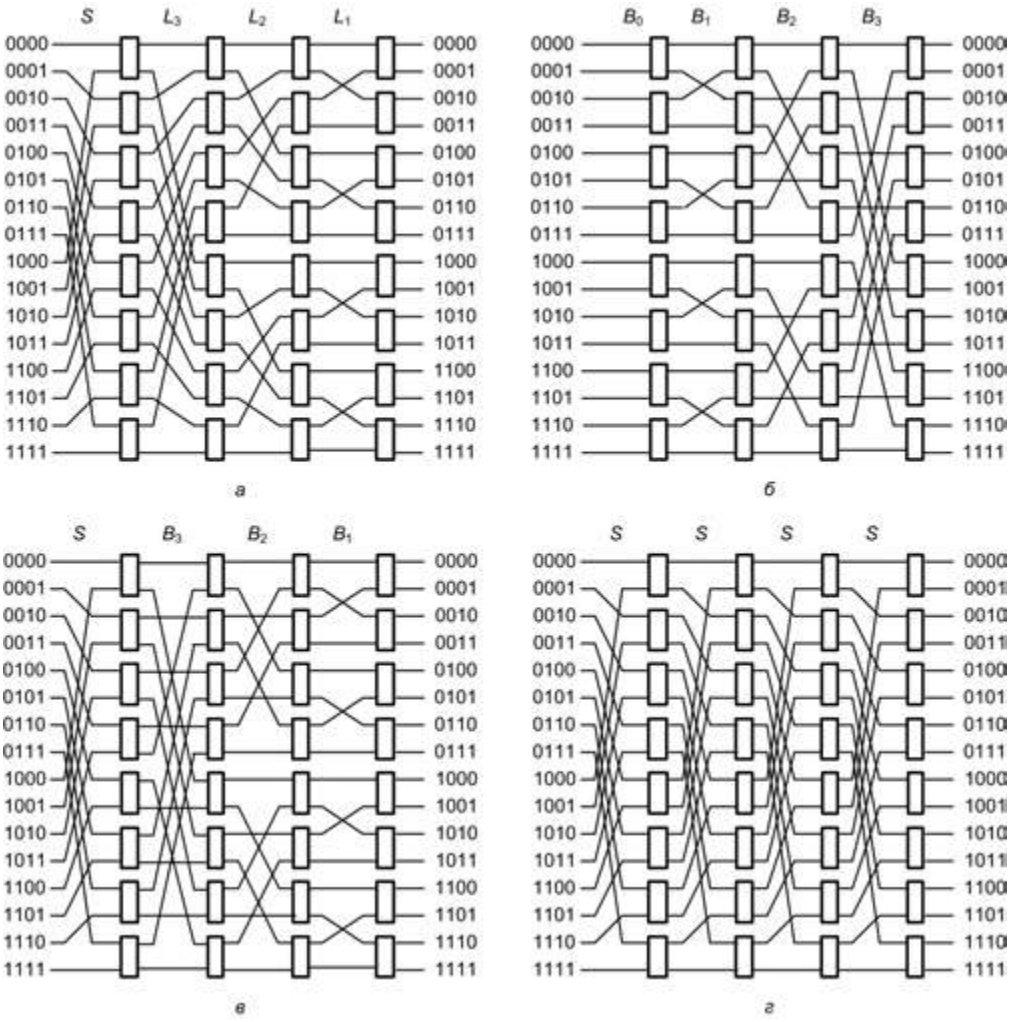


Рис. 12.25. Примеры сетей класса Дельта размерности 16 × 16 с топологией: *а* — базисной линии; *б* — баттерфляй; *в* — обобщенного куба; *г* — Омега

Представленные дельта-топологии [63, 106, 145] различаются тем, какие функции маршрутизации использованы между ступенями сети и на ее входе. Эти функции указаны в верхней части каждой схемы. Отметим, что по отношению к произвольному трафику все сети обеспечивают эквивалентную производительность.

Существенным достоинством рассматриваемых сетей, определившим их популярность, является свойство *самомаршрутизации*. Чтобы доставить сообщение к узлу-получателю, используется адрес этого узла, содержащийся в заголовке передаваемого пакета. Этот адрес не только определяет маршрут сообщения к нужному узлу, но и используется для управления прохождением сообщения по этому маршруту. Число битов в двоичном представлении адреса равно числу ступеней сети, причем

каждый бит соответствует определенной ступени: старший бит — нулевой (левой) ступени, младший бит — последней (правой) ступени.

Каждый БКЭ, куда попадает пакет, просматривает один бит адреса (соответствующий ступени сети, где этот БКЭ расположен), и в зависимости от его значения направляет сообщение на верхний или нижний выход. Если значение бита равно 0, то сообщение пропускается через верхний выход БКЭ, а при единичном значении — через нижний.

Проиллюстрируем этот алгоритм на примере сети «Омега» 8 × 8 (рис. 12.26, *а*). На рис. 12.26, *а* показан процесс маршрутизации сообщения с входного терминала 2_{10} (010_2) на выходной терминал с номером 6_{10} (110_2). Старший бит адреса назначения управляет коммутаторами нулевой ступени, средний бит — первой ступени, младший бит — второй ступени.

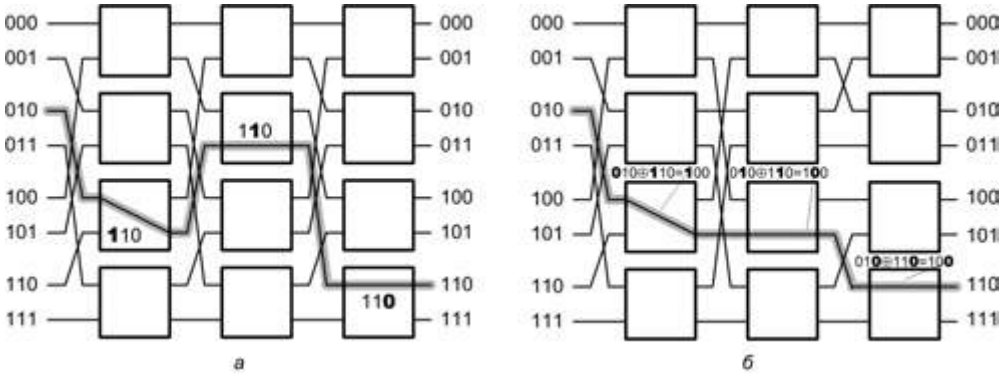


Рис. 12.26. Самомаршрутизация в дельта-сетях размерности 8 × 8: *а* — по адресу получателя; *б* — по адресам источника и получателя

Возможен и иной алгоритм самомаршрутизации, но в сообщении помимо адреса получателя необходимо передавать и адрес источника. Эти адреса в двоичном представлении имеют длину m битов (m — число ступеней). Самомаршрутизация реализуется за счет следующего алгоритма установки β-элементов сети. Состояние, в которое переключается БКЭ на i-й ступени, определяется с помощью операции сложения по модулю 2 значений i-го бита в адресах входного ($a_{m-1} \ldots a_0$) и выходного ($b_{m-1} \ldots b_0$) терминальных узлов. Если $a_i \oplus b_i = 0$, то БКЭ, расположенный на $(m - i - 1)$-й ступени сети, обеспечивает прямую связь входа с выходом, а при $a_i \oplus b_i = 1$ — перекрестное соединение. На рис. 12.26, *б* показан процесс прохождения сообщения по сети с топологией обобщенного куба размерности 8 × 8 от входного терминала 2_{10} (010_2) к выходному терминалу 6_{10} (110_2).

В то же время не следует забывать, что рассматриваемый класс топологий относится к блокирующим. Количество соединений, обеспечиваемых дельта-сетью, равно $n^{\frac{n}{2}}$, что гораздо меньше, чем $n!$, то есть топология также является блокирующей. Так, при $n = 8$ процент комбинаций, возможных в омега-сети, по отношению к

потенциально допустимому числу комбинаций составляет $\frac{8^4}{8!} = \frac{4096}{40320} = 0{,}1016$ или 10,16%. Блокирование в омега-сети иллюстрирует рис. 12.27, *а*, где показана ситуация одновременной передачи сообщений с входного терминала 0_{10} (000_2) на выходной терминал 1_{10} (001_2) и с входного терминала 4_{10} (100_2) на выходной терминал 2_{10} (010_2). Как видно, данная ситуация предполагает такую коммутацию в левом верхнем БКЭ, которая для базовых коммутирующих элементов невозможна. Аналогичная ситуация возникает в другом варианте дельта-сети (рис. 12.27, *б*) при попытке одновременной передачи пакетов с входного терминала 1_{10} (001_2) на выходной терминал 5_{10} (101_2) и с входного терминала 3_{10} (011_2) на выходной терминал 4_{10} (100_2).

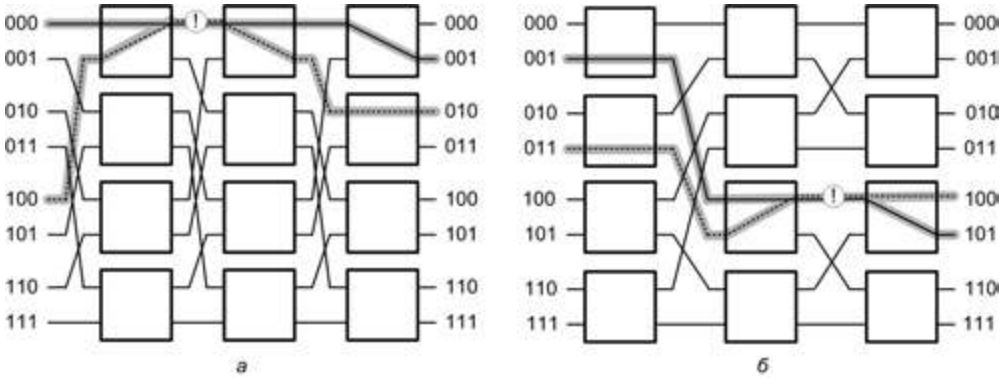


Рис. 12.27. Иллюстрация блокировки в дельта-сетях размерности 8 × 8: *а* — «Омега»; *б* — базисной линии

Топология «Баньян» весьма популярна из-за того, что коммутация обеспечивается простыми БКЭ, работающими с одинаковой скоростью, сообщения передаются параллельно. Кроме того, большие сети могут быть построены из стандартных модулей меньшего размера.

Неблокирующие многоступенчатые сети

Топология Клоза

В 1953 году Клоз показал, что многоступенчатая сеть на основе элементов типа кроссбар, содержащая не менее трех ступеней, может обладать характеристиками неблокирующей сети [54, 67].

Сеть Клоза с тремя ступенями, показанная на рис. 12.28, содержит r_1 кроссбаров во входной ступени, m кроссбаров в промежуточной ступени и r_2 кроссбаров в выходной ступени. У каждого коммутатора входной ступени есть n_1 входов и m выходов — по одному выходу на каждый кроссбар промежуточной ступени. Коммутаторы промежуточной ступени имеют r_1 входов по числу кроссбаров входной ступени и r_2 выходов, что соответствует количеству переключателей в выходной ступени

сети. Выходная ступень сети строится из кроссбаров с m входами и n_2 выходами. Отсюда числа n_1, n_2, r_1, r_2 и m полностью определяют сеть. Число входов сети $N = r_1 n_1$, а выходов — $M = r_2 n_2$.

Связи внутри составного коммутатора организованы по следующим правилам:

- k-й выход i-го входного коммутатора соединен с i-м входом k-го промежуточного коммутатора;
- k-й вход j-го выходного коммутатора соединен с j-м выходом k-го промежуточного коммутатора.

Каждый модуль первой и третьей ступеней сети соединен с каждым модулем второй ее ступени.

Хотя в рассматриваемой топологии обеспечивается путь от любого входа к любому выходу, ответ на вопрос, будет ли сеть неблокирующей, зависит от числа промежуточных звеньев. Клоз доказал, что подобная сеть является неблокирующей, если количество кроссбаров в промежуточной ступени m удовлетворяет условию: $m \geqslant n_1 + n_2 - 1$. Если $n_1 = n_2 = n$, то матричные переключатели в промежуточной ступени представляют собой «полный кроссбар» и критерий неблокируемости приобретает вид: $m \geqslant 2n - 1$. При условии $m \geqslant n_2$ сеть Клоша можно отнести к неблокирующим сетям с реконфигурацией. Во всех остальных соотношениях между m, n_1 и n_2 данная топология становится блокирующей.

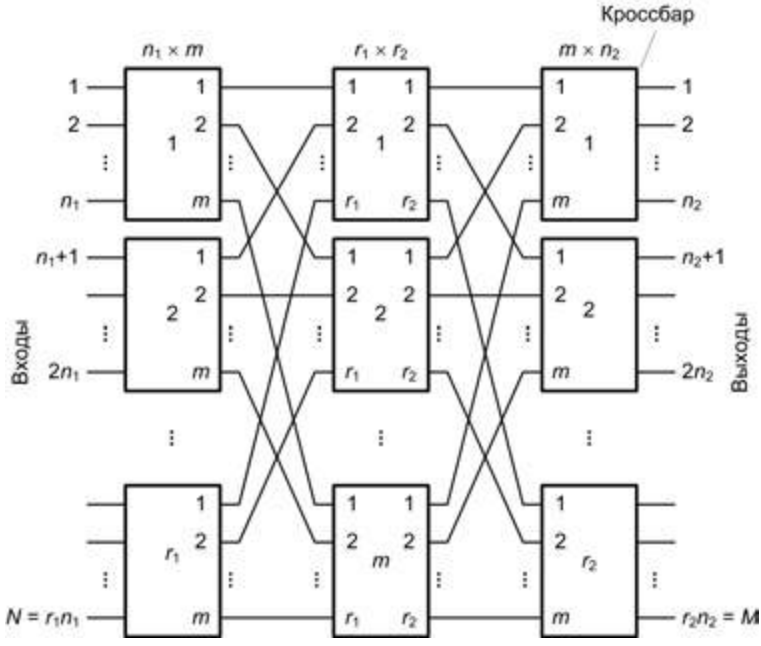


Рис. 12.28. Трехступенчатая сеть с топологией Клоза

Вычислительные системы, в которых соединения реализованы согласно топологии Клоза, выпускают многие фирмы, в частности Fujitsu (FETEX-150), Nippon Electric Company (ATOM), Hitachi. Частный случай сети Клоза при $n_1 = r_1 = r_2 = n_2$ называется сетью «Мемфис». Топология «Мемфис» нашла применение в вычислительной системе GF-11 фирмы IBM.

Сети Бэтчера-Баньяна

Как уже отмечалось, баньян-топология относится к блокирующим, то есть может приводить к конфликтам, если два сообщения пытаются выйти на один и тот же выход БКЭ. В случае коллизии одно сообщение передается, а другое — отбрасывается. Однако если сообщения, поступающие на входы такой сети, упорядочены определенным образом, то конфликтов не происходит. В частности, баньян-сети $n \times n$ в состоянии передавать одновременно вплоть до n пакетов (при условии, что они предназначены разным получателям), если адреса назначения в пакетах, поступающих на входы сети с возрастающими номерами, также упорядочены в порядке возрастания. Такое упорядочение можно реализовать, если перед баньян-сетью поместить так называемую сортирующую сеть, например сеть *Бэтчера* [52].

Сеть *Бэтчера* также состоит из стандартных БКЭ, но они работают несколько по-иному, чем аналогичные элементы баньян-сети. При получении двух пакетов коммутирующий элемент сравнивает хранящиеся в них адреса назначения и направляет пакет с большим адресом по стрелке, а пакет с меньшим адресом — в противоположном направлении. Если имеется только один пакет, то он посылается в направлении, противоположном указанию стрелки. Данный алгоритм иллюстрирует рис. 12.29, в левой части которого показан порядок сортировки пакетов, одновременно направляемых на 4 выходных терминала.

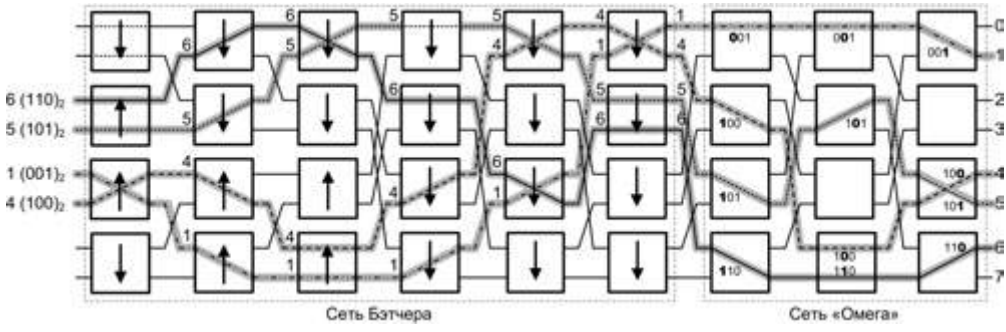


Рис. 12.29. Сеть Бэтчера–Баньяна

С ростом числа входов n количество БКЭ в сети растет в соответствии с зависимостью $n \log_2 n$. При получении k сообщений сеть Бэтчера упорядочивает их в порядке возрастания адресов назначения и выдает на свои первые k выходов. Если теперь выходы сети Бэтчера по схеме полного тасования соединить с входами баньян-сети типа, то образовавшаяся сеть становится неблокирующей. Это относится ко всем вариантам баньян-сетей.

Реконфигурируемые многоступенчатые сети

Топология Бенеша

Рассматриваемая топология [109, 123] относится к типу реконфигурируемых сетей. Это означает, что возможна такая установка коммутаторов сети, при которой одновременная передача сообщений между множеством пар входных и выходных узлов будет происходить без взаимного блокирования. Реконфигурирование предполагает разрыв и переустановку всех соединений [109, 123].

В качестве коммутаторов в *сетях Бенеша* используются β-элементы. В сети $n \times n$ общее количество БКЭ равно $n \log_2 n$. Ввиду того, что рассматриваемые сети не обладают свойством самомаршрутизации, то есть для коммутации маршрута недостаточно информации, содержащейся во входном пакете, для определения состояния всех β-элементов необходим специальный контроллер.

Сущность реконфигурирования и достигаемый при этом эффект иллюстрирует рис. 12.30. На рис. 12.30, *а* показана топология сети Бенеша, обеспечивающая одновременную пересылку трех сообщений: с входа 0_{10} (000_2) на выход 4_{10} (100_2), входа 1_{10} (001_2) на выход 0_{10} (000_2) и входа 2_{10} (010_2) на выход 2_{10} (010_2). Как видно, блокировки сообщений не происходит. При попытке одновременно с этим передать пакет с входа 3_{10} (011_2) на выход 1_{10} (001_2) передача блокируется. Однако блокировки можно избежать, если изменить топологию сети. Рисунок 12.30, *б* иллюстрирует состояние сети после такой реконфигурации. Благодаря изменению трактов прохождения всех четырех сообщений конфликт не возникает.

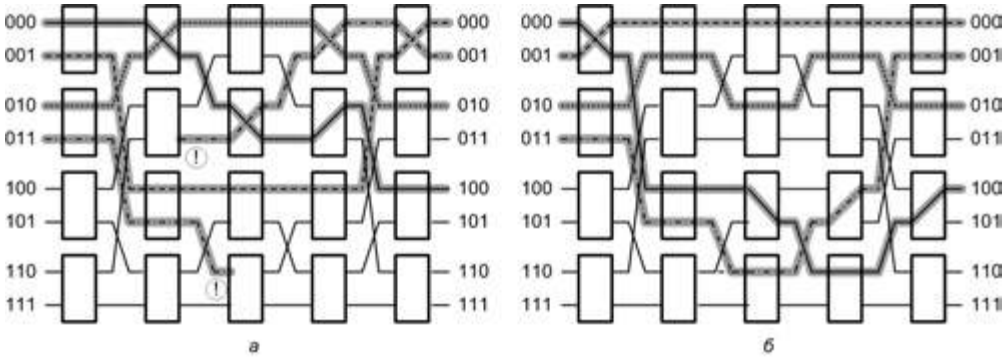


Рис. 12.30. Сеть Бенеша: *а* — до реконфигурации; *б* — после реконфигурации

Обратим внимание на то, что все тракты передачи сообщений должны формироваться одновременно. Если новый тракт формируется при уже функционирующих других трактах, неблокируемость без реконфигурирования «старых» трактов не гарантируется.

Сеть Бенеша может рассматриваться как частный случай сети Клоза, для которой выполняется условие $m \geqslant n$. В этом случае, как доказал Бенеш, сеть Клоза можно отнести к неблокирующим сетям с реконфигурацией.

С добавлением к такой сети дополнительной ступени БКЭ число возможных маршрутов удваивается. Дополнительные пути позволяют изменять трафик сообщения с целью устранения конфликтов. Сеть Бенеша с n входами и n выходами имеет симметричную структуру, в каждой половине которой (верхней и нижней) между входными и выходными БКЭ расположена такая же сеть Бенеша, но с $\frac{n}{2}$ входами и $\frac{n}{2}$ выходами.

Контрольные вопросы

1. Прокомментируйте классификацию сетей по топологии, а также стратегиям синхронизации, коммутации и управления.
2. Дайте развернутую характеристику метрик, описывающих соединения сети.
3. Сформулируйте достоинства и недостатки наиболее известных функций маршрутизации данных.
4. Обоснуйте достоинства и недостатки линейной топологии сети.
5. Дайте характеристику плюсов и минусов кольцевой топологии сети. Какие варианты этой топологии практически используются?
6. Проведите сравнительный анализ звездообразной и древовидной топологий сети.
7. Выполните сравнительный анализ известных вариантов решетчатой топологии сети.
8. Поясните идею динамической топологии. Охарактеризуйте оба класса сетей с динамической топологией.
9. В чем суть деления сетей на основе коммутаторов на блокирующие, неблокирующие и реконфигурируемые?
10. Проведите сравнительный анализ одношинной и многошинной топологий динамических сетей, акцентируя их сильные и слабые стороны.
11. Дайте развернутую характеристику коммутирующих элементов для сетей с динамической топологией.
12. Какая цель преследуется при использовании многоступенчатых динамических сетей? Как классифицируются эти сети?
13. Сравните популярные разновидности баньян-сетей: «Омега», «Дельта».
14. Дайте развернутое объяснение отличий топологии Клоза от баньян-сетей. Можно ли найти у них сходные черты, и если можно, то какие?
15. Какую проблему баньян-сетей позволяет решить сеть Бэтчера-Баньяна? Каким именно образом? Ответ обоснуйте, приведя конкретный пример.
16. Можно ли сказать, что сеть Бенеша занимает промежуточное положение между баньян-сетью и сетью Клоза? Ответ обоснуйте.

Глава 13

Вычислительные системы класса SIMD

SIMD-системы были первыми вычислительными системами, состоящими из большого числа процессоров, они же одними из первых преодолели барьер производительности порядка GFLOPS[1]. Согласно классификации Флинна, к классу SIMD относятся ВС, где множество элементов данных подвергается параллельной, но однотипной обработке. SIMD-системы во многом похожи на классические фоннеймановские ВМ: в них также имеется одно устройство управления, обеспечивающее последовательное выполнение команд программы. Различие касается стадии выполнения, когда общая команда транслируется множеству процессоров (в простейшем случае — операционному устройству), каждый из которых обрабатывает свои данные.

Ранее уже отмечалась нечеткость классификации Флинна, из-за чего разные типы ВС могут быть отнесены к тому или иному классу. Тем не менее в настоящее время принято считать, что класс SIMD составляют векторные, матричные, ассоциатив-

[1] FLOPS (flop/s, Floating point Operations Per Second) — величина, используемая для измерения производительности ВМ и ВС, она показывает, сколько операций с плавающей запятой в секунду выполняет данная система. Поскольку современные ВМ обладают высоким уровнем производительности, более распространены производные величины от FLOPS, образуемые путем использования стандартных приставок. Примеры: GFLOPS = 10^9FLOPS = = 1 миллиард оп. ПЗ/с, TFLOPS = 10^{12}FLOPS = 1 триллион оп. ПЗ/с, PFLOPS = 10^{15}FLOPS = = 1 квадриллион оп. ПЗ./с. Несмотря на кажущуюся простоту и однозначность, flops является достаточно плохой мерой производительности по нескольким причинам. Во-первых, по сути flops измеряет не производительность, а среднее быстродействие для некоторого частотного вектора операций с ПЗ, оставляя «за кадром» взаимодействие процессора с памятью, ввод/вывод и т. д. Во-вторых, само определение flops уже неоднозначно. Под «операцией с плавающей запятой» может скрываться масса разных понятий, связанных с частотами применения конкретных операций с ПЗ, разрядностью операндов. Но, «что выросло, то и выросло» ☺.

ные и систолические[1] вычислительные системы. Именно эти ВС и будут предметом рассмотрения в настоящей главе.

Векторные вычислительные системы

Хотя производительность ВС общего назначения неуклонно возрастает, по-прежнему остаются задачи, требующие существенно большей вычислительной мощности. К таким, прежде всего, следует отнести задачи моделирования реальных процессов и объектов, для которых характерна обработка больших массивов чисел в форме с плавающей запятой. Такие массивы представляются матрицами и векторами, а алгоритмы их обработки описываются в терминах матричных операций. Как известно, основные матричные операции сводятся к однотипным действиям над парами элементов исходных матриц, которые, чаще всего, можно производить параллельно. В универсальных ВС, ориентированных на скалярные операции, обработка матриц выполняется поэлементно и последовательно. При большой размерности массивов последовательная обработка элементов матриц занимает слишком много времени, что и приводит к неэффективности универсальных ВС для рассматриваемого класса задач. Для обработки массивов требуются вычислительные средства, позволяющие с помощью единой команды производить действие сразу над всеми элементами массивов — средства *векторной обработки*.

Понятие вектора и размещение данных в памяти

В средствах векторной обработки под *вектором* понимается одномерный массив данных (обычно в форме с плавающей запятой), размещенных в памяти ВС. Количество элементов массива называется *длиной вектора*. Многомерные массивы считаются наборами одномерных массивов-векторов. Проиллюстрируем сказанное примером. Пусть имеется двумерный массив данных A, представляющий собой матрицу размерности 4×4. Как показано на рис. 13.1, матрицу можно рассматривать как четыре 4-элементных вектора-строки (и размещать в памяти по строкам) или как четыре вектора-столбца (и размещать в памяти по столбцам).

Действия над многомерными массивами учитывают специфику их размещения. Способ размещения многомерного массива влияет на шаг изменения адреса элемента, выбираемого из памяти. Так, если рассмотренная в примере матрица расположена в памяти построчно, адреса соседних элементов строки различаются на единицу, а для элементов столбца шаг равен четырем. При размещении матрицы по столбцам единице будет равен шаг по столбцу, а четырем — шаг по строке. В векторной концепции для обозначения шага, с которым элементы вектора извлекаются из памяти, применяют термин *шаг по индексу* (stride).

[1] Некоторые специалисты в области вычислительной техники причисляют систолические ВС к классу MISD.

Рис. 13.1. Способы размещения в памяти матрицы 4×4

Понятие векторного процессора

Векторный процессор — это процессор, в котором операндами некоторых команд могут выступать массивы данных — векторы. Векторный процессор может быть реализован в двух вариантах. В первом он представляет собой дополнительный блок к универсальной вычислительной машине (системе). Во втором — векторный процессор является основой самостоятельной ВС.

В архитектуре средств векторной обработки используется один из двух подходов — векторно-параллельный или векторно-конвейерный.

В векторно-параллельном процессоре одновременные операции над элементами векторов проводятся с помощью нескольких функциональных блоков с плавающей запятой (ФБ), каждый из которых отвечает за одну пару элементов (рис. 13.2).

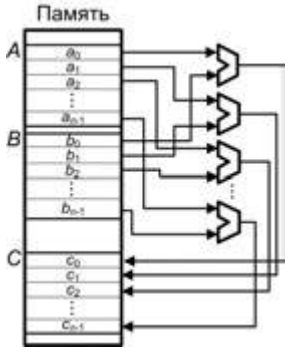


Рис. 13.2. Векторно-параллельная обработка

В векторно-конвейерном варианте (рис. 13.3) обработка элементов векторов производится одним конвейерным ФБ. Операции с числами в форме с ПЗ достаточно

сложны, но поддаются разбиению на отдельные шаги. Каждый этап обработки может быть реализован с помощью отдельной ступени конвейерного ФБ. Очередная пара элементов векторов-операндов подается на вход конвейера как только освобождается его первая ступень.

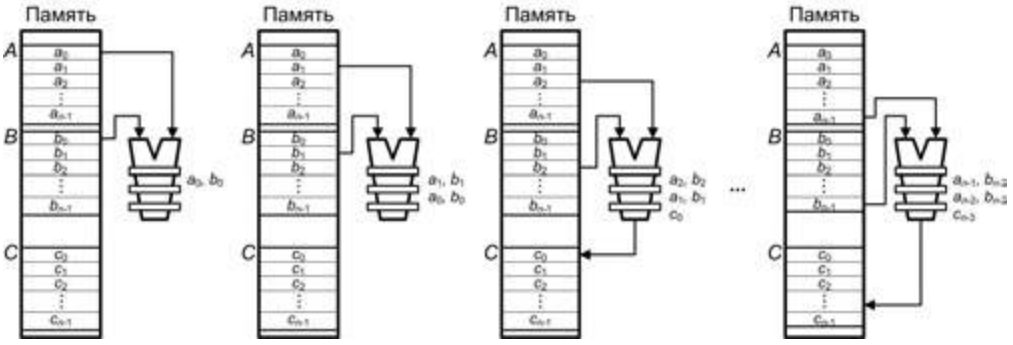


Рис. 13.3. Векторно-конвейерная обработка

Одновременные операции над элементами векторов можно проводить и с помощью нескольких конвейерных ФБ. Такого рода обработка, где совмещены векторно-параллельный и векторно-конвейерный подходы, показана на рис. 13.4.

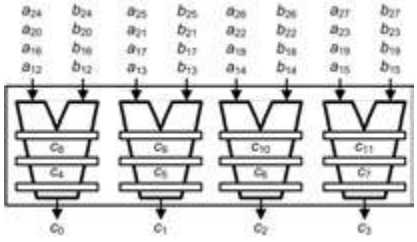


Рис. 13.4. Параллельная обработка векторов несколькими конвейерными ФБ

В настоящее время наиболее распространены векторно-конвейерные ВС с одним или несколькими функциональными блоками, по этой причине в дальнейшем будут обсуждаться векторные ВС только этого типа.

Архитектуры векторной обработки «память-память» и «регистр-регистр»

Принципиальным моментом в архитектуре векторных процессоров является способ доступа к операндам, поскольку векторы-операнды хранятся в памяти ВС и туда же помещается вектор-результат. Для известных векторных ВС можно выделить два варианта архитектуры векторной обработки, известные как «*память-память*» и «*регистр-регистр*».

В векторных процессорах с архитектурой «*память-память*» элементы векторов поочередно извлекаются из памяти и сразу же направляются в функциональный

блок. По мере обработки элементы вектора результата, появляющиеся на выходе ФБ, сразу же заносятся в память.

В архитектуре «*регистр-регистр*» операнды сначала загружаются из памяти в *векторные регистры*. Векторный регистр представляет собой совокупность скалярных регистров, объединенных в очередь типа FIFO, способную хранить 50–100 чисел с плавающей запятой (чаще всего — 64). Операция выполняется над векторами, размещенными в векторных регистрах операндов, а ее результат сначала заносится в векторный регистр результата, а уже из него переписывается в память.

В обеих структурах необходимо обеспечить требуемую последовательность извлечения элементов векторов-операндов из памяти и занесения элементов вектора-результата в память. Эта задача в векторном процессоре реализуется с помощью генератора адресов (рис. 13.5), на выходе которого формируется адрес очередного элемента вектора в памяти. Изначально на вход генератора подается базовый адрес — начальный адрес области памяти, хранящей элементы вектора. Очередной адрес вычисляется путем увеличения предыдущего адреса на величину шага по индексу. Возвращаясь к примеру, приведенному на рис. 13.1, в зависимости от шага (1 или 4) можно считывать из памяти строки или столбцы матрицы.

Об ускорении доступа к векторам, хранящимся в памяти. Ранее отмечалось, что для доступа к структурированным данным в памяти (массивам, векторам), в которых элементы с последовательно возрастающими индексами размещаются в ячейках с последовательно возрастающими адресами, память выгоднее строить как блочную с расслоением. В этом случае адреса ячеек чередуются по циклической схеме (следующий адрес — в следующем банке памяти). Такой прием позволяет почти параллельно читать (записывать) элементы векторов в обеих архитектурах (рис. 13.6).

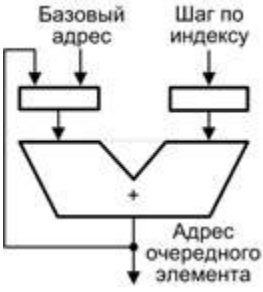


Рис. 13.5. Генератор адресов

Рис. 13.6. Организация памяти векторной системы

Преимущество векторных процессоров «память-память» состоит в возможности обработки длинных векторов, в то время как в процессорах «регистр-регистр» приходится разбивать длинные векторы на сегменты фиксированной длины. К сожалению, за гибкость режима «память-память» приходится расплачиваться относительно большими издержками, известными как *время запуска*, представляющее собой временной интервал между инициализацией команды и моментом, когда первый результат появится на выходе конвейера. Большое время запуска в процессорах «память-память» обусловлено скоростью доступа к памяти, которая намного

меньше скорости доступа к внутреннему регистру. Однако когда конвейер заполнен, результат формируется в каждом цикле. Модель времени работы векторного процессора (ВП) имеет вид:

$$T = s + \alpha \times N,$$

где s — время запуска, α — константа, зависящая от команды (обычно ½, 1 или 2) и N — длина вектора.

Архитектура «память-память» реализована в вычислительных системах Advanced Scientific Computer фирмы Texas Instruments Inc., семействе вычислительных систем фирмы Control Data Corporation, прежде всего Star 100, серии Cyber 200 и ВС типа ETA-10. Все эти вычислительные системы появились в середине 70-х прошлого века после длительного цикла разработки, но к середине 80-х годов от них отказались. Причиной послужило слишком большое время запуска — порядка 100 циклов процессора. Это означает, что операции с короткими векторами выполняются очень неэффективно, и даже при длине векторов в 100 элементов процессор достигал только половины потенциальной производительности.

В вычислительных системах «регистр-регистр» векторы имеют сравнительно небольшую длину (в ВС семейства Cray — 64), но время запуска значительно меньше, чем в случае «память-память». Этот тип векторных систем гораздо более эффективен при обработке коротких векторов, но при операциях над длинными векторами векторные регистры должны загружаться сегментами несколько раз. В настоящее время ВП «регистр-регистр» доминируют на компьютерном рынке. Это вычислительные системы фирмы Cray Research Inc., а также векторные ВС фирм Fujitsu, Hitachi и NEC, например NEC SX-8R (2006). Время цикла в современных ВП составляет порядка 2–2,5 нс.

Структура векторного процессора

Обобщенная структура векторного процессора показана в рамках векторно-конвейерной вычислительной системы, приведенной на рис. 13.7. На схеме показаны основные узлы такого процессора, без детализации связей между ними.

Обработка всех n компонентов векторов-операндов задается одной *векторной командой*. Общепринято, что элементы векторов представляются числами в форме с плавающей запятой (ПЗ). Операционное устройство векторного процессора может быть реализовано в виде единого конвейерного функционального блока, способного выполнять все предусмотренные операции над числами с ПЗ. Более распространена, однако, иная структура, — в ней операционное устройство состоит из отдельных блоков сложения и умножения, а иногда и блока для вычисления обратной величины, когда операция деления $\frac{X}{Y}$ реализуется в виде $X\left(\frac{1}{Y}\right)$. Каждый из таких блоков также конвейеризирован.

Поскольку рассматриваемый векторный процессор относится к типу «регистр-регистр», в нем имеется набор векторных регистров. Обмен информацией между этими регистрами и памятью ВС обеспечивает блок управления чтением/записью векторов, в состав которого входит генератор адресов. Блок загружает в векторные регистры векторы-операнды и записывает в память вектор результата.

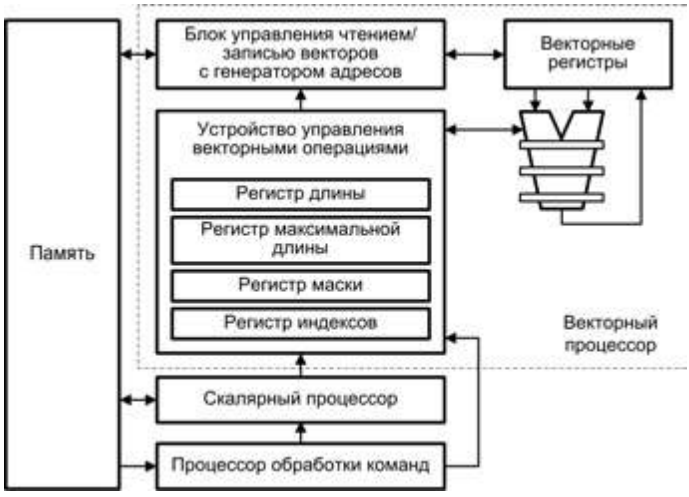


Рис. 13.7. Упрощенная структура векторно-конвейерной вычислительной системы

Функционирование векторного процессора задает устройство управления векторными операциями. Именно оно «отвечает» за выполнение векторных команд. Система команд векторного процессора включает в себя команды:

- загрузки векторного регистра содержимым последовательности ячеек памяти, для которой указан адрес первой ячейки;
- выполнения операций над всеми элементами векторов, находящимися в векторных регистрах;
- сохранения содержимого векторного регистра в последовательности ячеек памяти, указанной адресом первой ячейки.

В состав устройства управления векторными операциями входят несколько регистров специального назначения: регистр длины вектора, регистр максимальной длины, регистр маски и регистр индексов.

Регистр длины вектора определяет, сколько элементов фактически содержит обрабатываемый вектор, то есть сколько индивидуальных операций с элементами нужно сделать.

Регистр максимальной длины вектора фиксирует максимальное число элементов вектора, которое может быть одновременно обработано аппаратурой процессора. Этот регистр используется при разделении очень длинных векторов на сегменты, длина которых соответствует максимальному числу элементов, обрабатываемых аппаратурой за один прием.

Достаточно часто приходится выполнять такие операции, в которых должны участвовать не все элементы векторов. Данный режим организуется с помощью *регистра маски вектора*. В этом регистре каждому элементу вектора соответствует один бит. Установка бита в единицу разрешает запись соответствующего элемента вектора результата в выходной векторный регистр, а сброс в ноль — запрещает. Регистр маски используется также в операциях *уплотнения/развертывания* (compress/

expand). Суть уплотнения состоит в упаковке незамаскированных элементов вектора из векторного регистра-источника и их размещении в регистре-приемнике, при развертывании выполняется обратное действие — распаковка (рис. 13.8).

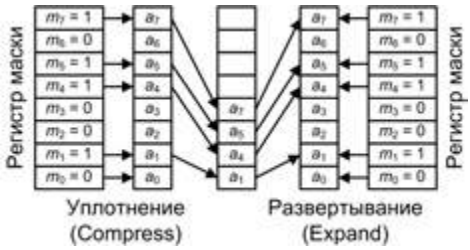


Рис. 13.8. Операции уплотнения/развертывания

Как уже упоминалось, элементы векторов в памяти расположены регулярно, и при выполнении векторных операций достаточно указать значение шага по индексу. Впрочем, возможны исключения: вектор в векторном регистре надо сформировать из «нерегулярных» элементов, элементы вектора результата следует записать в память нерегулярно. Для поддержки подобных исключений в системе команд ВП предусмотрены операции *сбора/рассеяния* (Gather/Scatter). В них используется вектор индексов, сохраняемый в *регистре вектора индексов*. В векторе индексов i-й элемент содержит индекс того элемента исходного массива, который соответствует i-й позиции в векторном регистре. Операции сбора/рассеяния иллюстрирует рис. 13.9. На рисунке показана несколько утрированная ситуация — на самом деле значения индексов имеют регулярный характер, который нарушается, например, при переходе от элементов одного измерения многомерного массива к элементам другого измерения.

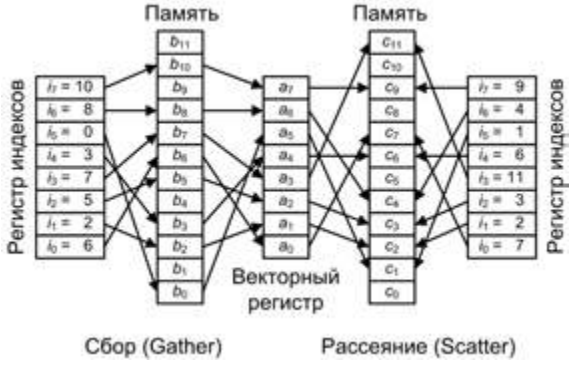


Рис. 13.9. Операции сбора/рассеяния

Структура векторной вычислительной системы

В реальных задачах векторная обработка составляет только часть общей вычислительной нагрузки. Значительный вес имеют и скалярные операции. По этой

причине векторная ВС, помимо векторного процессора, содержит еще и скалярный процессор (рис. 13.7). Как и положено для SIMD-системы, выполняется единая программа, содержащая как скалярные, так и векторные команды. Программа и данные хранятся в памяти ВС. Команды программы последовательно выбираются из памяти процессором обработки команд, который направляет скалярные и векторные команды в скалярный или векторный процессор соответственно.

Ускорение векторных вычислений

Для повышения скорости обработки векторов все функциональные блоки векторных процессоров строятся по конвейерной схеме, причем так, чтобы каждая ступень любого из конвейеров справлялась со своей операцией за один такт (число ступеней в разных функциональных блоках может быть различным). В некоторых векторных ВС, например Cray C90, этот подход несколько усовершенствован — конвейеры во всех функциональных блоках продублированы (рис. 13.10).

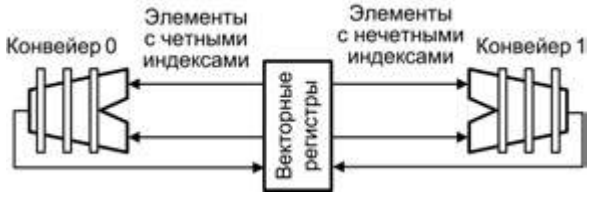


Рис. 13.10. Выполнение векторных операций при двух конвейерах

На конвейер 0 всегда подаются элементы векторов с четными номерами, а на конвейер 1 — с нечетными. В начальный момент на первую ступень конвейера 0 из векторных регистров операндов поступают нулевые элементы векторов. Одновременно первые элементы векторов из этих регистров подаются на первую ступень конвейера 1. На следующем такте на конвейер 0 подаются вторые элементы векторов, а на конвейер 2 — третьи элементы и т. д. Аналогично происходит распределение результатов в выходном векторном регистре. В итоге функциональный блок при максимальной загрузке в каждом такте выдает не один результат, а два.

Интересной особенностью некоторых ВП типа «регистр-регистр», например ВС фирмы Cray Research Inc., является так называемое *зацепление векторов* (vector chaining или vector linking), когда векторный регистр результата одной векторной операции используется в качестве входного регистра для последующей векторной операции. Такая комбинация из последовательности умножения и суммирования характерна для операции свертки и встречается во многих векторных и матричных вычислениях. Сущность зацепления векторов в том, что исполнение векторной команды начинается сразу, как только образуются компоненты участвующих в ней векторных операндов, не дожидаясь завершения вычисления полного вектора операнда и занесения его в соответствующий векторный регистр. Образуются цепочки операций. Логику этого метода ускорения вычислений иллюстрирует рис. 13.11.

Завершая обсуждение векторных ВС, следует отметить, что с середины 90-х годов прошлого века этот вид ВС стал уступать свои позиции другим более техно-

логичным видам систем. Тем не менее одна из последних разработок корпорации NEC (2007 год) — вычислительная система SX-9 — по сути представляет собой векторно-конвейерную ВС. Пиковая производительность системы с 16 ядрами составляет 26,2 TFLOPS (триллионов операций с плавающей запятой в секунду).

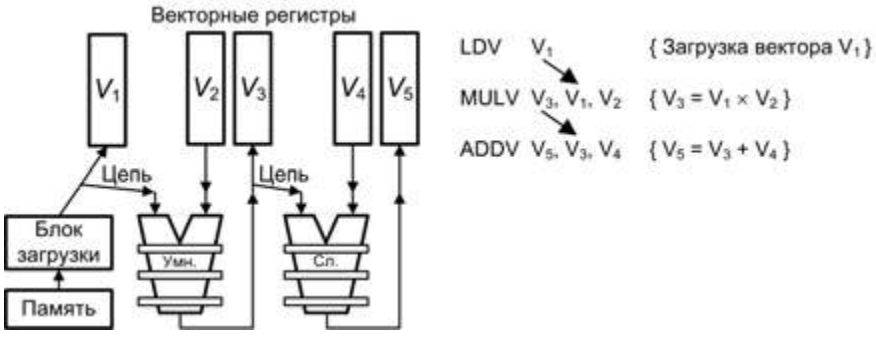


Рис. 13.11. Зацепление векторов

Матричные вычислительные системы

Назначение *матричных вычислительных систем* во многом схоже с назначением векторных ВС — обработка больших массивов данных. В основе матричных систем лежит *матричный процессор* (array processor), состоящий из массива процессорных элементов (ПЭ). Организация систем подобного типа на первый взгляд достаточно проста. Они имеют общее управляющее устройство, генерирующее поток команд, и большое число ПЭ, работающих параллельно и обрабатывающих каждый свой поток данных. Однако на практике, чтобы обеспечить достаточную эффективность системы при решении широкого круга задач, необходимо организовать связи между процессорными элементами так, чтобы наиболее полно загрузить процессоры работой. Именно характер связей между ПЭ и определяет разные свойства системы. Подобная схема применима и для векторных вычислений.

Между матричными и векторными системами есть существенная разница. Матричный процессор интегрирует множество идентичных функциональных блоков (ФБ), логически объединенных в матрицу и работающих в SIMD-стиле. Не столь существенно, как конструктивно реализована матрица процессорных элементов — на едином кристалле или на нескольких. Важен сам принцип — ФБ логически скомпонованы в матрицу и работают синхронно, то есть присутствует только один поток команд для всех. Векторный процессор имеет встроенные команды для обработки векторов данных, что позволяет эффективно загрузить конвейер из функциональных блоков. В свою очередь, векторные процессоры проще использовать, потому что команды для обработки векторов — это более удобная для человека модель программирования, чем SIMD.

Структуру матричной вычислительной системы можно представить в виде, показанном на рис. 13.12 [171].

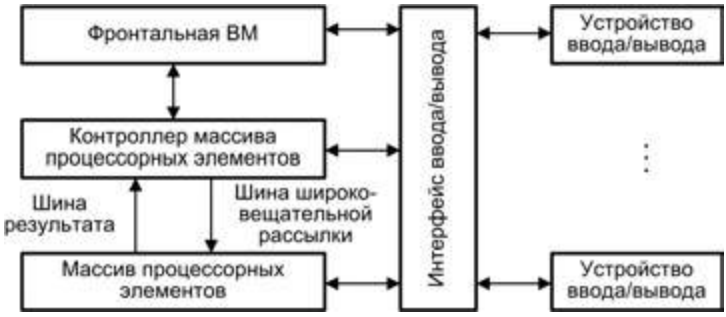


Рис. 13.12. Обобщенная модель матричной SIMD-системы

Собственно параллельную обработку множественных элементов данных обеспечивает *массив процессорных элементов* (МПЭ). Единый поток команд, управляющий обработкой данных в МПЭ, генерируется *контроллером массива процессорных элементов* (КМП). КМП выполняет последовательный программный код, реализует операции условного и безусловного переходов, транслирует в МПЭ команды, данные и сигналы управления. Команды обрабатываются процессорными элементами в режиме жесткой синхронизации. Сигналы управления используются для синхронизации команд и пересылок, а также для управления процессом вычислений, в частности определяют, какие ПЭ массива должны выполнять операцию, а какие — нет. Команды, данные и сигналы управления передаются из КМП в массив процессорных элементов по *шине широковещательной рассылки*. Поскольку выполнение операций условного перехода зависит от результатов вычислений, результаты обработки данных в массиве процессоров транслируются в КМП по *шине результата*.

Для обеспечения пользователя удобным интерфейсом при создании и отладке программ в состав подобных ВС обычно включают *фронтальную ВМ* (front-end computer). В роли такой ВМ выступает универсальная вычислительная машина, на которую дополнительно возлагается задача загрузки программ и данных в КМП. Кроме того, такая загрузка может производиться и напрямую с *устройств ввода/вывода*, например с магнитных дисков. После загрузки КМП приступает к выполнению программы, транслируя в МПЭ по широковещательной шине соответствующие SIMD-команды.

Рассматривая массив ПЭ, следует учитывать, что для хранения множественных наборов данных в нем, помимо множества процессорных элементов, должно присутствовать и множество модулей памяти. Кроме того, в массиве должна быть реализована сеть взаимосвязей, как между ПЭ, так и между процессорными элементами и модулями памяти. Таким образом, под термином *массив процессорных элементов* понимают блок, состоящий из собственно процессорных элементов, модулей памяти и сети соединений.

Дополнительную гибкость при работе с рассматриваемой системой обеспечивает механизм *маскирования*, позволяющий вовлекать в операции лишь определенное подмножество ПЭ массива. Маскирование возможно как на стадии компиляции, так и на этапе выполнения, при этом ПЭ, исключенные путем установки в ноль соответствующих битов маски, во время выполнения команды простаивают.

Фронтальная ВМ

Фронтальная ВМ (ФВМ) соединяет матричную SIMD-систему с внешним миром, используя для этого какой-либо из сетевых интерфейсов, например Ethernet, как это имеет место в системе MasPar MP-1. Фронтальная ВМ работает под управлением операционной системы, чаще всего UNIX-подобной. На ФВМ пользователи подготавливают, компилируют и отлаживают свои программы. Перед выполнением программы сначала загружаются из фронтальной ВМ в контроллер массива процессорных элементов, который исполняет последовательную часть программы и распределяет распараллеленные команды и данные по процессорным элементам массива. В некоторых ВС при создании, компиляции и отладке программ КМП и фронтальная ВМ используются совместно.

На роль ФВМ подходят различные вычислительные машины. Так, в системе CM-2 в этом качестве выступает рабочая станция SUN-4, а в системе MasPar — DECstation 3000.

Контроллер массива процессорных элементов

Контроллер массива процессорных элементов выполняет последовательный программный код, реализует команды ветвления программы, транслирует команды и сигналы управления в процессорные элементы. Рисунок 13.13 иллюстрирует одну из возможных реализаций КМП, в частности принятую в устройстве управления матричной вычислительной системы PASM.

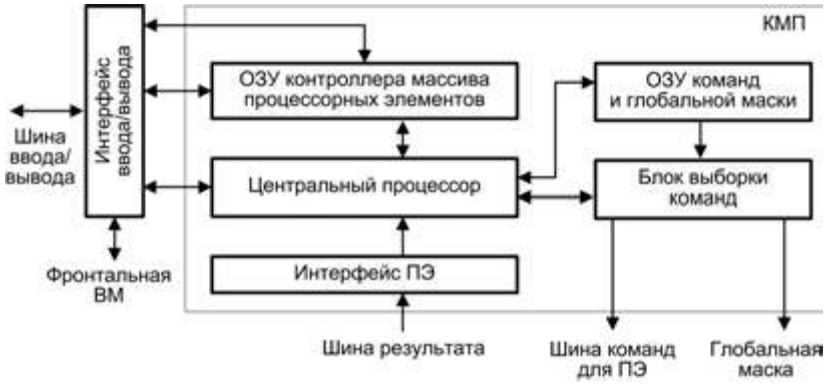


Рис. 13.13. Модель контроллера массива процессорных элементов [164]

При загрузке из ФВМ программа через интерфейс ввода/вывода заносится в оперативное запоминающее устройство контроллера массива процессорных элементов (ОЗУ КМП). Команды для процессорных элементов и глобальная маска, формируемая на этапе компиляции, также через интерфейс ввода/вывода загружаются в ОЗУ команд и глобальной маски (ОЗУ КГМ). Затем контроллер начинает выполнять программу, извлекая либо одну скалярную команду из ОЗУ КМП, либо множественные команды из ОЗУ КГМ. Скалярные команды — команды, осуществляющие операции над хранящимися в КМП скалярными данными, выполняются

центральным процессором (ЦП) контроллера. В свою очередь, команды, оперирующие параллельными переменными, хранящимися в каждом ПЭ, преобразуются в блоке выборки команд в более простые единицы выполнения — *нанокоманды*. Нанокоманды совместно с маской пересылаются через шину команд для ПЭ на исполнение в массив процессорных элементов. Например, команда сложения 32-разрядных слов в КМП системы MPP преобразуется в 32 нанокоманды одноразрядного сложения, которые каждым ПЭ обрабатываются последовательно.

В большинстве алгоритмов дальнейший порядок вычислений зависит от результатов и/или флагов предшествующих операций. Для обеспечения такого режима в матричных системах статусная информация, хранящаяся в процессорных элементах, должна быть собрана в единое слово и передана в КМП для выработки решения о ветвлении программы. Например, в предложении IF ALL (условие A) THEN DO B оператор B будет выполнен, если условие A справедливо во всех ПЭ. Для корректного включения/отключения процессорных элементов КМП должен знать результат проверки условия A во всех ПЭ. Такая информация передается в КМП по однонаправленной шине результата. В системе CM-2 эта шина названа GLOBAL. В системе MPP для той же цели организована структура, называемая *деревом SUM-OR*. Каждый ПЭ помещает содержимое своего одноразрядного регистра признака на входы дерева, которое с помощью операции логического сложения комбинирует эту информацию и формирует слово результата, используемое в КМП для принятия решения.

Массив процессорных элементов

В матричных SIMD-системах распространение получили два основных типа архитектурной организации массива процессорных элементов (рис. 13.14).

В первом варианте, известном как архитектура типа «*процессорный элемент-процессорный элемент*» («ПЭ-ПЭ»), *N* процессорных элементов (ПЭ) связаны между собой коммуникационной сетью (рис. 13.14, *а*). Каждый ПЭ — это процессор с локальной памятью. Процессорные элементы выполняют команды, получаемые из КМП по шине широковещательной рассылки, и обрабатывают данные, как хранящиеся в их локальной памяти так и поступающие из КМП. Обмен данными между процессорными элементами производится по *коммуникационной сети*, в то время как *шина ввода/вывода* служит для обмена информацией между ПЭ и устройствами ввода/вывода. Для трансляции результатов из отдельных ПЭ в контроллер массива процессорных элементов служит *шина результата*. Благодаря использованию локальной памяти аппаратные средства ВС рассматриваемого типа могут быть построены весьма эффективно. Во многих алгоритмах действия по пересылке информации по большей части локальны, то есть происходят между ближайшими соседями. По этой причине архитектура, где каждый ПЭ связан только с соседними, очень популярна. В качестве примеров вычислительных систем с рассматриваемой архитектурой можно упомянуть MasPar MP-1, Connection Machine CM-2, GF11, DAP, MPP, STARAN, PEPE, ILLIAC IV.

Второй вид архитектуры — «*процессор-память*» — показан на рис. 13.14, *б*. В такой конфигурации двунаправленная сеть соединений связывает *N* процессоров с *M* модулями памяти. Процессоры управляются КМП через широковещательную шину.

Обмен данными между процессорами осуществляется как через сеть, так и через модули памяти. Пересылка данных между модулями памяти и устройствами ввода/вывода обеспечивается шиной ввода/вывода. Для передачи данных из конкретного модуля памяти в КМП служит шина результата. Примерами ВС с рассмотренной архитектурой могут служить Burroughs Scientific Processor (BSP), Texas Reconfigurable Array Computer TRAC.

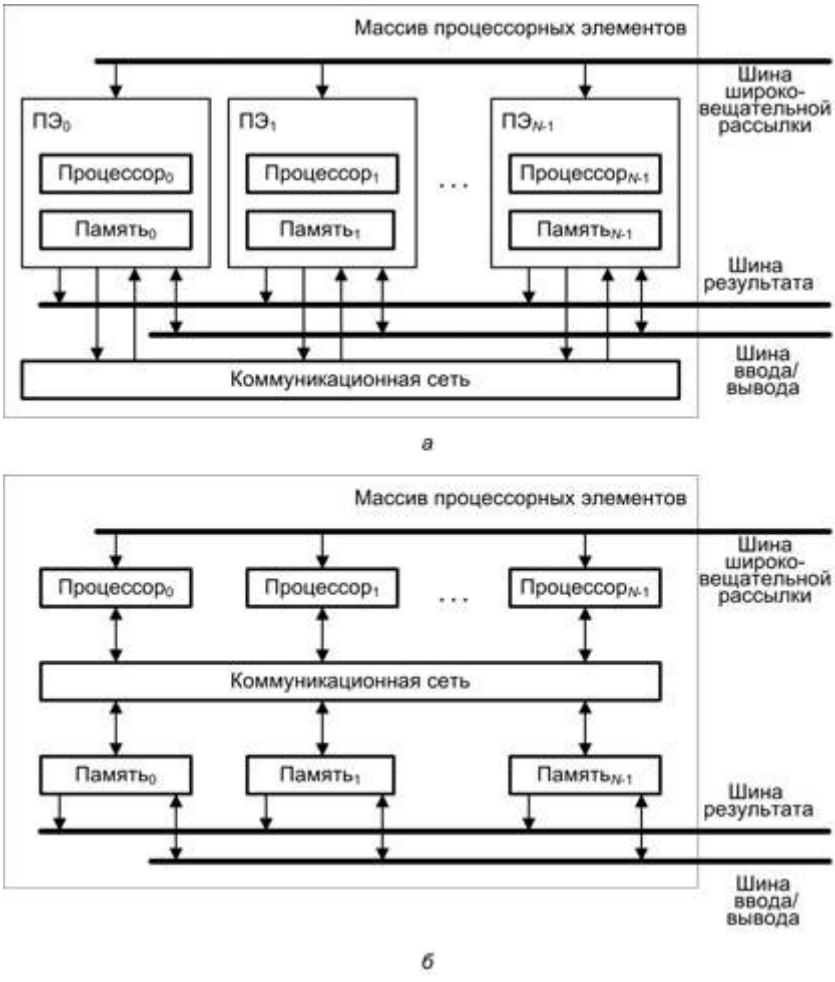


Рис. 13.14. Модели массивов процессоров: *а* — «процессорный элемент-процессорный элемент»; *б* — «процессор-память»

Структура процессорного элемента

В большинстве матричных SIMD-систем в качестве процессорных элементов применяются простые RISC-процессоры с локальной памятью ограниченной емкости. Например, каждый ПЭ системы MasPar MP-1 состоит из четырехразрядного

процессора с памятью емкостью 64 Кбайт. В системе MPP используются одноразрядные процессоры с памятью 1 Кбит каждый, а в СМ-2 процессорный элемент представляет собой одноразрядный процессор с 64 Кбит локальной памяти. Благодаря простоте ПЭ массив может быть реализован в виде одной сверхбольшой интегральной микросхемы (СБИС). Это позволяет сократить число связей между микросхемами и, следовательно, габариты ВС. Так, одна СБИС в системе СМ-2 содержит 16 процессоров (без блоков памяти), а в системе MasPar MP-1 СБИС состоит из 32 процессоров (также без блоков памяти). В системе MP-2 просматривается тенденция к применению более сложных микросхем, в частности 32-разрядных процессоров с 256 Кбайт памяти в каждом.

Неотъемлемыми компонентами ПЭ (рис. 13.15) в большинстве вычислительных систем являются:

- арифметико-логическое устройство (АЛУ);
- регистры данных;
- сетевой интерфейс (СИ), который может включать в свой состав регистры пересылки данных;
- номер процессора;
- регистр флага разрешения маскирования (F);
- локальная память.

Рис. 13.15. Модель процессорного элемента [164]

Процессорные элементы, управляемые командами, поступающими по широковещательной шине из КМП, могут выбирать данные из своей локальной памяти и регистров, обрабатывать их в АЛУ и сохранять результаты в регистрах и локальной памяти. ПЭ могут также обрабатывать те данные, которые поступают по шине

широковещательной рассылки из КМП. Кроме того, каждый процессорный элемент вправе получать данные из других ПЭ и отправлять их в другие ПЭ по коммуникационной сети, используя для этого свой сетевой интерфейс. В некоторых матричных системах, в частности в MasPar MP-1, элемент данных из ПЭ-источника можно передавать в ПЭ-приемник непосредственно, в то время как в других, например в MPP, — данные предварительно должны быть помещены в специальный регистр пересылки данных, входящий в состав сетевого интерфейса. Пересылка данных между ПЭ и устройствами ввода/вывода осуществляется через шину ввода/вывода ВС. В ряде систем (MasPar MP-1) ПЭ подключены к шине ввода/вывода посредством сети соединений и канала ввода/вывода системы. Результаты вычислений любое ПЭ выдает в КМП через шину результата.

Каждому из N ПЭ в массиве процессоров присваивается уникальный номер, называемый также *адресом ПЭ,* который представляет собой целое число от 0 до $N - 1$. Чтобы указать, должен ли данный ПЭ участвовать в общей операции, в процессорном элементе имеется регистр флага разрешения F. Состояние этого регистра может быть изменено сигналами управления из КМП либо по результатам операций в самом ПЭ.

Еще одной существенной характеристикой матричной системы является способ синхронизации работы ПЭ. Так как все ПЭ получают и выполняют команды одновременно, их работа жестко синхронизируется. Это особенно важно в операциях пересылки информации между ПЭ. В системах, где обмен производится с четырьмя соседними ПЭ, передача информации осуществляется в режиме «регистр-регистр».

Подключение и отключение процессорных элементов

В процессе вычислений в ряде операций должны участвовать только определенные ПЭ, в то время как остальные ПЭ остаются бездействующими. Разрешение и запрет работы ПЭ могут исходить от контроллера массива процессорных элементов (*глобальное маскирование*) и реализуются с помощью схем маскирования ПЭ. В этом случае решение о необходимости маскирования принимается на этапе компиляции кода. Решение о маскировании может также приниматься во время выполнения программы (*маскирование, определяемое данными*), при этом опираются на хранящийся в ПЭ флаг разрешения маскирования F.

При маскировании, определяемом данными, каждый ПЭ самостоятельно объявляет свой статус «подключен/не подключен». В составе системы команд имеются наборы маскируемых и немаскируемых команд. Маскируемые команды выполняются в зависимости от состояния флага F, в то время как немаскируемые команды этот флаг просто игнорируют. Процедуру маскирования рассмотрим на примере предложения IF-THEN-ELSE. Пусть x — локальная переменная (хранящаяся в локальной памяти каждого ПЭ). Предположим, что процессорные элементы массива параллельно выполняют ветвление:

```
IF (x > 0) THEN <оператор A> ELSE <оператор B>
```

и каждый ПЭ оценивает условие IF. Те ПЭ, для которых условие $x > 0$ справедливо, установят свой флаг F в единицу, тогда как остальные ПЭ — в ноль. Далее КМП распределяет оператор A по всем ПЭ. Команды, реализующие этот оператор, должны быть маскируемыми. Оператор A будет выполнен только теми ПЭ, где флаг F

установлен в единицу. Далее КМП передает во все ПЭ немаскируемую команду ELSE, которая заставит все ПЭ инвертировать состояние своего флага F. Затем КМП транслирует во все ПЭ оператор B, который также должен состоять из маскируемых команд. Оператор будет выполнен теми ПЭ, где флаг F после инвертирования был установлен в единицу, то есть где результат проверки условия $x > 0$ был отрицательным.

При использовании схемы глобального маскирования контроллер массива процессорных элементов вместе с командами посылает во все ПЭ глобальную маску. Каждый ПЭ декодирует эту маску и выясняет, должен ли он выполнять данную команду или нет.

В зависимости от способа кодирования маски существует несколько различных схем глобального маскирования. В схеме, примененной в вычислительной системе ILLIAC IV с 64 64-разрядными ПЭ, маска представляет собой N-разрядный вектор. Каждый бит вектора отражает состояние одного ПЭ. Если бит содержит единицу, соответствующий ПЭ будет активным, в противном случае — пассивным. Несмотря на свою универсальность, при больших значениях N схема становится неудобной. В варианте маскирования с адресом ПЭ используется $2m$-разрядная маска ($m = \log_2 N$), в которой каждая позиция соответствует одному разряду в двоичном представлении адреса ПЭ. Каждая позиция может содержать 0, 1 или X. Таким образом, маска состоит из $2m$ битов. Если для всех i ($0 \le i < m$) i-я позиция в маске и i-я позиция в адресе ПЭ совпадают или в i-й позиции маски стоит X, ПЭ будет активным. Например, маска 000X1 представляет процессорные элементы с номерами 1 и 3, в то время как маска XXXX0 представляет все ПЭ с четными номерами (все это для массива из 32 ПЭ). Здесь можно активизировать только подмножество из всех возможных комбинаций процессорных элементов массива, что на практике не является ограничением, так как в реальных алгоритмах обычно участвуют не произвольные ПЭ, а лишь расположенные регулярным образом.

Глобальные и локальные схемы маскирования могут комбинироваться. В таком случае активность ПЭ в равной мере определяется как флагом F, так и глобальной маской.

Сети взаимосвязей процессорных элементов

Эффективность сетей взаимосвязей процессорных элементов во многом определяет возможную производительность всей матричной системы. Применение находят самые разнообразные топологии сетей.

Поскольку процессорные элементы в матричных системах функционируют синхронно, обмениваться информацией они также должны по согласованной схеме, причем необходимо обеспечить возможность синхронной передачи от нескольких ПЭ-источников к одному ПЭ-приемнику. Когда для передачи информации в сетевом интерфейсе задействуется только один регистр пересылки данных, это может привести к потере данных, поэтому в ряде ВС для предотвращения подобной ситуации предусмотрены специальные механизмы. Так, в системе СМ-2 используется оборудование, объединяющее сообщения, поступившие к одному ПЭ. Объединение реализуется за счет операций арифметического и логического сложения,

наложения записей, нахождения меньшего и большего из двух значений. В некоторых SIMD-системах, например MP-1, имеется возможность записать одновременно пришедшие сообщения в разные ячейки локальной памяти.

Хотя пересылки данных по сети инициируются только активными ПЭ, пассивные процессорные элементы также вносят вклад в эти операции. Если активный ПЭ инициирует чтение из другого ПЭ, операция выполняется вне зависимости от статуса ПЭ, из которого считывается информация. То же самое происходит и при записи.

Наиболее распространенными топологиями в матричных системах являются решетчатые и гиперкубические. Так, в ILLIAC IV, MPP и CM-2 каждый ПЭ соединен с четырьмя соседними. В MP-1 и MP-2 каждый ПЭ связан с восьмью смежными ПЭ. В ряде систем реализуются многоступенчатые динамические сети соединений (MP-1, MP-2, GF11).

Ввод/вывод

Хотя программа вычислений хранится в памяти фронтальной ВМ или, иногда, в КМП, входные и выходные данные процессорных элементов и КМП могут храниться также на внешних ЗУ. Такие ЗУ могут подключаться к массиву процессорных элементов и/или КМП посредством каналов ввода/вывода или процессоров ввода/вывода.

Ассоциативные вычислительные системы

Ассоциативные вычислительные системы (АВС) — еще один вид вычислительных систем класса SIMD. Как и в матричных системах, здесь также множество процессорных элементов параллельно по одной команде однотипно обрабатывают множественные данные. Однако в отличие от матричных систем обращение к данным производится не по адресам, где хранятся эти данные, а по отличительным признакам, содержащимся в самих данных или в приданных этим данным дополнительных разрядах. Такая возможность обеспечивается ассоциативными процессорами (АП), на базе которых строятся АВС.

Ассоциативные процессоры

Ассоциативным процессором называют специализированный процессор, реализованный на базе ассоциативного запоминающего устройства (АЗУ), где, как известно, доступ к информации осуществляется не по адресу операнда, а по отличительным признакам, содержащимся в самом операнде. От АЗУ традиционного применения ассоциативный процессор (АП) отличают две особенности: наличие средств обработки данных и возможность параллельной записи во все ячейки, для которых было зафиксировано совпадение с ассоциативным признаком. Последнее свойство АП известно как *мультизапись*.

Концепцию ассоциативной обработки поясним на примере упрощенной схемы ассоциативного процессора (рис. 13.16). Запоминающий массив (ЗМ) ассоциативного процессора состоит из n ячеек. Каждое из $m + r$-разрядных слов, хранящихся в ячейках запоминающего массива, состоит из двух частей: m-разрядного слова

данных Д и r-разрядного признака (тега) Т. Тег позволяет отличить данное слово от множества других слов. Так, если в ЗМ хранится вектор А, то теги ячеек, где расположены элементы этого вектора, в какой-то своей части будут одинаковыми, что обозначает принадлежность содержимого ячейки к вектору А. Остальные разряды тега могут содержать, например, индекс элемента вектора. Это позволяет в процессе ассоциативного поиска идентифицировать как все элементы вектора, так и его конкретный элемент. При доступе к ЗМ шаблон для поиска читаемого или записываемого слова (ассоциативный признак) заносится в регистр признака РгП, выходы которого поразрядно связаны со схемами сравнения $СхСр_1$,..., $СхСр_n$. По второму входу каждая из схем сравнения соединена с теговой частью запоминающего массива. При совпадении ассоциативного признака в РгП с теговой частью i-й ячейки вырабатывается сигнал c_i. Сигнал c_i идентифицирует ячейку, для которой имело место совпадение, и разрешает выборку из нее данных, выполнение над ними в процессорном элементе $ПЭ_i$ операции, определенной командой, поступающей из процессора управления, и занесение результата в i-ю ячейку. Процессорные элементы АП позволяют выполнять над данными арифметические и логические операции.

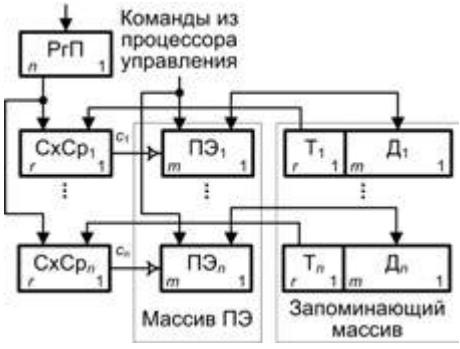


Рис. 13.16. Иллюстрации концепции ассоциативного процессора

Таким образом, идентификация ячеек, отвечающих признаку поиска, обработка содержащихся в них данных и занесение результатов в идентифицированные ячейки осуществляется параллельно по всем словам ЗМ. В упрощенной схеме не показаны связи между ПЭ, позволяющие взаимно обмениваться операндами и результатами.

Способ выполнения операций над словами позволяет определить четыре класса ассоциативных процессоров:

- параллельные;
- поразрядно-последовательные;
- пословно-последовательные;
- блочно-ориентированные.

Два последних класса не слишком перспективны для универсальных вычислений, и ориентированы в основном на задачи информационного поиска. По этой причине основное внимание уделим параллельным и поразрядно-последовательным АП.

Параллельные ассоциативные процессоры

Упрощенная структура параллельного ассоциативного процессора представлена на рис. 13.17 [22].

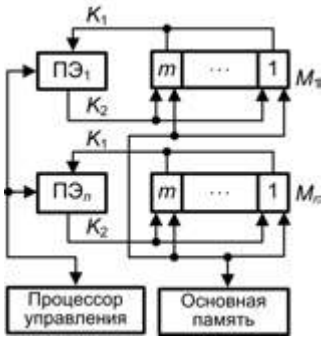


Рис. 13.17. Упрощенная структура параллельного ассоциативного процессора

В качестве элементов обработки используются многоразрядные процессорные элементы. Каждый ПЭ$_i$ работает со своим модулем ассоциативной памяти M_i и осуществляет поиск, а также арифметическую и логическую обработку m-разрядных слов. Пересылку выбранных по содержанию слов между АЗУ и ПЭ обеспечивают коммутирующие цепи K_1 и K_2. Процессорные элементы одновременно выполняют одну и ту же команду, поступающую из процессора управления. Кроме того, предусмотрена возможность обмена данными между модулями ассоциативной памяти и основной памятью, причем обращения по этому каналу производятся как и в обычной памяти — по адресам.

Параллельные АП по сравнению с другими классами ассоциативных процессоров обладают наиболее высоким быстродействием, однако это достигается за счет больших аппаратурных затрат.

Поразрядно-последовательные ассоциативные процессоры

Данный класс АП в настоящее время является наиболее распространенным. Запоминающий массив ассоциативного процессора обычно представляет собой матрицу $n \times n$ 1-разрядных запоминающих элементов (ЗЭ). Считывание и запись информации могут производиться по двум срезам запоминающего массива — либо это все разряды одного слова (рис. 13.18, *а*), либо один и тот же разряд всех слов (рис. 13.18, *б*). Каждый разряд среза в АП снабжен собственным одноразрядным процессорным элементом, что позволяет между считыванием информации и ее записью производить необходимую обработку, то есть параллельно выполнять операции арифметического сложения, поиска, а также эмулировать многие черты матричных ВС, таких, например, как ILLIAC IV. Все ПЭ одновременно выполняют одну и ту же команду, поступающую из процессора управления. АП оперирует m ($m \le n$) разрядами среза, причем значение m устанавливается программистом. При необходимости выделения отдельных разрядов среза лишние позиции допустимо маскировать.

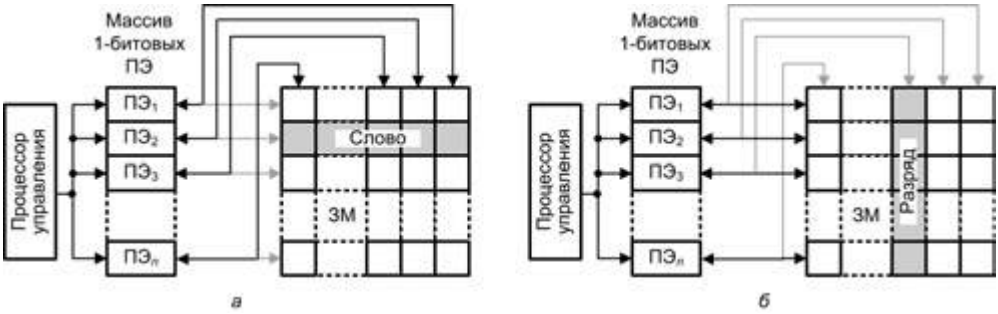


Рис. 13.18. Обработка среза ассоциативной памяти: *а* — все разряды одного слова; *б* — один разряд всех слов

Упрощенная структура процессорного элемента поразрядно-последовательного ассоциативного процессора [22] приведена на рис. 13.19.

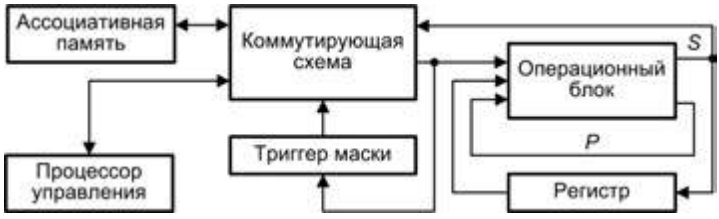


Рис. 13.19. Структура процессорного элемента поразрядно-последовательного ассоциативного процессора

Входящий в ПЭ операционный блок представляет собой одноразрядное арифметико-логическое устройство с цепью переноса P из младшего разряда в старший. Промежуточные результаты S могут быть временно сохранены в регистре. Ассоциативный доступ к разрядам среза, выборку и сохранение данных в ассоциативной памяти, а также связь с процессором управления, откуда поступают команды для ПЭ, обеспечивает коммутирующая схема. Триггер маски используется для блокирования процессорного элемента. При нулевом его состоянии ПЭ не выполняет команду, поступившую одновременно на все процессорные элементы ассоциативного процессора. В целом, ПЭ реализует арифметические, логические, а также поисковые и системные операции.

Конструктивно поразрядно-последовательный ассоциативный процессор исполняется в виде матрицы, содержащей m процессорных элементов и ассоциативную память емкостью $n \times n$ битов.

Ассоциативные многопроцессорные системы

Ассоциативная вычислительная система (АВС) представляет собой многопроцессорную ВС, объединяющую множество ассоциативных процессоров, процессор управления, процессор ввода/вывода и основную память. Упрощенная структура ассоциативной системы показана на рис. 13.20.

Рис. 13.20. Упрощенная структура ассоциативной вычислительной системы

Программа для АВС хранится в основной памяти. В процессе реализации программы процессор управления выбирает из ОП очередную команду программы, декодирует ее и, если эта команда предполагает обработку ассоциативными процессорами, передает ее для параллельного выполнения во все ассоциативные процессоры, готовые к обработке. Кроме того, по ходу обработки процессор управления обеспечивает обмен данными между ассоциативной памятью каждого из АП и основной памятью системы. Связь АВС с внешними устройствами реализуется средствами процессора ввода/вывода. Следует подчеркнуть особую роль, которую в АВС исполняет процессор управления. Помимо уже отмеченных функций, на это устройство возлагаются задачи, которые в матричных ВС обычно выполняет фронтальная ВМ, в частности обеспечение трансляции программ, их редактирование, распараллеливание обработки данных и т. д.

Вычислительные системы с систолической структурой

В фон-неймановских машинах данные, считанные из памяти, однократно обрабатываются в процессорном элементе, после чего снова возвращаются в память (рис. 13.21, *а*). Авторы идеи систолической матрицы Кунг и Лейзерсон предложили организовать вычисления так, чтобы данные на своем пути от считывания из памяти до возвращения обратно пропускались через как можно большее число ПЭ (рис. 13.21, *б*).

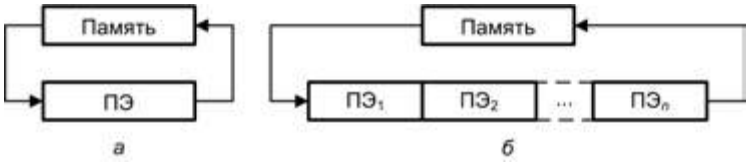


Рис. 13.21. Обработка данных в ВС: *а* — фон-неймановского типа; *б* — систолической структуры

Если проводить аналогию между структурами ВС и живого организма, то памяти можно отвести роль сердца, множеству ПЭ — роль тканей, а поток данных следует рассматривать как циркулирующую кровь. Отсюда и происходит название *систолическая матрица* (систола — сокращение предсердий и желудочков сердца при котором кровь нагнетается в артерии). Систолические структуры эффективны при выполнении матричных вычислений, обработке сигналов, сортировке данных

и т. д. В качестве примера авторами идеи был предложен линейный массив для алгоритма матричного умножения, показанный на рис. 13.22.

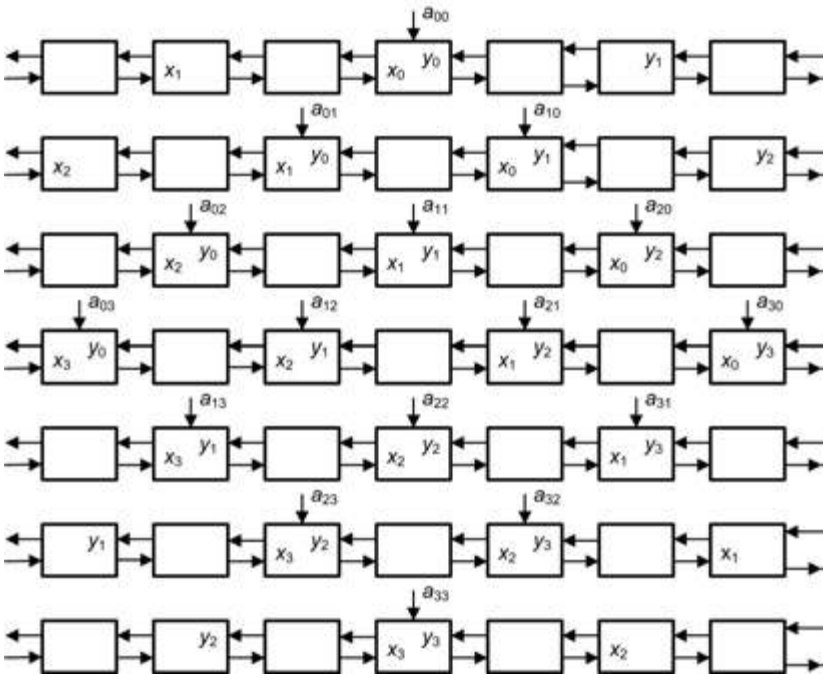


Рис. 13.22. Процесс векторного умножения матриц ($n = 4$)

В основе схемы лежит ритмическое прохождение двух потоков данных x_i и y_i навстречу друг другу. Последовательные элементы каждого потока разделены одним тактовым периодом, чтобы любой из них мог пересечься с любым элементом встречного потока. Если бы они следовали в каждом периоде, то элемент x_i никогда бы не встретился бы с элементами $y_{i+1}, y_{i+3}, \ldots$ Вычисления выполняются параллельно в процессорных элементах, каждый из которых реализует один шаг в операции вычисления скалярного произведения (IPS, Inner Product Step) и носит название *IPS-элемента* (рис. 13.23).

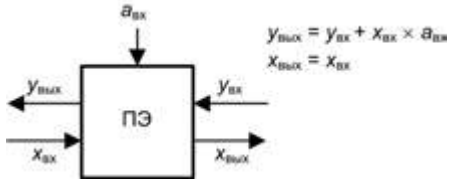


Рис. 13.23. Функциональная схема IPS-элемента

Значение $y_{вх}$, поступающее на вход ПЭ, суммируется с произведением входных значений $x_{вх}$ и $a_{вх}$. Результат выходит из ПЭ как $y_{вых}$. Для возможного последующего

использования остальной частью массива значение $x_{вх}$ транслируется через ПЭ без изменений и покидает его в виде $x_{вых}$.

Таким образом, *систолическая структура* — это однородная вычислительная среда из процессорных элементов, совмещающая в себе свойства конвейерной и матричной обработки и обладающая следующими особенностями:

- вычислительный процесс в систолических структурах представляет собой непрерывную и регулярную передачу данных от одного ПЭ к другому без запоминания промежуточных результатов вычисления;
- каждый элемент входных данных выбирается из памяти однократно и используется столько раз, сколько необходимо по алгоритму, ввод данных осуществляется в крайние ПЭ матрицы;
- образующие систолическую структуру ПЭ однотипны и каждый из них может быть менее универсальным, чем процессоры обычных многопроцессорных систем;
- потоки данных и управляющих сигналов обладают регулярностью, что позволяет объединять ПЭ локальными связями минимальной длины;
- алгоритмы функционирования позволяют совместить параллелизм с конвейерной обработкой данных;
- производительность матрицы можно улучшить за счет добавления в нее определенного числа ПЭ, причем коэффициент повышения производительности при этом линеен.

В настоящее время достигнута производительность систолических процессоров порядка 1000 млрд операций/с.

Классификация систолических структур

Анализ различных типов систолических структур и тенденций их развития позволяет классифицировать эти структуры по нескольким признакам.

По *степени гибкости* среди систолических структур выделяют:

- специализированные;
- алгоритмически ориентированные;
- программируемые.

Специализированные структуры ориентированы на выполнение определенного алгоритма. Эта ориентация отражается не только в конкретной геометрии систолической структуры, статичности связей между ПЭ и числе ПЭ, но и в выборе типа операции, выполняемой всеми ПЭ. Примерами являются структуры, ориентированные на рекурсивную фильтрацию, быстрое преобразование Фурье для заданного количества точек, конкретные матричные преобразования.

Алгоритмически ориентированные структуры, как явствует из их названия, ориентированы не на один конкретный алгоритм, а на определенный класс алгоритмов. Речь идет, главным образом, об алгоритмах, предполагающих выполнение однотипных операций над векторами, матрицами и другими числовыми множествами. В одних структурах настройка на нужный алгоритм производится путем

частичного перепрограммирования процессорных элементов, в других — путем изменения конфигурации связей в систолической матрице, осуществляемого программными средствами.

В программируемых систолических структурах имеется возможность программирования как процессорных элементов, так и конфигурации связей между ними. При этом ПЭ могут обладать локальной памятью программ, и хотя все они имеют одну и ту же организацию, в один и тот же момент времени допускается выполнение различных операций из некоторого набора. Команды или управляющие слова, хранящиеся в памяти программ таких ПЭ, могут изменять и направление передачи операндов.

По *разрядности процессорных элементов* различают систолические структуры:

- одноразрядные;
- многоразрядные.

В одноразрядных матрицах ПЭ в каждый момент времени выполняет операцию над одним двоичным разрядом; а в многоразрядных — над словами фиксированной длины.

По *характеру локально-пространственных связей* систолические структуры бывают:

- одномерные;
- двухмерные;
- трехмерные.

Выбор структуры зависит от вида обрабатываемой информации. Одномерные схемы применяются при обработке векторов, двухмерные — матриц, трехмерные — коллекций иного типа.

Топология систолических структур

В настоящее время разработаны систолические матрицы с различной геометрией связей: линейные, квадратные, гексагональные, трехмерные и др. Перечисленные конфигурации систолических матриц приведены на рис. 13.24.

Каждая конфигурация матрицы наиболее приспособлена для выполнения определенных функций, например линейная матрица оптимальна для реализации фильтров в реальном масштабе времени; гексагональная — для выполнения операций обращения матриц, трехмерная — для нахождения значений нелинейных дифференциальных уравнений в частных производных или для обработки сигналов антенной решетки. Наиболее универсальными и наиболее распространенными, тем не менее, можно считать матрицы с линейной структурой.

Для решения сложных задач систолическая структура может представлять собой набор отдельных матриц, сложную сеть взаимосвязанных матриц, либо обрабатывающую поверхность. Под *обрабатывающей поверхностью* понимается бесконечная прямоугольная сетка ПЭ, где каждый ПЭ соединяется со своими четырьмя соседями (или большим числом ПЭ). Одним из наиболее подходящих элементов для реализации обрабатывающей поверхности является матрица простых ПЭ или транспьютеров.

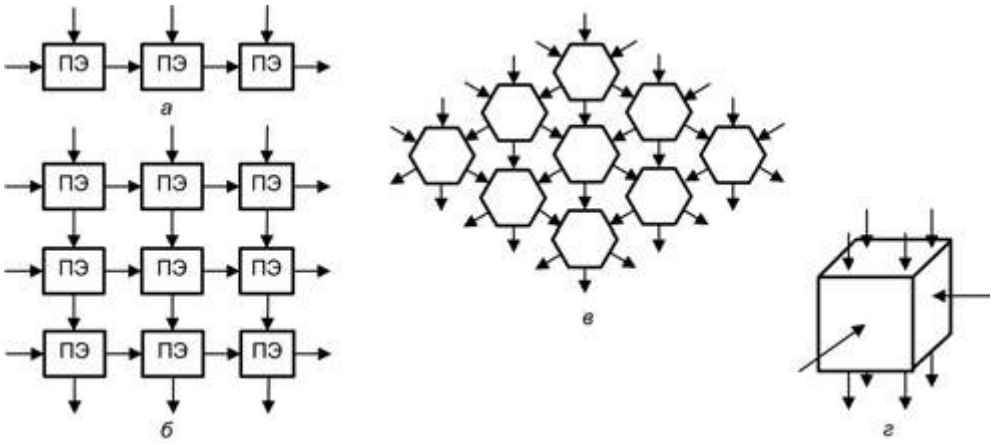


Рис. 13.24. Конфигурация систолических матриц: *а* — линейная; *б* — прямоугольная; *в* — гексагональная; *г* — трехмерная

Учитывая то, что матрицы ПЭ обычно реализуются на основе сверхбольших интегральных схем, возникающие при этом ограничения привели к тому, что наиболее распространены матрицы с одним, двумя и тремя трактами данных и с одинаковым либо противоположным направлением передачи, обозначаемые как ULA, BLA и TLA соответственно.

ULA (Unidirectional Linear Array) — это однонаправленный линейный процессорный массив, где потоки данных перемещаются в одном направлении. ПЭ в массиве могут быть связаны одним, двумя или тремя трактами.

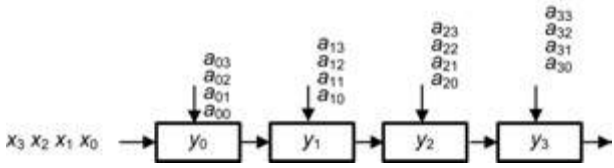


Рис. 13.25. Поток данных при векторном перемножении матриц в ULA ($n = 4$)

При реализации алгоритма векторного произведения матриц один из потоков данных перемещается вправо, в то время как второй резидентно расположен в массиве (рис. 13.25). Используемый ПЭ представляет собой модифицированный IPS-элемент, поскольку имеется только один тракт данных, а элементы второго потока хранятся в ПЭ массива.

BLA (Bidirectional Linear Array) — это двунаправленный линейный процессорный массив, в котором два потока данных движутся навстречу друг другу. Массив типа BLA, где один из потоков является выходным, называется *регулярным*.

Реализация рассмотренной ранее операции с применением BLA показана на рис. 13.26. В версии ULA процессоры используются более эффективно, поскольку в них элементы потока следуют в каждом такте, а не через такт, как в BLA.

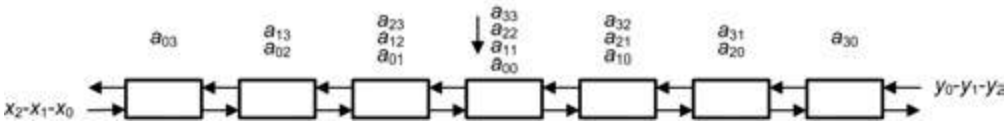


Рис. 13.26. Поток данных при векторном перемножении матриц в BLA (n = 4)

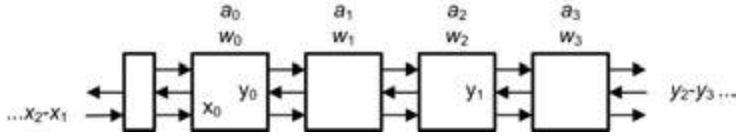


Рис. 13.27. Поток данных в TLA фильтра ARMA (n = 3)

TLA (Three-path communication Linear Array) — линейный процессорный массив с тремя коммуникационными трактами, в котором по разным направлениям перемещаются три потока данных. На рис. 13.27 показан пример фильтра ARMA, предложенного Кунгом и построенного по схеме TLA. Возможны несколько вариантов такого фильтра, в зависимости от числа выходных потоков данных и от значений, хранящихся в памяти (в примере фигурирует один выходной поток). Процессорные элементы выполняют две операции IPS и обычно называются *сдвоенными IPS-элементами*. Две версии таких ПЭ представлены на рис. 13.28. ПЭ могут использовать как хранимые в памяти значения (рис. 13.28, *а*, *б*), так и внешние данные (рис. 13.28, *в*, *г*).

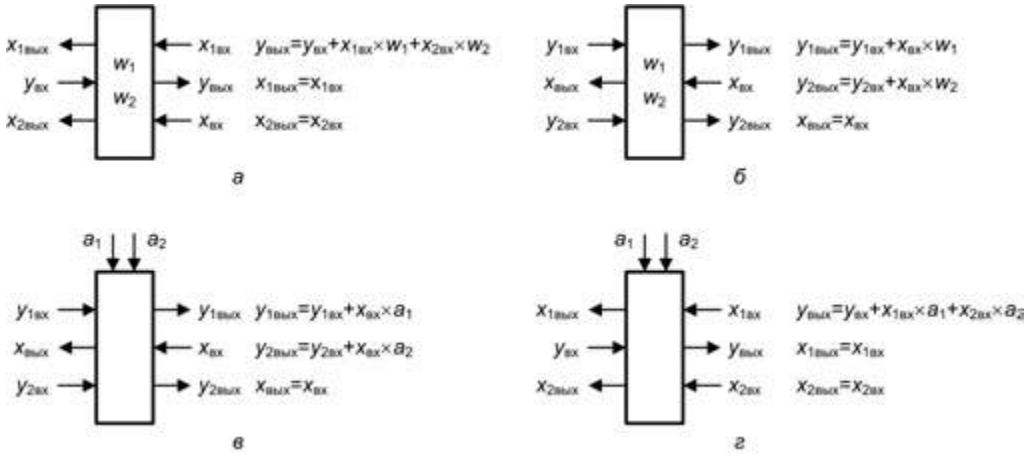


Рис. 13.28. Сдвоенные IPS-элементы: *а–б* — с хранимыми в памяти двумя значениями; *в–г* — с внешними данными

TLA часто называют сдвоенным конвейером, поскольку он может быть разделен на два линейных конвейера типа BLA. Соответственно, TLA можно получить объединением двух BLA с одним общим потоком данных.

Представленные реализации алгоритма векторного произведения матриц выполняют эту операцию за одно и то же время, но в случае ULA в вычислениях участвуют вдвое меньше процессорных элементов. С другой стороны, ULA использует хранимые в памяти данные, на чтение и запись которых нужно какое-то время. В свою очередь, в схеме BLA требуется дополнительное время на операции ввода/вывода.

Структура процессорных элементов

Тип ПЭ выбирается в соответствии с назначением систолической матрицы и структурой пространственных связей. Наиболее распространены процессорные элементы, ориентированные на умножение с накоплением.

На рис 13.29 показаны ПЭ для двух типов матриц: прямоугольной (рис. 13.29, *а*) и гексагональной (рис. 13.29, *б*).

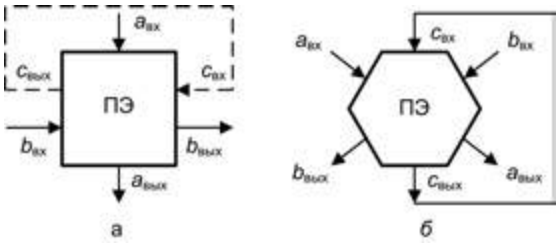


Рис. 13.29. Структура ПЭ: *а* — для прямоугольной систолической матрицы; *б* — для гексагональной систолической матрицы

В обоих случаях на вход ПЭ подаются два операнда $a_{вх}$, $b_{вх}$, а выходят операнды $a_{вых}$, $b_{вых}$ и частичная сумма $c_{вых}$. На n-м шаге работы систолической системы ПЭ выполняет операцию

$$c^{(n)}{}_{вых} = c^{(n-1)}{}_{вх} + a^{(n-1)}{}_{вх} \times b^{(n-1)}{}_{вх}$$

на основе операндов, полученных на $(n - 1)$-м шаге, при этом операнды на входе и выходе ПЭ одинаковы:

$$a^{(n)}{}_{вых} = a^{(n-1)}{}_{вх},\ b^{(n)}{}_{вых} = b^{(n-1)}{}_{вх}.$$

Частичная сумма поступает на вход ПЭ либо с данного процессорного элемента (штриховая линия), либо с соседнего ПЭ матрицы.

Пример вычислений с помощью систолического процессора

Организацию вычислительного процесса в систолических массивах различной конфигурации с использованием ПЭ, функциональная схема которого показана на рис. 13.30, удобнее всего пояснить на примере умножения матрицы $\mathbf{A} = \{a_{ij}\}$ на вектор $\mathbf{X} = \{x_1, x_2, ..., x_n\}$.

Элементы вектора произведения $\mathbf{Y} = \{y_1, y_2, ..., y_n\}$ могут быть получены периодически повторяющимися операциями

$$y_i(1) = 0;\ y_i(k+1) = y_i(k) + a_i(k) \times x_i(k);\ y_i = y_i(n+1),$$

где k — номер шага вычислений.

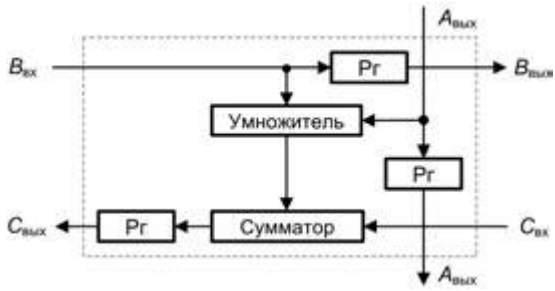


Рис. 13.30. Функциональная схема процессорного элемента систолической матрицы

Пусть имеется матрица **A** размером $n \times n$ с шириной полосы ненулевых элементов $p + q - 1 = 4$. Схема умножения вектора на матрицу в этом случае представлена на рис. 13. 31.

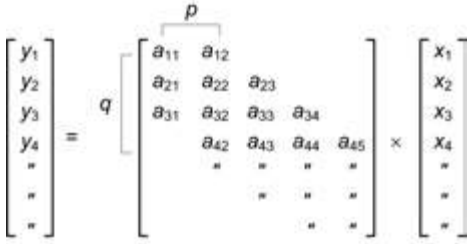


Рис. 13.31. Схема умножения вектора на матрицу

Определенная выше последовательность операций для вычисления компонентов вектора **Y** может быть получена за счет конвейерного прохождения x_i и y_i через $p + q - 1$ последовательно соединенных ПЭ (рис. 13.32)

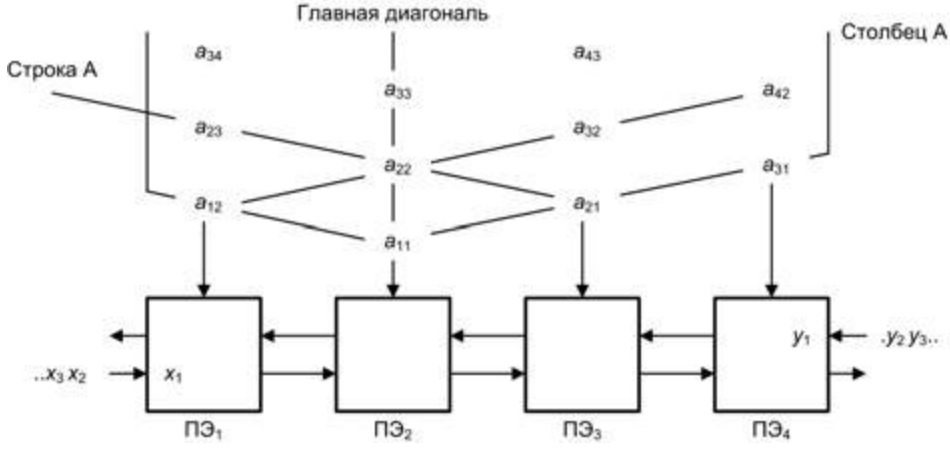


Рис. 13.32. Организация вычислений в линейной систолической структуре

Компоненты y_i ($i = 1, ..., n$) вектора **Y**, имеющие в начальный момент нулевое значение, поступают на вход массива и продвигаются через ПЭ справа налево, в то время как компоненты вектора **X** движутся слева направо. Элементы матрицы $\mathbf{A} = \{a_{ij}\}$ в порядке, указанном на рисунке, вводятся в ПЭ сверху вниз. Промежуточные результаты $y_i(k)$ накапливаются по мере продвижения от одного ПЭ к другому.

В табл. 13.1 показаны первые 6 шагов алгоритма умножения для рассматриваемой структуры.

Таблица 13.1. Последовательность умножения матрицы на вектор в систолической ВС

Шаг	Состояние				Комментарий
	$ПЭ_1$	$ПЭ_2$	$ПЭ_3$	$ПЭ_4$	
0				y_1	Элемент y_1 поступил в $ПЭ_4$
1	x_1		y_1		Элемент x_1 поступил в $ПЭ_1$, элемент y_1 движется влево
2		y_1 a_{11} x_1		y_2	Элемент a_{11} поступил в $ПЭ_2$, $y_1 = y_1 + a_{11} \times x_1$, то есть $y_1 = a_{11} + x_1$
3	y_1 a_{12} x_2		y_2 a_{21} x_1		Элемент a_{12} поступил в $ПЭ_1$, a_{21} — в $ПЭ_3$, $y_1 = a_{11} \times x_1 + a_{12} \times x_2$, $y_2 = a_{21} \times x_1$
4		y_2 a_{22} x_2		y_3 a_{31} x_1	Элемент y_1 вышел из $ПЭ_1$, $y_2 = a_{21} \times x_1 + a_{22} \times x_2$, $y_3 = a_{31} \times x_1$
5	y_2 a_{23} x_3		y_3 a_{32} x_2		$y_2 = a_{21} \times x_1 + a_{22} \times x_2 + a_{23} \times x_3$, $y_3 = a_{31} \times x_1 + a_{32} \times x_2$
6		y_3 a_{33} x_3		y_4 a_{42} x_2	Элемент y_2 вышел из $ПЭ_1$, $y_4 = a_{42} \times x_2$, $y_3 = a_{31} \times x_1 + a_{32} \times$ $\times x_2 + a_{33} \times x_3$

Заметим, что при такой организации вычислительного процесса для каждого ПЭ такты выполнения операции чередуются с тактами простоя. Таким образом, в каждый момент времени активны только $\frac{p+q-1}{2}$ процессорных элементов, следовательно, каждый выходной результат формируется за два такта. Для вычисления всех n элементов выходного вектора **Y** необходимо $2n + p + q - 1$ тактов.

Контрольные вопросы

1. Какой уровень параллелизма в обработке информации обеспечивают вычислительные системы класса SIMD?
2. На какие структуры данных ориентированы средства векторной обработки?
3. Благодаря чему многомерные массивы при обработке можно рассматривать в качестве одномерных векторов?

4. Чем схожи и в чем различие архитектур векторной обработки «память-память» и «регистр-регистр»?
5. Поясните различие между векторно-параллельными и векторно-конвейерными вычислительными системами.
6. Поясните назначение регистров векторного процессора: регистра длины вектора, регистра максимальной длины вектора, регистра вектора индексов и регистра маски.
7. Для чего используются операции уплотнения/развертывания вектора?
8. Оцените выигрыш в быстродействии векторного процессора за счет зацепления векторов.
9. В чем заключается принципиальное различие между векторными и матричными вычислительными системами?
10. Какими средствами обеспечивается подготовка программ для матричных вычислительных систем и их загрузка?
11. По какому принципу в матричной ВС команды программы распределяются между центральным процессором и массивом процессоров?
12. Каким образом в матричной ВС реализуются предложения типа IF-THEN-ELSE?
13. Как идентифицируются отдельные процессорные элементы в массиве процессоров матричной ВС?
14. Какие схемы глобального маскирования применяются в матричных ВС и в каких случаях каждая из них является предпочтительной?
15. Могут ли участвовать в вычислениях замаскированные (пассивные) процессорные элементы матричной ВС и в каким виде это участие проявляется?
16. Поясните различие между ассоциативной памятью и ассоциативным процессором.
17. В чем выражается аналогия между матричными и ассоциативными ВС?
18. Какую особенность систолической ВС отражает ее название?
19. Объясните достоинства и недостатки систолических массивов типа ULA, BLA, TLA.

Глава 14

Вычислительные системы класса MIMD

Технология SIMD исторически стала осваиваться раньше, чем MIMD, что и предопределило ее широкое распространение. В настоящее время наметился устойчивый интерес к MIMD-системам, которые обладают большей гибкостью, в частности могут работать и как высокопроизводительные однопользовательские системы, и как многопрограммные ВС, выполняющие множество задач параллельно. Кроме того, архитектура MIMD позволяет наиболее эффективно распорядиться всеми преимуществами современной микропроцессорной технологии и добиться наивысших показателей производительности.

В MIMD-системе каждый процессорный элемент (ПЭ) выполняет свою программу достаточно независимо от других ПЭ. В то же время ПЭ должны как-то взаимодействовать друг с другом. Различие в способе такого взаимодействия определяет условное деление MIMD-систем на ВС с разделяемой памятью и системы с распределенной памятью.

В *системах с разделяемой памятью*, которые характеризуют как *сильно связанные* (tightly coupled), реализовано единое адресное пространство, доступное всем процессорным элементам посредством общей шины или сети соединений. Такие ВС часто называют мультипроцессорами.

В *системах с распределенной памятью* или *слабо связанных* (loosely coupled) вычислительных системах вся память распределена между процессорными элементами, и каждый блок памяти доступен только «своему» процессору. Фактически, в качестве элементов ВС выступают не процессоры, а вычислительные машины, связанные между собой сетью соединений. Иногда для обозначения подобных ВС используют термин «мультикомпьютеры».

Базовой моделью вычислений на MIMD-системе является совокупность независимых процессов, эпизодически обращающихся к совместно используемым данным. Существует множество вариантов этой модели. На одном конце спектра — распределенные вычисления, в рамках которых программа делится на довольно большое

число параллельных фрагментов, состоящих из множества подпрограмм. На другом конце — модель потоковых вычислений, где каждая операция в программе может рассматриваться как отдельный процесс. Такая операция ожидает поступления входных данных (операндов), которые должны быть переданы ей другими процессами. При их получении операция выполняется, и результирующее значение передается тем процессам, которые в нем нуждаются.

MIMD-системы с разделяемой памятью

К этому виду MIMD-систем причисляют ВС, в которых каждый из множества процессорных элементов имеет доступ к единому адресному пространству, благодаря чему обеспечивается взаимодействие ПЭ при совместном решении общих задач. Конечно, наряду с разделяемой памятью каждый ПЭ может обладать и локальной памятью. Тем не менее взаимодействие процессорных элементов обеспечивается через разделяемую часть памяти ВС.

Принято выделять три основных группы MIMD-систем с разделяемой памятью: симметричные мультипроцессорные системы (SMP), параллельные векторные системы (PVP) и системы с неоднородным доступом к памяти (NUMA).

Симметричные мультипроцессорные системы

До недавнего времени практически все однопользовательские персональные ВМ и рабочие станции содержали по одному микропроцессору общего назначения. По мере возрастания требований к производительности и снижения стоимости микропроцессоров поставщики вычислительных средств как альтернативу однопроцессорным ВМ стали предлагать *симметричные мультипроцессорные вычислительные системы*, так называемые *SMP-системы* (SMP, Symmetric Multiprocessor). Это понятие относится как к архитектуре ВС, так и к операционной системе, обслуживающей данную архитектуру.

SMP-система состоит из множества процессорных элементов, каждый из которых имеет равноправный доступ к логически единой памяти (физически память обычно строится по блочному принципу). Таким образом, в системе реализована концепция однородного доступа к памяти (UMA), рассмотренная в главе 11. Обычно все ПЭ идентичны, однако допускается использование и различных ПЭ. В последнем случае главное условие — сопоставимая производительность, с тем чтобы при распределении вычислительной нагрузки ее можно было без проблем возложить на любой ПЭ системы. Одинаковые возможности ПЭ и их равноправие при доступе к памяти обусловили термин «симметричная» в названии данного вида ВС.

Все процессорные элементы управляются единственным экземпляром операционной системы (ОС), загружаемой в совместно используемую память. ОС планирует распределение выполняемых заданий между процессорными элементами. Для этого ОС выделяет в процессах фрагменты (нити) и образует из этих фрагментов единую очередь. При освобождении какого-либо ПЭ (неважно, какого) ему сразу же передается очередной фрагмент из очереди. Современные ОС обычно поддерживают работу 16 или 32 ПЭ, хотя в некоторых UNIX-подобных операционных системах заложена поддержка 64 процессорных элементов.

Хотя технически SMP-системы симметричны, в их работе присутствует небольшой фактор перекоса, который вносит программное обеспечение. На время загрузки системы один из процессоров получает статус ведущего (master). Это не означает, что позже, во время работы какие-то процессоры будут ведомыми — все они в SMP-системе равноправны. Термин «ведущий» лишь указывает, какой из процессоров будет руководить первоначальной загрузкой ВС. В некоторых SMP-системах ведущим назначается ПЭ с наибольшим номером.

На рис. 14.1, *а* в самом общем виде показана архитектура симметричной мультипроцессорной ВС.

Рис. 14.1. Организация симметричной мультипроцессорной системы: *а* — с локальной кэш-памятью; *б* — с разделяемой кэш-памятью

Типовая SMP-система содержит от двух до 32 идентичных процессоров, в качестве которых обычно выступают недорогие RISC-процессоры. В последнее время наметилась тенденция оснащения SMP-систем также и CISC-процессорами.

Каждый процессор снабжен локальной кэш-памятью. Согласованность содержимого кэш-памяти всех процессоров обеспечивается аппаратными средствами. В некоторых SMP-системах проблема когерентности снимается за счет разделяемой кэш-памяти (рис. 14.1, *б*). К сожалению, этот прием технически и экономически оправдан лишь при числе процессоров не большем четырех. Применение разделяемой кэш-памяти сопровождается повышением стоимости и снижением быстродействия.

Все процессоры ВС имеют равноправный доступ к разделяемой основной памяти и устройствам ввода/вывода. Такая возможность обеспечивается коммуникационной системой. Обычно процессоры взаимодействуют между собой через основную память (сообщения и информация о состоянии оставляются в области общих данных). В некоторых SMP-системах предусматривается также прямой обмен сигналами между процессорами.

Память системы обычно строится по блочному принципу и организована так, что допускается одновременное обращение к разным ее банкам. В некоторых конфигурациях (в дополнение к совместно используемым ресурсам) каждый процессор обладает также собственными дополнительными средствами (локальной основной памятью и каналами ввода/вывода).

Важным аспектом архитектуры симметричных мультипроцессоров является способ взаимодействия процессоров с общими ресурсами (памятью и системой ввода/вывода). С этих позиций можно выделить следующие виды архитектуры SMP-систем:

- с общей шиной;
- с коммутатором типа «кроссбар»;
- с многопортовой памятью;
- с централизованным устройством управления.

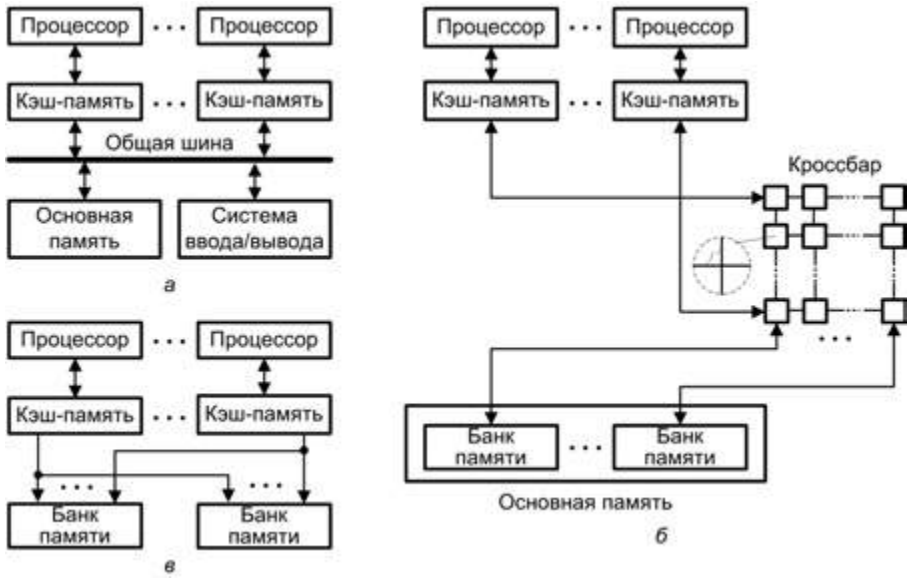


Рис. 14.2. Архитектура SMP-систем: *а* — с общей шиной; *б* — с коммутатором типа «кроссбар»

Пример *архитектуры с общей шиной* приведен на рис. 14.2, *а*. Физический интерфейс, логика адресации, арбитража и разделения времени общей шины остаются теми же, что и в однопроцессорных системах. Общая шина позволяет легко расширять систему путем подключения большего числа процессоров. Кроме того, поскольку шина является пассивной средой, отказ подключенного устройства не распространяется на другие устройства системы.

Основным недостатком SMP-систем на базе общей шины является невысокая производительность (скорость системы ограничена временем цикла шины). Для компенсации недостатка каждому процессору придается собственная кэш-память (в итоге уменьшается число обращений к шине). Увы, появляется новая проблема: когерентность кэш-модулей (обычное решение — применение протокола наблюдения). Все эти факторы существенно ограничивают число процессоров в ВС. Так, в системах Compaq AlphaServer GS140 и 8400 используется не более 14 процессоров Alpha 21264. SMP-система HP N9000 в максимальном варианте состоит из

8 процессоров PA-8500, а система SMP Thin Nodes для RS/6000 фирмы IBM может включать в себя от двух до четырех процессоров PowerPC 604.

Архитектура с общей шиной широко распространена в SMP-системах, построенных на микропроцессорах x86.

Архитектура с коммутатором типа «кроссбар» (рис. 14.2, *б*) ориентирована на блочное построение общей памяти и призвана разрешить проблему ограниченной пропускной способности систем с общей шиной.

Коммутатор обеспечивает множественность путей между процессорами и банками памяти, причем топология связей может быть как двухмерной, так и трехмерной. Результатом становится более высокая полоса пропускания, что позволяет строить SMP-системы, содержащие больше процессоров, чем в случае общей шины. Типичное число процессоров в SMP-системах на базе матричного коммутатора составляет 32 или 64. Отметим, что выигрыш в производительности достигается лишь когда разные процессоры обращаются к разным банкам памяти.

По логике кроссбара строится и взаимодействие процессоров с устройствами ввода/вывода (на схеме не показано).

Приведем ряд примеров. Система Enterprise 10000 состоит из 64 процессоров, связанных с памятью посредством матричного коммутатора Gigaplane-XB фирмы Sun Microsystems (кроссбар 16×16). В IBM RS/6000 Enterprise Server Model S70 коммутатор типа «кроссбар» обеспечивает работу 12 процессоров RS64. В SMP-системах ProLiant 8000 и 8500 фирмы Compaq для объединения памяти и восьми процессоров Pentium III Xeon применена комбинация нескольких шин и кроссбара.

В основе *архитектуры с многопортовой памятью* лежит использование многопортовых запоминающих устройств. Многопортовая организация ЗУ обеспечивает любому процессору (модулю ввода/вывода) непосредственный доступ к банкам основной памяти (рис. 14.2, *в*). Такой подход требует существенного усложнения логики управления ЗУ, хотя и обеспечивает подъем производительности. Другое преимущество многопортовой организации — возможность назначать отдельные банки памяти в качестве локальной памяти конкретного процессора. В результате улучшается защита данных от несанкционированного доступа со стороны других процессоров.

В *архитектуре с централизованным устройством управления* (ЦУУ) устройство управления выполняет следующие функции: трассирует потоки данных между процессорами, памятью, устройствами ввода/вывода; буферизирует запросы; выполняет синхронизацию и арбитраж; отслеживает состояние процессоров и их кэш-памяти. Недостаток архитектуры заключается в сложности ЦУУ, которое может ограничивать производительность. В настоящее время подобная архитектура встречается редко, но она широко использовалась при создании вычислительных систем на базе машин семейства IBM 370.

На практике наиболее распространена архитектура с общей шиной, и именно такие SMP-системы будут обсуждаться при сравнении с MIMD-системами других видов.

Обобщая сказанное, SMP можно определить как вычислительную систему, обладающую следующими характеристиками:

- Имеются два или более процессоров сопоставимой производительности.
- Процессоры совместно используют основную память и работают в едином виртуальном и физическом адресном пространстве.
- Все процессоры связаны между собой посредством шины или по иной схеме, так что время доступа к памяти любого из них одинаково.
- Все процессоры разделяют доступ к устройствам ввода/вывода либо через одни и те же каналы, либо через разные каналы, обеспечивающие доступ к одному и тому же внешнему устройству.
- Все процессоры способны выполнять одинаковые функции;
- Любой из процессоров может обслуживать внешние прерывания;
- Вычислительная система управляется единой операционной системой, которая организует и координирует взаимодействие между процессорами и программами на уровне заданий, задач, файлов и элементов данных.

SMP-система представляется конечному пользователю как единая ВМ высокой производительности. В этом качестве небольшие SMP-системы из 2–4 процессорных элементов часто используются в качестве процессорных элементов в MIMD-системах других видов. Отметим негативные свойства SMP-систем. Во-первых, при увеличении числа процессоров стоимость системы растет быстрее, чем производительность. Во-вторых, из-за задержек при одновременном обращении нескольких ПЭ к общей памяти (даже если они параллельно выполняют независимые программы) происходит взаимное торможение.

Параллельные векторные системы

Еще один вид MIMD-систем с однородным доступом к разделяемой памяти — параллельные векторные системы (PVP, Parallel Vector Processor). По своей сути — это SMP-системы, где роль процессорных элементов исполняют векторно-конвейерные процессоры. PVP-система содержит сравнительно небольшое число индивидуальных векторных процессоров, связанных широкополосным коммутатором типа «кроссбар» (рис. 14.3). Благодаря кроссбару количество ПЭ в системе может быть больше, чем в SMP-системах с шинной организацией, и может достигать 512, хотя типовые значения — 8–16 процессорных элементов.

PVP-системы ориентированы на приложения из таких областей промышленности, как аэрокосмическая, автомобильная, электронная, химическая, энергетическая и т. п. Разработчики PVP-систем предлагают пользователям компиляторы с эффективными средствами автоматической векторизации и распараллеливания программных вычислений. Получаемый объектный код позволяет наилучшим образом использовать потенциал множества векторных процессорных элементов. Кроме того, нужно учитывать, что передача данных в векторном формате происходит на два порядка быстрее, чем в скалярном. Как следствие, затраты времени на взаимодействие между параллельными потоками данных существенно снижаются, что ощутимо сказывается на общей производительности системы.

Основными производителями, поддерживающими производство PVP-систем, являются японские фирмы NEC, Fujitsu и Hitachi. Подобную архитектуру имеют системы семейства VPP (Fujitsu). Каждый процессорный элемент системы состоит из

векторного и скалярного функциональных устройств, блока памяти и устройства сопряжения. ПЭ связаны коммутатором типа «кроссбар». В зависимости от модели система может включать от 8 до 256 процессорных элементов. В максимальном варианте пиковая производительность ВС достигает 4,9 TFLOPS. Работу системы поддерживает специализированная операционная система, основанная на ОС UNIX.

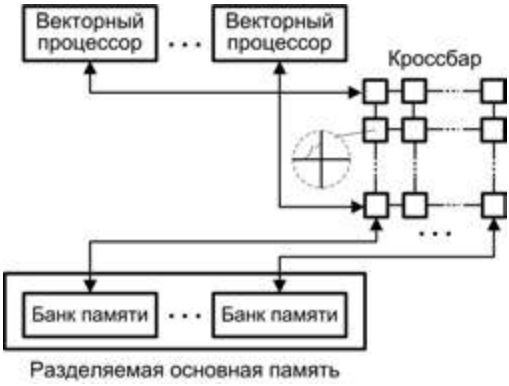


Рис. 14.3. Архитектура PVP-системы

Вычислительные системы с неоднородным доступом к памяти

В отличие от SMP-систем с разделяемой памятью, где максимальное количество ПЭ колеблется в диапазоне 16–64, в этих системах реализована технология неоднородного доступа к памяти (NUMA, Non-Uniform Memory Access).

Технология NUMA считается одним из путей создания крупномасштабных вычислительных систем. В архитектуре NUMA память физически распределена, но логически общедоступна. Это позволяет сохранить преимущества архитектуры с единым адресным пространством, а также ощутимо расширяет возможности масштабирования ВС.

На рис. 14.4 показана типичная организация системы типа ccNUMA. Имеется множество независимых составляющих ВС (узлов), объединенных с помощью какой-либо сети соединений (например, кроссбара, кольца и т. д.). Узел содержит процессор с кэш-памятью, а также локальную основную память, рассматриваемую как часть глобальной основной памяти системы.

Согласно технологии неоднородного доступа, каждый узел в системе владеет локальной памятью, но с позиций системы имеет место глобальное адресное пространство, где каждая ячейка любой локальной основной памяти имеет уникальный системный адрес. Когда процессор инициирует доступ к памяти и нужная ячейка отсутствует в его локальной кэш-памяти, организуется операция выборки. Если нужная ячейка находится в локальной памяти, выборка производится с использованием локальной шины. Если же требуемая ячейка хранится в удаленной секции глобальной памяти (локальной памяти другого процессора), то автоматически

формируется запрос, посылаемый по сети соединений на локальную шину узла, где находится запрошенная информация, и уже по ней на подключенную к данной локальной шине кэш-память. Все эти действия выполняются автоматически, прозрачны для процессора и его кэш-памяти.

Рис. 14.4. Организация систем типа ccNUMA

Как и в любой ВС с разделяемой памятью, особое внимание уделяется когерентности кэшей. В подавляющем большинстве NUMA-систем реализована аппаратная поддержка когерентности кэш-памяти процессорных элементов (ccNUMA), хотя известны системы, где такая поддержка отсутствует (nccNUMA). Поскольку для взаимодействия узлов системы используется не шина, а сеть соединений с более сложной топологией, в ccNUMA-системах проблема когерентности решается с помощью протоколов на основе распределенных справочников. Хотя отдельные реализации и отличаются в деталях, общим является то, что каждый узел содержит справочник. Взаимодействуя между собой, справочники позволяют определить физическое расположение любой информации в глобальном адресном пространстве.

Логику функционирования ccNUMA-системы поясним примером (см. рис. 14.4). Пусть процессор узла 2 ($ПР_2$) запрашивает ячейку с адресом 798, расположенную в узле 1. Будет наблюдаться такая последовательность действий:

1. Процессор $ПР_2$ выдает на шину наблюдения узла 2 запрос чтения ячейки 798.
2. Справочник узла 2 видит запрос и распознает, что нужная ячейка не принадлежит к адресному пространству узла 2, а также то, что нужная ячейка находится в узле 1.
3. Справочник узла 2 посылает запрос узлу 1, который принимается справочником узла 1.
4. Справочник узла 1, действуя как заменитель процессора $ПР_2$, запрашивает ячейку 798 (как будто он сам является процессором).
5. Кэш-память узла 1 реагирует тем, что помещает затребованные данные на локальную шину узла 1 (если копия ячейки в кэш-памяти отсутствует, то предварительно организуется ее загрузка в кэш-память из локальной памяти узла 1).
6. Справочник узла 1 перехватывает данные с шины.
7. Считанное значение через сеть соединений передается обратно в справочник узла 2.
8. Справочник узла 2 помещает полученные данные на локальную шину узла 2, действуя при этом как заместитель той части памяти, где эти данные фактически хранятся.

9. Данные перехватываются и передаются в кэш-память процессора $ПР_2$ и уже оттуда попадают в процессор $ПР_2$.

В реальных NUMA-системах узлы обычно содержат не одиночные процессорные элементы, а сборки из нескольких ПЭ, чаще всего — SMP-системы. Так, одна из наиболее производительных ВС — Tera 10 — состоит из 544 SMP-узлов, каждый из которых содержит от 8 до 16 процессоров Itanium 2.

NUMA-системы, как правило, работают под управлением единой операционной системы.

Масштабируемость NUMA-систем ограничена лишь величиной адресного пространства, возможностями аппаратных средств поддержки когерентности кэшей и возможностями операционной системы по управлению большим числом процессоров. Например, NUMA-система Silicon Graphics Origin поддерживает до 1024 процессоров R10000 [162], а система Sequent NUMA-Q объединяет 252 процессора Pentium II [113]. Очередным этапом развития технологии NUMA стала архитектура NumaFlex, используемая в семействе SGI 3000, где допускается наращивание системы даже за счет различных процессоров.

Подводя итог, дадим следующую характеристику вычислительных систем с неоднородным доступом к памяти:

- Имеются два или более узлов, каждый из которых может быть представлен процессорным элементом с кэш-памятью или SMP-системой.
- Каждый узел имеет локальную память, рассматриваемую как часть глобальной памяти системы.
- Узлы соединены посредством высокоскоростной сети, в роли которой чаще всего выступает коммутатор типа «кроссбар».
- Поддерживается единое адресное пространство, то есть любой узел имеет доступ к памяти других узлов, однако время доступа к локальной памяти существенно меньше времени обращения к удаленной памяти (локальной памяти других узлов).
- Когерентность кэшей обычно поддерживается аппаратными средствами (ccNUMA), но возможен вариант без таких средств (nccNUMA).
- Вычислительная система управляется единой операционной системой, как в SMP, но возможен вариант с разбиением системы на разделы, работающие под управлением разных операционных систем.

Благодаря своей архитектуре и заложенному в нее потенциалу масштабируемости NUMA-системы считаются перспективным направлением при создании мощных вычислительных систем.

MIMD-системы с распределенной памятью

Вычислительные системы с распределенной памятью, в сущности, представляют собой совокупность взаимодействующих вычислительных машин, каждая со своей процессорной частью и основной памятью (далее будем называть составляющие таких систем узлами). В отличие от ранее рассмотренных MIMD-систем, процессор

узла имеет доступ только к своей локальной памяти и не имеет возможности напрямую обратиться к памяти другого узла. Взаимодействие узлов обеспечивается путем обмена сообщениями.

В рамках MIMD-систем с распределенной памятью можно выделить три семейства: системы с массовой параллельной обработкой (MPP), кластеры вычислительных машин и кластеры больших SMP-систем (Constellations — «созвездия»).

Системы с массовой параллельной обработкой (MPP)

Основные причины появления *систем с массовой параллельной обработкой* (MPP, Massively Parallel Processing) — это, во-первых, необходимость построения ВС с гигантской производительностью и, во-вторых, стремление раздвинуть границы производства ВС в большом диапазоне производительности и стоимости. Для MPP-системы, в которой количество вычислительных узлов может меняться в широких пределах, всегда реально подобрать конфигурацию с заранее заданной вычислительной мощностью и финансовыми вложениями.

MPP-система состоит из множества однородных вычислительных узлов, число которых может исчисляться тысячами. Узел содержит полный комплекс устройств, необходимых для независимого функционирования (процессор, память, подсистему ввода/вывода, коммуникационное оборудование), то есть, по сути, является полноценной вычислительной машиной. Узлы объединены коммуникационной сетью с высокой пропускной способностью и малыми задержками.

Работу узлов MPP-системы координирует главная управляющая вычислительная машина (хост-компьютер). Это может быть отдельная ВМ, как, например, в Cray T3D, или один из узлов системы. В последнем случае функции главной ВМ возлагаются на какой-то определенный узел ВС (в течение всего сеанса вычислений). Напомним, что в SMP-системе особые полномочия одному из ПЭ придаются только на время загрузки системы, после чего он опять становится равноправным элементом системы.

Обобщенная структура MPP-системы показана на рис. 14.5.

Рис. 14.5. Структура вычислительной системы с массовой параллельной обработкой

Если система содержит выделенный хост-компьютер, то полноценная операционная система (ОС) функционирует только на нем, а на узлы устанавливается ее

урезанный вариант, поддерживающий лишь функции ядра ОС. При отсутствии главной ВМ полноценная ОС устанавливается на каждый узел MPP-системы. Таким образом, каждый узел функционирует под управлением собственной операционной системы.

Хост-компьютер (или его заменитель из числа узлов) распределяет задания между множеством подчиненных ему узлов. Схема взаимодействия в общих чертах довольно проста:

- хост-компьютер формирует очередь заданий, каждому из которых назначается некоторый уровень приоритета;
- по мере освобождения узлов им передаются задания из очереди;
- узлы оповещают хост-компьютер о ходе выполнения задания; в частности о завершении выполнения или о потребности в дополнительных ресурсах;
- у хост-компьютера имеются средства для контроля работы подчиненных процессоров, в том числе для обнаружения нештатных ситуаций, прерывания выполнения задания в случае появления более приоритетной задачи и т. п.

В некотором приближении имеет смысл считать, что на главной ВМ выполняется ядро операционной системы (планировщик заданий), а на подчиненных ей узлах — приложения. Подчиненность может быть реализована как на аппаратном, так и на программном уровне.

Процессоры в узлах имеют доступ только к своей локальной памяти. Для доступа к памяти другого узла пара узлов (клиент и «сервер») должна обменяться сообщениями. Такой режим исключает возможность конфликтов, возникающих в разделяемой памяти при одновременном обращении к ней со стороны нескольких процессоров. Снимается также проблема когерентности кэш-памяти. Как следствие, появляется возможность наращивания количества процессоров до нескольких тысяч. По этой причине основным признаком, по которому вычислительную систему относят к MPP-типу, часто служит количество процессоров n. Строгой границы не существует, но обычно при $n \geqslant 128$ считается, что это уже MPP. На самом деле, отличительной чертой MPP является ее архитектура, позволяющая добиться высочайшей производительности. Благодаря свойству масштабируемости MPP-системы являются сегодня лидерами по достигнутой производительности. Так, список наиболее производительных ВС на конец 2009 года возглавляла MPP-система Jaguar Cray XT5-HE с теоретической пиковой производительностью 2331 TFLOPS. Система состоит из 224 256 вычислительных ядер. Каждое ядро содержит два 6-ядерных процессора Opteron, 16GB памяти и маршрутизатор — сетевое устройство, обеспечивающее объединение узлов ВС и передачу сообщений по этой сети. Помимо системы Jaguar в десятку лидеров по производительности входят еще 5 вычислительных систем с архитектурой MPP.

Следует отметить, что распараллеливание вычислений в MPP-системах — трудная задача. Достаточно сложно найти задания, которые сумели бы эффективно загрузить множество вычислительных узлов. Сегодня не так уж много приложений могут эффективно выполняться на MPP-системе. Появляется также проблема переносимости программ между системами с различной архитектурой. Эффективность распараллеливания во многих случаях сильно зависит от деталей архитектуры MPP-системы, например топологии соединения вычислительных узлов.

Самой эффективной была бы топология, в которой любой узел мог бы напрямую связаться с любым другим узлом, но в ВС на основе MPP это технически нереализуемо. Если в первых MPP-компьютерах использовались топологии двухмерной решетки (SGI/Pyramid RM1000) и гиперкуба (nCube [9]), то в современных наиболее масштабируемых и производительных ВС, например семействах XT (Cray) и Blue Gene/P (IBM), применяют трехмерный тор. Диаметр такой сети для различных моделей упомянутых систем лежит в диапазоне от 20 до 60, он существенно влияет на задержки в передаче сообщений. Когда множественные сообщения начинают разделять ресурсы сети, данный эффект (применительно к сообщениям большого размера) становится очевидным. Таким образом, при распределении задач по узлам ВС необходимо учитывать топологию системы.

Время передачи информации от узла к узлу зависит от стартовой задержки и скорости передачи. В любом случае, за время передачи процессорные узлы успевают выполнить много команд, и такое соотношение (быстродействия процессорных узлов и передающей системы), вероятно, будет сохраняться — прогресс в производительности процессоров гораздо весомее, чем в пропускной способности каналов связи. Поэтому инфраструктура каналов связи в MPP-системах является объектом наиболее пристального внимания разработчиков.

Слабым местом MPP является хост-компьютер — при выходе его из строя вся система оказывается неработоспособной. Повышение надежности главной ВМ лежит на путях упрощения аппаратуры хост-компьютера и/или ее дублирования.

Несмотря на все сложности, сфера применения ВС с массовым параллелизмом постоянно расширяется. Различные системы этого класса эксплуатируются во многих ведущих суперкомпьютерных центрах мира. Следует особенно отметить тот факт, что мировой лидер производства векторных ВС, компания Cray Research, уже не ориентируется исключительно на векторные системы, отдавая предпочтение MPP. На ноябрь 2009 года доля MPP-систем составляет 16,2%.

Подводя итог, главные особенности, по которым вычислительную систему причисляют к классу MPP, можно сформулировать следующим образом:

- вычислительный узел обладает всеми средствами для независимого функционирования (стандартные микропроцессоры с кэш-памятью, локальная память, подсистема ввода/вывода);
- каждый узел содержит сетевой адаптер, используемый для объединения узлов;
- реализована модель распределенной памяти — доступ к памяти других узлов обеспечивается путем асинхронного обмена сообщениями;
- сеть соединений обычно проектируется под конкретную систему для обеспечения высокой пропускной способности и малых задержек;
- система хорошо масштабируется (до тысяч узлов);
- работа системы координируется главной ВМ (хост-компьютером) или одним из узлов, выполняющим роль главной ВМ;
- на каждом узле установлена операционная система или ее урезанный вариант, если имеется хост-компьютер с полноценной операционной системой;
- вычисления представляют собой множество процессов, имеющих отдельные адресные пространства.

Кластерные вычислительные системы

Одно из самых современных направлений в области создания вычислительных систем — *кластеризация*. Помимо термина «кластерные вычисления», достаточно часто применяют такие названия: *кластер рабочих станций* (workstation cluster), *гипервычисления* (hypercomputing), *параллельные вычисления на базе сети* (network-based concurrent computing), *ультравычисления* (ultracomputing).

Кластером называют группу взаимно соединенных вычислительных систем (узлов), работающих совместно и составляющих единый вычислительный ресурс, создавая иллюзию наличия единственной ВМ. Изначально перед кластерами ставились две задачи: достичь большой вычислительной мощности и обеспечить повышенную надежность ВС. Пионером в области кластерных архитектур считается корпорация DEC, разработавшая первый коммерческий кластер в начале 80-х годов прошлого века.

Архитектура кластерных систем во многом похожа на архитектуру MPP-систем. Тот же принцип распределенной памяти, использование в качестве вычислительных узлов законченных вычислительных машин, большой потенциал для масштабирования системы и целый ряд других особенностей. В первом приближении кластерную технологию можно рассматривать как развитие идей массовых параллельных вычислений. С другой стороны, многие черты кластерной архитектуры дают основание считать ее самостоятельным направлением в области MIMD-систем.

Обобщенная структура кластерной вычислительной системы показана на рис. 14.6.

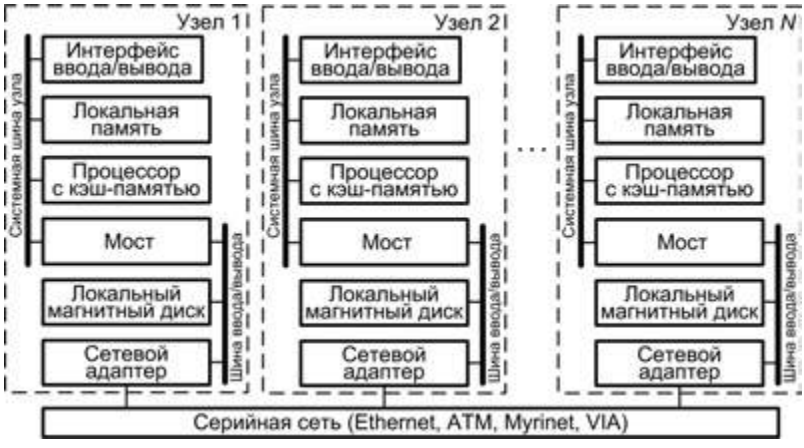


Рис. 14.6. Структура кластерной системы

В качестве узла кластера может выступать как однопроцессорная ВМ, так и ВС типа SMP (напомним, что логически SMP-система представляется как единственная ВМ). Как правило, это не специализированные устройства, приспособленные под использование в вычислительной системе, как в MPP, а серийно выпускаемые вычислительные машины и системы. Еще одна особенность кластерной архитектуры состоит в том, что в единую систему объединяются узлы разного типа,

от персональных компьютеров до мощных ВС. Кластерные системы с одинаковыми узлами называют *гомогенными кластерами*, а с разнотипными узлами — *гетерогенными кластерами*.

Использование машин массового производства существенно снижает стоимость ВС, а возможность варьирования различных по типу узлов позволяет получить необходимую производительность за приемлемую цену. Важно и то, что узлы могут функционировать самостоятельно и отдельно от кластера. Для этого каждый узел работает под управлением своей операционной системы. Чаще всего используются стандартные ОС: Linux, FreeBSD, Solaris и версии Windows, продолжающие направление Windows NT.

Узлы в кластерной системе объединены высокоскоростной сетью. Решения могут быть простыми, основанными на аппаратуре Ethernet, или сложными с высокоскоростными сетями пропускной способности в сотни мегабайтов в секунду (Мбит/с). К последней категории относятся сети SCI компании Scali Computer (~ 100 Мбит/с) и Mirynet (~ 120 Мбит/с). В принципе, за основу кластерной системы может быть взята стандартная локальная сеть (или сеть большего масштаба), с сохранением принятых в ней протоколов (правил взаимодействия). Аппаратурные изменения могут не потребоваться или, в худшем случае, сводятся к замене коммуникационного оборудования на более производительное. При соединении машин в кластер почти всегда поддерживаются прямые межмашинные связи.

Вычислительные машины (системы) в кластере взаимодействуют в соответствии с одним из двух транспортных протоколов. Первый из них, протокол TCP (Transmission Control Protocol), оперирует потоками байтов, гарантируя надежность доставки сообщения. Второй — UDP (User Datagram Protocol) пытается посылать пакеты данных без гарантии их доставки. В последнее время применяют специальные протоколы, которые работают намного лучше, например Virtual Interface Architecture (VIA).

При обмене информацией используются два программных метода: *передачи сообщений* и *распределенной, совместно используемой памяти*. Первый опирается на явную передачу информационных сообщений между узлами кластера. В альтернативном варианте также происходит пересылка сообщений, но движение данных между узлами кластера скрыто от программиста.

Подключение узлов к сети осуществляется посредством сетевых адаптеров. Учитывая роль коммуникаций, для связи ядра вычислительного узла с сетью используется выделенная шина ввода/вывода. К этой шине также подключаются локальные магнитные диски. Наличие таких дисков типично для кластерных систем, но не характерно для MPP-систем. Элементы вычислительного ядра объединяются посредством локальной системной шины. Связь между этой шиной и шиной ввода/вывода обеспечивает мост.

Неотъемлемая часть кластера — специализированное программное обеспечение (ПО), организующее бесперебойную работу при отказе одного или нескольких узлов. Такое ПО должно быть установлено на каждый узел кластера. Оно реализует механизм передачи сообщений над стандартными сетевыми протоколами и может рассматриваться как часть операционной системы. Именно благодаря

специализированному ПО группа ВМ, объединенных сетью, превращается в кластерную вычислительную систему. Кластерное ПО перераспределяет вычислительную нагрузку при отказе одного или нескольких узлов кластера, а также восстанавливает вычисления при сбое в узле. ПО каждого узла постоянно контролирует работоспособность всех остальных узлов. Этот контроль основан на периодической рассылке каждым узлом сигнала, известного как *keepalive* («пока жив») или *heartbeat* («сердцебиение»). Если сигнал от некоторого узла не поступает, то узел считается вышедшим из строя; ему не дается возможность выполнять ввод/вывод, его диски и другие ресурсы (включая сетевые адреса) переназначаются другим узлам, а выполнявшиеся им программы перезапускаются в других узлах. При наличии в кластере совместно используемых дисков, кластерное ПО поддерживает единую файловую систему.

Кластеры обеспечивают высокий уровень доступности — в них отсутствуют единая операционная система и совместно используемая память, то есть нет проблемы когерентности кэшей.

При создании кластерных систем используется один из двух подходов.

Первый подход применяется для построения небольших кластерных систем, например, на базе небольших локальных сетей организаций или их подразделений. В таком кластере каждая ВМ продолжает работать как самостоятельная единица, одновременно выполняя функции узла кластерной системы.

Второй подход ориентирован на использование кластерной системы в роли мощного вычислительного ресурса. Узлами кластера служат только системные блоки вычислительных машин, компактно размещаемые в специальных стойках. Управление системой и запуск задач осуществляет полнофункциональный хост-компьютер. Он же поддерживает дисковую подсистему кластера и разнообразное периферийное оборудование. Отсутствие у узлов собственной периферии существенно удешевляет систему.

Кластеры хорошо масштабируются путем добавления узлов, что позволяет достичь высочайших показателей производительности. Благодаря этой особенности архитектуры кластеры с сотнями и тысячами узлов положительно зарекомендовали себя на практике. До недавнего времени именно кластерная ВС занимала первую позицию в этом списке самых производительных систем. Речь идет о кластерной системе Roadrunner BladeCenter QS22, созданной компанией IBM. Теоретическая пиковая производительность системы составляет 1376 TFLOPS, состоит она из 122 400 узлов на базе процессоров Opteron и PowerXCell.

В работе [59] перечисляются четыре преимущества, достигаемые с помощью кластеризации.

- **Абсолютная масштабируемость.** Возможно создание больших кластеров, превосходящих по вычислительной мощности даже самые производительные одиночные ВМ. Кластер в состоянии содержать десятки узлов, каждый из которых представляет собой мультипроцессор.
- **Наращиваемая масштабируемость.** Кластер строится так, что его можно наращивать, добавляя новые узлы небольшими порциями. Таким образом, пользователь может начать с умеренной системы, расширяя ее по мере необходимости.

- **Высокий коэффициент готовности.** Поскольку каждый узел кластера — самостоятельная ВМ или ВС, отказ одного из узлов не приводит к потере работоспособности кластера. Во многих системах отказоустойчивость автоматически поддерживается программным обеспечением.
- **Превосходное соотношение цена/производительность.** Кластер любой производительности можно создать, соединяя стандартные «строительные блоки», при этом его стоимость будет ниже, чем у одиночной ВМ с эквивалентной вычислительной мощностью.

В то же время нужно упомянуть и основной недостаток, свойственный кластерным системам, — взаимодействие между узлами кластера занимает гораздо больше времени, чем в других типах ВС.

Кластеры больших SMP-систем

Огромный потенциал масштабирования, свойственный кластерной архитектуре, делает ее очень перспективным направлением в области создания высокопроизводительных вычислительных систем. Масштабирование возможно как за счет увеличения числа узлов, так и путем применения в качестве узлов не одиночных ВМ, а также хорошо масштабируемых вычислительных систем, обычно SMP-типа. Это направление получило настолько широкое развитие, что было выделено в отдельную группу MIMD-устройств — Constellations. Термин, обусловленный названием одной из первых ВС данного типа — Sun «Constellation», —можно перевести как «созвездие». Поскольку русское название подобных MIMD-систем нельзя считать общепринятым, в дальнейшем будем использовать англоязычное именование.

Таким образом, Constellation-система — это кластер, узлами которого служат SMP-системы. В качестве отличительного признака выступает число процессорных элементов в узле. Изначально принималось, что система относится к Constellation-системам, если число узлов в кластере из SMP-систем меньше или равно количеству процессорных элементов в SMP-системе узла. Для современных систем с большим количеством узлов такое условие не всегда соблюдается, поэтому в настоящее время условием причисления ВС к Constellation-системам служит число ПЭ в узле — оно должно быть больше (равно) 16. Системы, не отвечающие данному условию, считаются классическими кластерными системами.

Перспективность Constellation-систем обусловлена удачным сочетанием преимуществ распределенной памяти (возможности наращивания количества узлов) и разделяемой памяти (эффективного доступа множества ПЭ к памяти).

Формально структура Constellation-системы (рис. 14.7) полностью соответствует кластерной архитектуре, однако явная направленность на высокую производительность может отражаться в некоторых конструктивных решениях.

В качестве примера Constellation-системы можно привести систему Tera 10 фирмы Bull с теоретической пиковой производительностью 58,8 TFLOPS. Система представляет собой кластер из 544 узлов, в котором каждый узел — это SMP-система, образованная 8 2-ядерными процессорами Itanium 2.

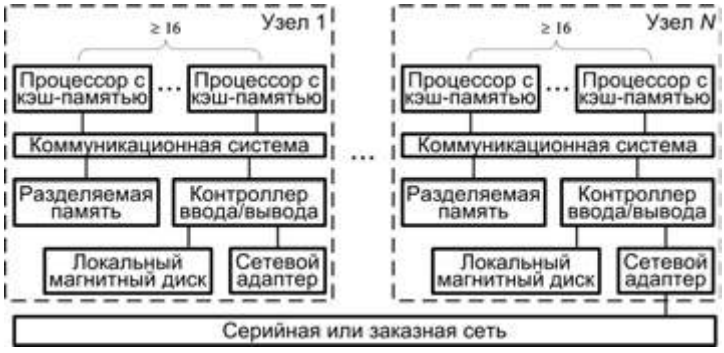


Рис. 14.7. Структура Constellation-системы

Вычислительные системы на базе транспьютеров

Появление транспьютеров связано с идеей создания различных по производительности ВС (от небольших до мощных массивно-параллельных) посредством прямого соединения однотипных процессорных чипов. Сам термин объединяет два понятия — «транзистор» и «компьютер».

Транспьютер — это сверхбольшая интегральная микросхема, заключающая в себе центральный процессор, блок операций с плавающей запятой, статическое оперативное запоминающее устройство, интерфейс с внешней памятью и несколько каналов связи (рис. 14.8).

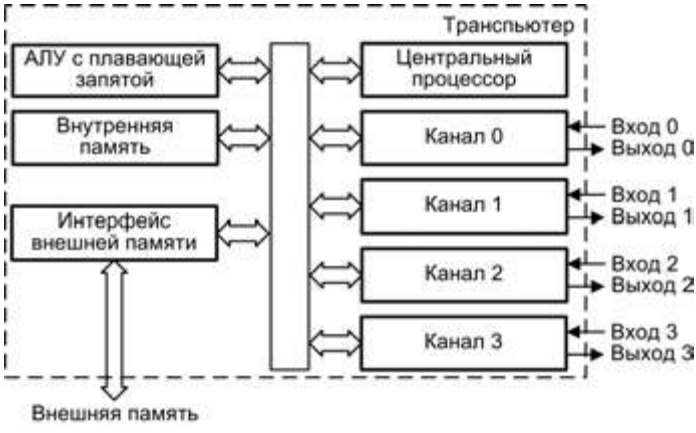


Рис. 14.8. Структура транспьютера

Одна из важнейших отличительных черт транспьютера — каналы связи (линки), благодаря которым транспьютеры можно объединять, создавая вычислительные системы с различной вычислительной мощностью.

Канал связи состоит из двух последовательных линий для двухстороннего обмена. Он позволяет объединить транспьютеры между собой и обеспечить взаимные

коммуникации. Одна из последовательных линий используется для пересылки данных, а вторая — подтверждений. Передача информации производится синхронно под воздействием либо общего генератора тактовых импульсов (ГТИ), либо локальных ГТИ с одинаковой частотой следования импульсов. Информация передается в виде пакетов. Каждый раз, когда пересылается *пакет данных*, приемник отвечает пакетом *подтверждения* (рис. 14.9).

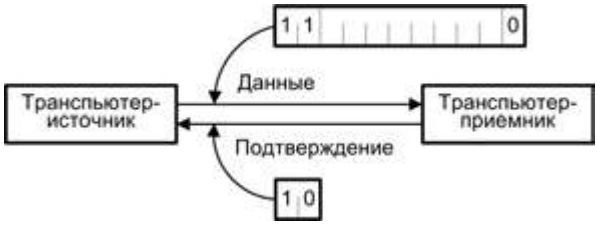


Рис. 14.9. Организация ввода/вывода в транспьютерной системе

Пакет данных состоит из двух битов-единиц, за которыми следуют 8-битовые данные и ноль (всего 11 битов). Пакет подтверждения — это простая комбинация 10 (всего два бита), она может быть передана, как только пакет данных будет идентифицирован интерфейсом входного канала.

На базе транспьютеров легко могут быть построены различные виды ВС. Так, четыре канала связи обеспечивают построение двухмерного массива, где каждый транспьютер связан с четырьмя ближайшими соседями. Возможны и другие конфигурации, например объединение транспьютеров в группы с последующим соединением групп между собой. Если группа состоит из двух транспьютеров, для подключения ее к другим группам свободными остаются шесть каналов связи (рис. 14.10, *а*). Комплекс из трех транспьютеров также оставляет свободными шесть каналов (рис. 14.10, *б*), а для связи с «квартетом» транспьютеров остаются еще четыре канала связи (рис. 14.10, *в*). Группа из пяти транспьютеров может иметь полный набор взаимосвязей, но за счет потери возможности подключения к другим группам. Наконец, транспьютерную систему можно создать на основе кроссбара (рис. 14.10, *г*).

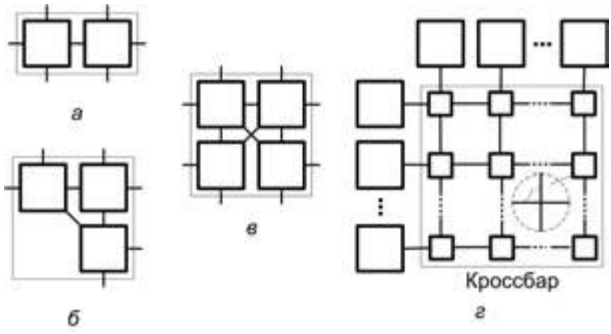


Рис. 14.10. Группы из полностью взаимосвязанных транспьютеров: *а* — два транспьютера; *б*— три транспьютера; *в* — четыре транспьютера; *г* — система на базе кроссбара

Особенности транспьютеров потребовали разработки для них специального языка программирования *Occam*. Название языка связано с именем философа-схоласта четырнадцатого века Оккама — автора концепции «бритвы Оккама»: «entia non sunt multiplicanda praeter necessitatem» — «понятия не должны усложняться без необходимости». Язык обеспечивает описание простых операций пересылки данных между двумя точками, а также позволяет явно указать на параллелизм при выполнении программы несколькими транспьютерами. Основным понятием программы на языке Occam является *процесс*, состоящий из одного или более операторов программы, которые могут быть выполнены последовательно или параллельно. Процессы могут быть распределены по транспьютерам вычислительной системы, при этом оборудование транспьютера поддерживает совместное использование транспьютера одним или несколькими процессами.

Транспьютеры успешно использовались в различных областях от встроенных систем до суперЭВМ вплоть до 90-х годов прошлого века. Интересные возможности, связанные с построением вычислительных систем без привлечения дополнительного оборудования, оказали ощутимое влияние на архитектурные идеи матричных, систолических, SMP и MPP вычислительных систем. Однако появление универсальных микропроцессоров, также обладающих возможностью соединения их в систему, привело к прекращению производства транспьютеров и их вытеснению похожими разработками ведущих фирм, которые уже не позиционируются как транспьютеры.

Тенденции развития высокопроизводительных вычислительных систем

Разнообразие архитектур вычислительных систем порождено, главным образом, стремлением разработчиков создать ВС максимально возможной производительности. В то же время именно различия в организации ВС не позволяют однозначно отдать предпочтение какой-либо одной архитектуре. Данные о производительности, предоставляемые разработчиками ВС, не всегда объективны. Для сравнительной оценки существующих высокопроизводительных ВС и отслеживания тенденций в их развитии необходим источник объективной информации. Таким источником обычно служит рейтинг Top500 самых мощных ВС, составляемый дважды в год (начиная с 1993 года) общепризнанными экспертами в области вычислительной техники. Выбор наиболее производительных ВС и их ранжирование осуществляется на основании показателей производительности, получаемых при выполнении теста LINPACK. Помимо этого публикуется информация о тестируемых системах, которая представляется в виде различного рода статистических отчетов. Рейтинг доступен на сайте http://www.top500.org/. Наличие упомянутых данных позволяет оценить основные тенденции и перспективы в развитии высокопроизводительных вычислительных систем. Разнообразие информации в Top500 и возможность представления ее в различных формах позволяет любому желающему самостоятельно проанализировать состояние дел в области высокопроизводительных вычислений. В учебнике оценим лишь наиболее общие тенденции развития ВС, касающиеся их архитектуры и организации, исходя из последних имеющихся данных.

Прежде всего обратимся к диаграмме, отражающей распространенность различных архитектур среди наиболее производительных ВС (рис. 14.11).

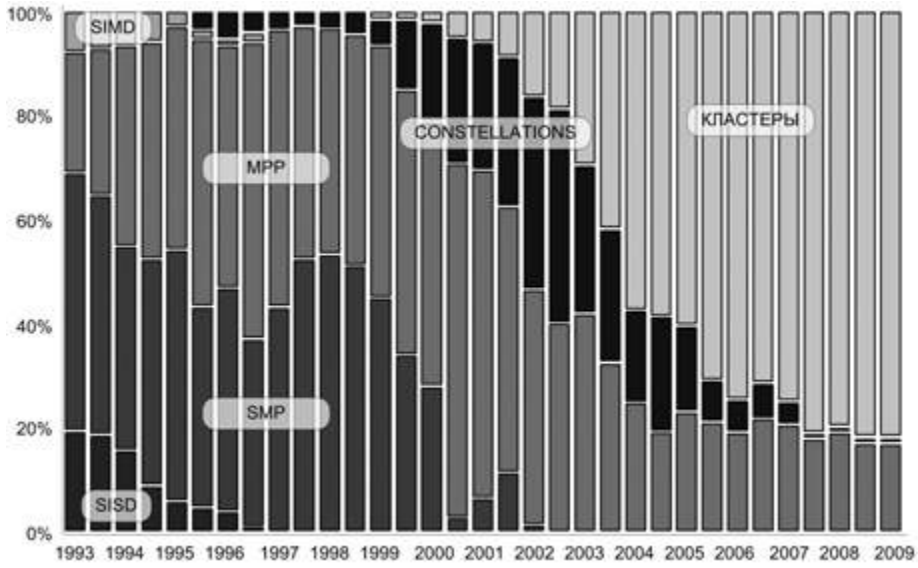


Рис. 14.11. Архитектуры наиболее производительных вычислительных систем

Из графика видно, каким архитектурам ВС отдавалось предпочтение в разные годы. В 2009 году среди 500 наиболее мощных систем были представлены только три архитектуры: кластерная (83,4%), MPP (16,2%) и Constellation (0,4%). Характерно, что все это — MIMD-системы.

Из приведенной статистики видно, что повышение производительности обеспечивалось в основном в рамках кластерных систем. С другой стороны, данные рейтинга на ноябрь 2009 года (табл. 14.1) свидетельствуют о том, что доминирующее положение по производительности в настоящее время занимают MPP-системы.

Таблица 14.1. 10 наиболее производительных вычислительных систем на ноябрь 2009 года

Позиция	Система	Архитектура	Максимальная производительность по LINPACK, TFLOPS
1	Jaguar	MPP	1759
2	Roadrunner	Кластер	1042
3	Kraken	MPP	831,7
4	Blue Gene/P	MPP	825,5
5	Tianhe-1	Кластер	563,1
6	Pleiades	MPP	544,3
7	BlueGene/L	MPP	478,2
8	Blue Gene/P Solution	MPP	458,6
9	Ranger	Кластер	433,2
10	Red Sky	Кластер	423,9

Рисунок 14.12 иллюстрирует прогресс в достигнутом уровне производительности ВС и ожидаемый прогресс в ближайшие годы. На диаграмме показана суммарная вычислительная мощность ВС представленных в Top500 (SUM), их средняя производительность (N = 500), и показатели производительности ВС, находившейся в определенном году на вершине топа (N = 1).

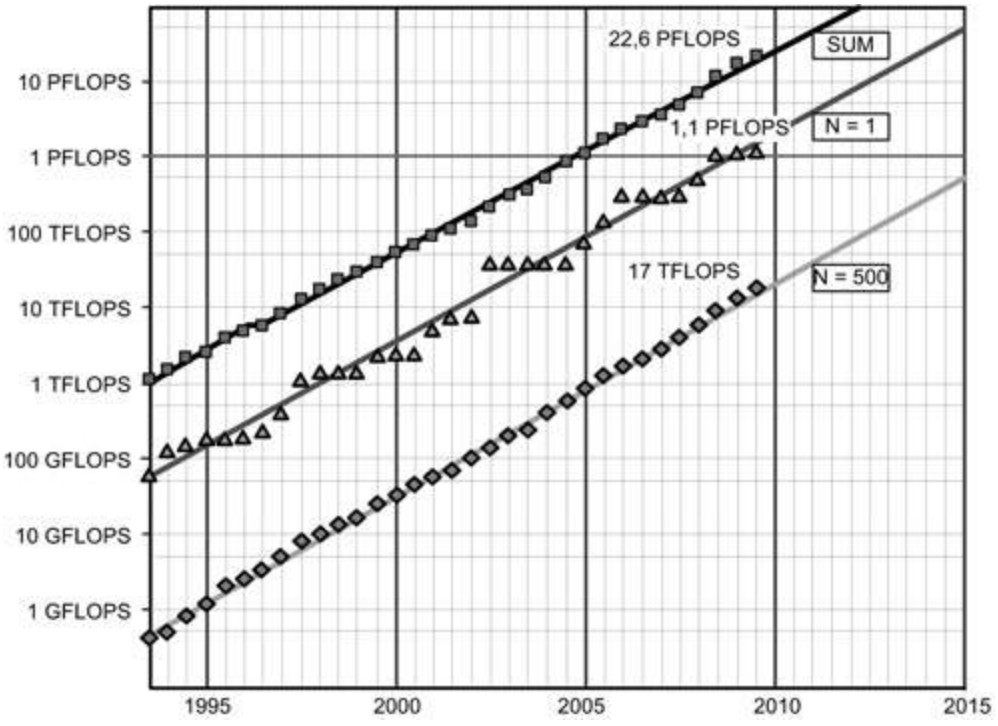


Рис. 14.12. Тенденции роста производительности ВС и прогноз на будущее

В течение последних 5-ти лет (2005–2010) максимальная достигнутая производительность ВС возросла в 23 раза (88% в год). Экстраполяция данных по производительности, произведенная по данным Top500, дает следующие прогнозируемые значения (в PFLOPS): 2011 г. — 7; 2013 г. — 25; 2015 г. — 90; 2019 г. — 1000.

Контрольные вопросы

1. По какому признаку вычислительную систему можно отнести к сильно связанным или слабо связанным ВС?
2. Какие уровни параллелизма реализуют симметричные мультипроцессорные системы?
3. Какими средствами поддерживается когерентность кэш-памяти в SMP-системах?
4. Оцените достоинства и недостатки различных SMP-архитектур.

5. В чем состоит принципиальное различие между матричными и симметричными мультипроцессорными вычислительными системами?
6. Какие две проблемы призвана решить кластерная организация вычислительной системы?
7. Существуют ли ограничения на число узлов в кластерной ВС? И если существуют, то чем они обусловлены?
8. Какие задачи в кластерной вычислительной системе возлагаются на специализированное (кластерное) программное обеспечение?
9. Каким образом может быть организовано взаимодействие между узлами кластерной ВС?
10. При каком количестве процессоров ВС можно отнести к системам с массовой параллельной обработкой?
11. Как организуется координация процессоров и распределение между ними заданий в MPP-системах?
12. Какие топологии можно считать наиболее подходящими для MPP-систем и почему?
13. Поясните назначение справочника в вычислительных системах типа ccNUMA.
14. Какие протоколы когерентности, на ваш взгляд, наиболее подходят для ВС, построенных по технологии ccNUMA?
15. Какие черты транспьютера отличают его от стандартной однокристальной ВМ?
16. Какими аппаратными и программными средствами поддерживается взаимодействие соседних транспьютеров в вычислительной системе?
17. Сколько линий поддерживает канал связи транспьютера, как они используются и в каком режиме осуществляется ввод/вывод?
18. Какие особенности транспьютеров облегчает реализовать язык Occam?
19. Опишите структуру пакета данных и пакета подтверждения, передаваемых в транспьютерных ВС.
20. Какие из рассмотренных типов вычислительных систем могут быть построены на базе транспьютеров и в каких случаях это наиболее целесообразно?

Глава 15

Вычислительные системы с нетрадиционным управлением вычислениями

Большинство современных ВМ построены в соответствии с фон-неймановской концепцией программного управления. То же самое можно сказать и о традиционных многопроцессорных системах или, по крайней мере, о процессорных элементах, из которых они образованы. Согласно этой концепции порядок вычислений определяет программа, представленная последовательностью команд. Эта последовательность сохраняется и при размещении программы в памяти. Команды программы, как правило, выполняются в порядке их следования в программе. Это обеспечивается наличием в каждом процессоре счетчика команд. Выполнение команд в каждом процессоре — поочередное и потому достаточно медленное. Для получения выигрыша программист или компилятор должны определить независимые команды, которые могут быть поданы на отдельные процессоры, причем так, чтобы коммуникационные издержки были не слишком велики.

Традиционные (фон-неймановские) вычислительные системы, управляемые с помощью счетчика команд, иногда называют *вычислительными системами, управляемыми последовательностью команд* (control flow computer). Данный термин применяется, когда нужно выделить этот тип ВС из альтернативных типов, где последовательность выполнения команд определяется не центральным устройством управления со счетчиком команд, а каким-либо иным способом. В вычислительных системах используют три механизма управления последовательностью вычислений:

- команда выполняется, после того как выполнена предшествующая ей команда последовательности;
- команда выполняется, когда становятся доступными ее операнды;
- команда выполняется, когда другим командам требуется результат ее выполнения.

Первый метод соответствует традиционному механизму с управлением последовательностью команд; второй механизм известен как *управляемый данными* (data driven) или *потоковый* (dataflow); третий вариант называют механизмом *управления по запросу* (demand driven) или *редукционным*.

Общие идеи нетрадиционных подходов к организации вычислительного процесса показаны на рис. 15.1, а их более детальному изложению посвящена данная глава.

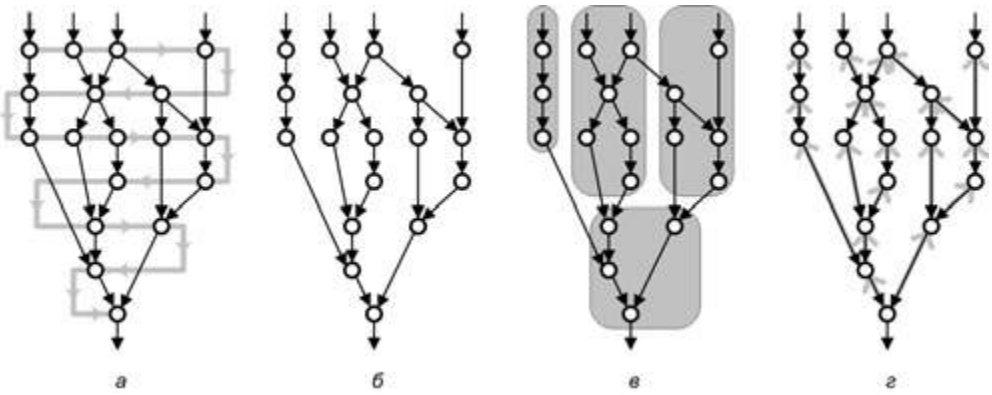


Рис. 15.1. Возможные вычислительные модели: *а* — фон-неймановская; *б* — потоковая; *в* — мультипотоковая; *г* — редукционная

Вычислительные системы с управлением от потока данных

Идеология вычислений, управляемых потоком данных (потоковой обработки), была разработана в 60-х годах Карпом и Миллером. В начале 70-х годов Деннис, а позже и другие начали разрабатывать компьютерные архитектуры, основанные на вычислительной модели с управлением от потока данных.

Вычислительная модель потоковой обработки

В потоковой вычислительной модели для описания вычислений используется ориентированный граф, иногда называемый *графом потоков данных* (dataflow graph). Этот граф состоит из *вершин* (узлов), отображающих операции, и *ребер* или *дуг*, показывающих потоки данных между теми вершинами графа, которые они соединяют.

Операция в вершине выполняется лишь когда по дугам в нее поступила вся необходимая информация. Обычно такая операция требует одного или двух операндов, а для условных операций необходимо наличие входного логического значения. Операция формирует один, два или один из двух возможных результатов. Таким образом, у каждой вершины может быть от одной до трех входящих дуг и одна или две выходящих. После активации вершины и выполнения операции результат передается по дуге к ожидающей вершине. Процесс повторяется, пока не будут активированы все вершины и получен окончательный результат. Одновременно могут

быть активированы несколько вершин, при этом параллелизм в вычислительной модели выявляется автоматически.

На рис. 15.2, *а* показан простой потоковый граф для вычисления выражения $f = a/b + b \times c$. Входами служат переменные *a*, *b* и *c*, изображенные вверху графа. Дуги между вершинами показывают информационную взаимосвязь операций. Направление вычислений — сверху вниз. Используются три вычислительные операции: сложение, умножение и деление. Заметим, что *b* требуется в двух вершинах. Вершина копирования dup предназначена для формирования дополнительной копии переменной *b*.

Данные (операнды/результаты), перемещаемые вдоль дуг, содержатся в опознавательных информационных кадрах, маркерах специального формата — *токенах* (иначе фишках или маркерах доступа). Рисунок 15.2 иллюстрирует движение токенов между узлами. После поступления на граф входной информации токен, содержащий значение *a*, направляется в вершину деления; токен с переменной *b* — в вершину копирования; токен с переменной *c* — в вершину умножения. Активирована может быть только вершина dup, поскольку у нее лишь один вход и на нем уже присутствует токен. Когда токен из вершины dup появится на ее выходных дугах, узлы умножения и деления также получат все необходимые маркеры доступа и могут быть активированы. Последняя вершина ждет завершения операций умножения и деления, то есть когда на ее входе появятся все необходимые токены.

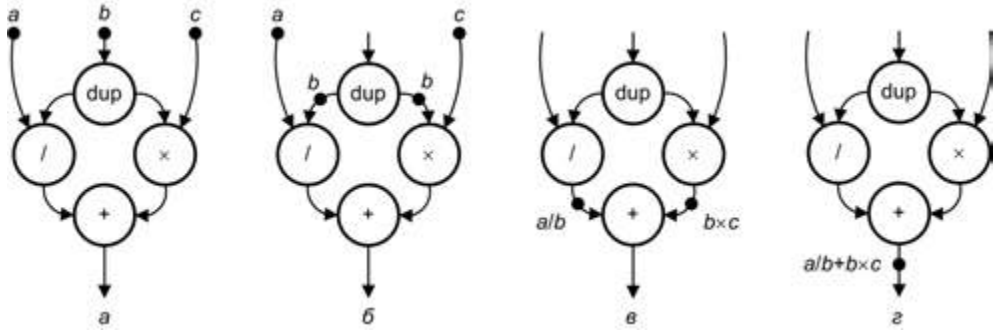


Рис. 15.2. Потоковый граф и движение маркеров при вычислении $a/b + b \times c$: *а* — после подачи входных данных; *б* — после копирования; *в* — после умножения и деления; *г* — после суммирования

Практические вычисления требуют некоторых дополнительных возможностей, например, при выполнении команд условного перехода. По этой причине в потоковых графах предусмотрены вершины-примитивы нескольких типов (рис. 15.3):

- *двухвходовая операционная вершина* — операции производятся над данными, поступающими с левой и правой входных дуг, а результат выводится через выходную дугу;
- *одновходовая операционная вершина* — операции выполняются над входными данными, результат выводится через выходную дугу;

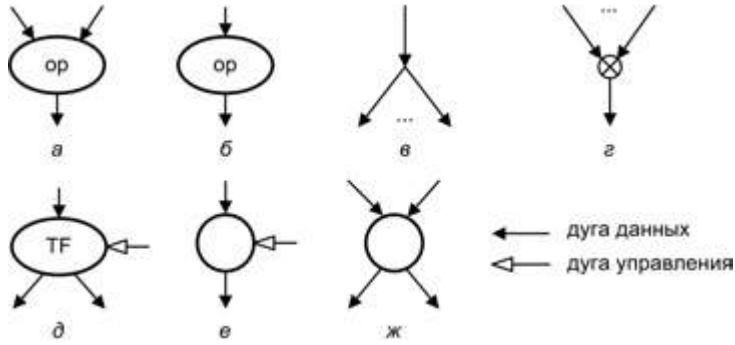


Рис. 15.3. Примитивы узлов: *а* — двухвходовая операционная вершина; *б* — одновходовая операционная вершина; *в* — вершина ветвления; *г* — вершина слияния; *д* — TF-коммутатор; *е* — вентиль; *ж* — арбитр

- *вершина ветвления* осуществляет копирование входных данных и их вывод через две выходных дуги (путем комбинации таких узлов можно строить вершины ветвления на m выходов);
- *вершина слияния* — данные поступают только с какого-нибудь одного из двух входов и без изменения подаются на выход (комбинируя такие узлы, можно строить вершины слияния с m входами);
- *вершина управления* — существует в перечисленных ниже трех вариантах:
  - ☐ *TF-коммутатор* — если значение на управляющем входе истинно (T — True), то входные данные выводятся через левый выход, а при ложном значении на управляющем входе (F — False) данные следуют через правый выход;
  - ☐ *вентиль* — при истинном значении на входе управления данные выводятся через выходную дугу;
  - ☐ *арбитр* — данные, поступившие на один из двух входов первыми, следуют через левую дугу, а прибывшие впоследствии — через правую выходную дугу (активация вершины происходит в момент прихода данных на какой-либо из входов).

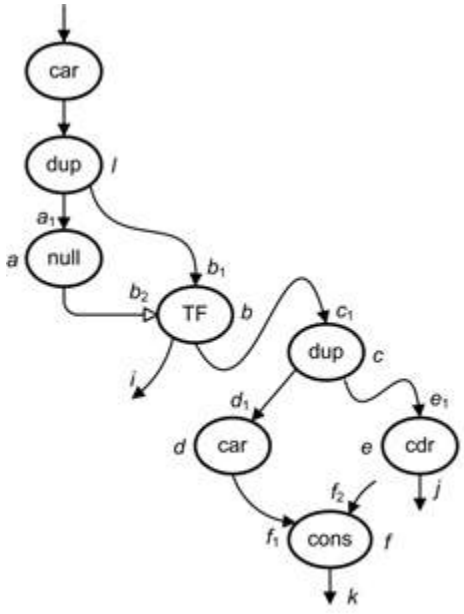
Процесс потоковой обработки может быть аналогичен конвейерному режиму: после обработки первого набора входных сигналов на вход графа может быть подан второй и т. д. Отличие состоит в том, что промежуточные результаты (токены) первого вычисления не обрабатываются совместно с промежуточными результатами второго и последующих вычислений.

Все известные потоковые вычислительные системы могут быть отнесены к двум основным типам: *статическим* и *динамическим*. В свою очередь, динамические потоковые ВС обычно реализуются по одной из двух схем: *архитектуре с помеченными токенами* и *архитектуре с явно адресуемыми токенами*.

Архитектура потоковых вычислительных систем

В потоковых ВС программа вычислений определена потоковым графом, который хранится в памяти системы в виде таблицы. На рис. 15.4 показаны пример графа

потоковой программы и содержание соответствующей ему таблицы [1]. Каждая запись в таблице представляет одну вершину потокового графа. Порядок размещения записей в памяти несущественен, поскольку момент активации вершины определяется лишь наличием данных на всех ее входах. Об этом можно судить по состоянию битов наличия (поле pres). В примере предполагается, что у вершины может быть не более двух входных дуг, поэтому достаточно двух битов наличия. Единичное состояние бита свидетельствует о поступлении данных на соответствующий вход вершины, а также о том, что они находятся в поле операнда, выделенное для этого входа. Если вершина имеет только одну вершину, другой бит сразу устанавливается в единицу. В каждой записи содержится информация, эквивалентная коду операции в обычной машинной команде (поле opc), а также указания о тех вершинах и их входах, куда должен быть передан результат (поле des). Отметим, что в полях операндов (opr1, opr2) должны располагаться значения операндов, а не ссылки на их расположение в памяти. На каждом шаге вычислений одновременно активируются все вершины, в записях которых все биты наличия содержат единицы.



а

opc – код операции;
opr1, opr2 – поля операндов;
des – адрес назначения;
pres – биты наличия данных

	opc	opr1	opr2	des	pres
	car			l	0, 1
l	dup			a_1, b_1	0, 1
a	null			b_2	0, 1
b	TF			*i*, c_1	0, 0
c	dup			d_1, e_1	0, 1
d	car			f_1	0, 1
e	cdr			*j*	0, 1
f	cons			*k*	0, 0

б

Рис. 15.4. Пример формы хранения потоковой программы: *а* — потоковый граф; *б* — представление графа в памяти системы

Статические потоковые вычислительные системы

Статическая потоковая архитектура, известная также под названием «единственный-токен-на-дугу» (single-token-per-arc dataflow), была предложена Деннисом в 1975 году [77]. В ней допускается присутствие на дуге графа не более одного токена. Это выражается в правиле активации вершины [78]: *вершина активируется,*

когда на всех ее входных дугах присутствует по токену и ни на одном из ее выходов токенов нет.

Токен в статической потоковой системе представляет собой триаду: <v, <f, n>, a>, где v — это данные, переносимые токеном, f — идентификатор функции, реализуемой текущим потоковым графом, n — номер целевой вершины, куда должен поступить данный токен, и, наконец, a — это номер дуги, по которой токен должен быть направлен к целевой вершине. Поле <f, n> называют тегом, оно определяет предназначение токена определенной вершине.

Для указания вершине о том, что ее выходной токен уже востребован последующей вершиной (вершинами) графа, в ВС обычно прибегают к механизму подтверждения с квитированием связи, как это показано на рис. 15.5.

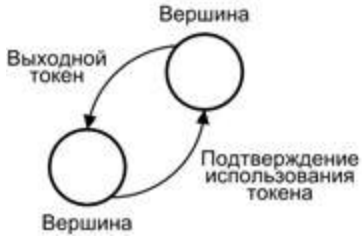


Рис. 15.5. Механизм подтверждения с квитированием

Типовую статическую потоковую архитектуру иллюстрирует рис. 15.6.

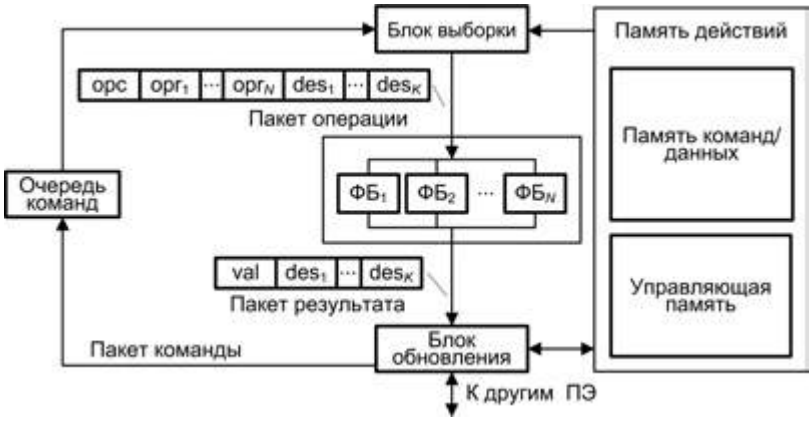


Рис. 15.6. Структура процессорного элемента типовой статической потоковой системы

Память действий состоит из двух блоков памяти: памяти команд/данных и управляющей памяти. Вершина потокового графа представлена в памяти команд/данных кадром, содержащим поле кода операции, N полей операндов и K полей «вершина/дуга». Каждому кадру в памяти команд/данных соответствует кадр в управляющей памяти, содержащий биты наличия операндов и занятости дуг. Бит

наличия операнда устанавливается в единицу, если этот операнд доступен, то есть если токен, содержащий данный операнд, уже поступил по входной дуге графа. Для каждого поля «вершина/дуга» установленный бит занятости означает, что выходная дуга, ассоциированная с данным полем, не содержит токена. Вершина графа, описанная в памяти команд/данных, может быть активирована (операция может быть выполнена), если все биты в соответствующем кадре памяти управления установлены в единицу. Когда данная ситуация распознается блоком обновления, он помещает пакет команды в очередь команд. Опираясь на пакет команды и содержимое памяти действий, блок выборки формирует пакет операции и направляет его в один из свободных функциональных блоков. После выполнения требуемой операции функциональный блок создает пакет результата и передает его в блок обновления, который в соответствии с полученным результатом обновляет содержимое памяти действий.

Основное преимущество рассматриваемой модели потоковых вычислений заключается в упрощенном механизме обнаружения активированных узлов. К сожалению, статическая модель обладает множеством серьезных недостатков [48], однако основной недостаток заключается в том, что данный механизм не допускает параллельного выполнения независимых итераций цикла. Например, операция сложения двух векторов может быть представлена циклическим процессом, где на каждой итерации суммируется пара одноименных элементов векторов-операндов. В статической потоковой ВС такое суммирование должно выполняться последовательно в порядке увеличения индексов элементов. Это связано с тем, что, во-первых, на каждой входной дуге вершины сложения может находиться лишь по одному элементу, а во-вторых, из-за того, что в теге элемента отсутствует информация о его индексе в векторе.

В качестве других примеров статических потоковых ВС можно упомянуть: LAU System [69, 70], TI's Distributed Data Processor [71], DDMI Utah Data Driven Machine [75], NEC Image Pipelined Processor [66], Hughes Dataflow Multiprocessor [160].

Динамические потоковые вычислительные системы

Производительность потоковых систем существенно возрастает, если они в состоянии поддерживать дополнительный уровень параллелизма, соответствующий одновременному выполнению отдельных итераций цикла или параллельной обработке пар элементов в векторных операциях. Кроме того, в современных языках программирования активно используются так называемые *реентерабельные процедуры*, когда в памяти хранится только одна копия кода процедуры, но эта копия является повторно входимой (реентерабельной). Это означает, что к процедуре можно еще раз обратиться, не дожидаясь завершения действий в соответствии с предыдущим входом в данную процедуру. Отсюда желательно, чтобы все обращения к реентерабельной процедуре также обрабатывались параллельно. Задача обеспечения дополнительного уровня параллелизма решается в динамических потоковых ВС и реализуется двумя вариантами архитектуры потоковой ВС: *архитектуры с помеченными токенами* и *архитектуры с явно адресуемыми токенами*.

Архитектура потоковых систем с помеченными токенами

Системы с помеченными токенами (tagged-token architecture) в известной мере свободны от основного недостатка статической модели. В них число токенов, одновременно присутствующих на дуге, не ограничивается. Для приведенного ранее примера циклического суммирования элементов векторов это открывает возможность выполнения итераций в произвольной последовательности, но требует учета принадлежности токенов к одной и той же итерации. С этой целью токен должен содержать информацию о вычислительном контексте, в котором он используется, например о номере итерации цикла. Этот контекст называют «цветом значения», а токен соответственно называют «окрашенным», в силу чего метод имеет еще одно называние — *метод окрашенных токенов*.

В модели с помеченными токенами структура токена сложнее, чем в статической модели: <v, <f, n, c, i>, a>, где c определяет фрагмент кода или тело цикла в составе реализуемой функции f, a i (индекс) представляет цвет токена. Каждая дуга потокового графа может рассматриваться как вместилище, способное содержать произвольное число токенов с различными тегами. Правило активирования вершины в модели с помеченными токенами имеет вид: *вершина активируется, когда на всех ее входных дугах присутствуют токены с идентичным цветом*.

Типовая структура потоковой системы с помеченными токенами показана на рис. 15.7. Для обнаружения одинаково окрашенных токенов (токенов с одинаковыми тегами) в процессорный элемент введен согласующий блок. Этот блок получает очередной токен из очереди токенов и проверяет, нет ли в памяти согласования его партнера (токена с идентичным тегом). Если такой партнер не обнаружен, принятый токен заносится в память согласования. Если же токен-партнер уже хранится в памяти согласования, то согласующий блок удаляет его оттуда и направляет оба токена с совпавшими тегами в блок выборки. На основе общего тега блок выборки находит в памяти команд/данных соответствующую команду и формирует пакет операции, который затем направляет в функциональный блок. Функциональный блок выполняет операцию, создает токены результата и помещает их в очередь токенов.

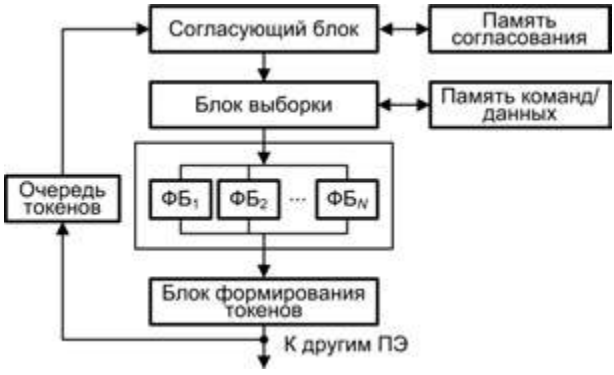


Рис. 15.7. Структура процессорного элемента типовой потоковой системы с помеченными токенами

Примеры динамических потоковых вычислительных систем с архитектурой окрашенных токенов: система Гурда и Ватсона, созданная в Манчестерском университете в 1980 году [89], системы SIGMA-1 [97, 98], NTT's Dataflow Processor Array System [148], DDDP Distributed Data Driven Processor [104], SDFA Stateless Data-Flow Architecture [142].

Основное преимущество динамических потоковых систем с помеченными токенами — повышенная производительность, достигаемая за счет возможности присутствия на дуге множества токенов. При этом, однако, основной проблемой становится эффективная реализация блока, который собирает токены с совпадающим цветом (токены с одинаковыми тегами). В плане производительности этой цели наилучшим образом отвечает ассоциативная память. К сожалению, такое решение является слишком дорогостоящим, поскольку число токенов, ожидающих совпадения тегов, как правило, достаточно велико. По этой причине в большинстве вычислительных систем вместо ассоциативных запоминающих устройств (ЗУ) используются обычные адресные ЗУ. В частности, сравнение тегов в упомянутой выше манчестерской системе производится с привлечением хэширования, что несколько снижает быстродействие.

В последнее время все более популярной становится другая организация динамической потоковой ВС, позволяющая освободиться от ассоциативной памяти и известная как архитектура с явно адресуемыми токенами.

Архитектура потоковых систем с явно адресуемыми токенами

Значительным шагом в архитектуре потоковых ВС стало изобретение механизма *явной адресации токенов* (explicit token-store), имеющего и другое название — *непосредственное согласование* (direct matching). В основе этого механизма лежит наблюдение: все токены в одной и той же итерации цикла и в одном и том же вхождении в реентерабельную процедуру имеют идентичный тег (цвет). При инициализации очередной итерации цикла или очередном обращении к процедуре формируется так называемый кадр токенов, содержащий токены, относящиеся к данной итерации или данному обращению, то есть токены с одинаковыми тегами. Использование конкретных ячеек внутри кадра задается на этапе компиляции. Каждому кадру выделяется отдельная область в специальной памяти кадров (frame memory), причем раздача памяти под каждый кадр происходит уже на этапе выполнения программы.

В схеме с явной адресацией токенов любое вычисление полностью описывается *указателем команды* (IP, Instruction Pointer) и *указателем кадра* (FP, Frame Pointer). Токен выглядит следующим образом: <v, <FP ,IP>>.

Команды, реализующие потоковый граф, хранятся в памяти команд и имеют формат: opc · i · dis. Здесь i (индекс в памяти кадров) определяет положение ячейки с нужным токеном внутри кадра, то есть какое число нужно добавить к FP, чтобы получить адрес интересующего токена. Поле dis указывает на местоположение команды, которой должен быть передан результат обработки данного токена. Адрес в этом поле также задан в виде смещения — числа, которое следует прибавить к текущему значению IP, чтобы получить исполнительный адрес команды назначения

в памяти команд. Если потребителей токена несколько, в поле dis заносится несколько значений смещения.

Простой пример кодирования потокового графа и токенов на его дугах показан на рис. 15.8.

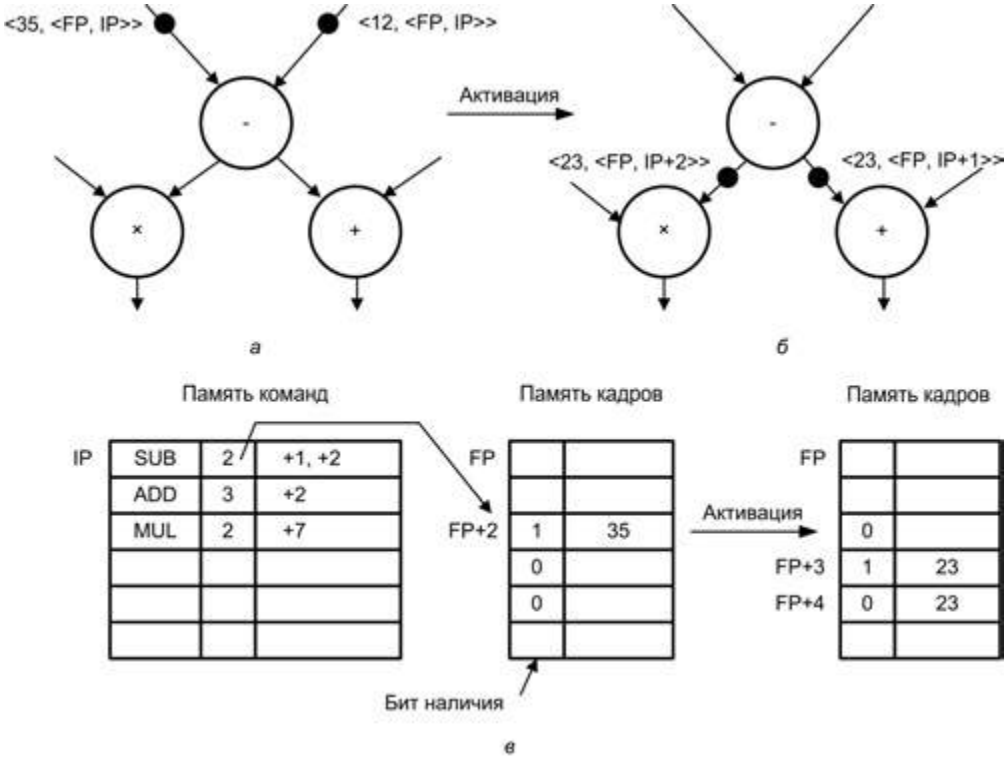


Рис. 15.8. Кодирование в архитектуре с явной адресацией токенов: *а* — активация вершины вычитания; *б* — активация вершин умножения и сложения; *в* — кодирование потокового графа

Каждому слову в памяти кадров придан *бит наличия*, единичное значение которого удостоверяет, что в ячейке находится токен, ждущий согласования, то есть что одно из искомых значений операндов уже имеется. Как и в архитектуре с окрашенными токенами, определено, что вершины могут иметь максимум две входных дуги. Когда на входную дугу вершины поступает токен <v1, <FP, IP>>, в ячейке памяти кадров с адресом FP + (IP.i) проверяется бит наличия (здесь IP.i означает содержимое поля i в команде, хранящейся по адресу, указанному в IP). Если бит наличия сброшен (ни один из пары токенов еще не поступал), поле значения пришедшего токена (v1) заносится в анализируемую ячейку памяти кадров, а бит наличия в этой ячейке устанавливается в единицу, фиксируя факт, что первый токен из пары уже доступен:

```
(FP + (IP.i).значение := v1
(FP + (IP.i).наличие := 1
```

Этот случай отражен на рис. 15.8, *а*, когда на вершину SUB по левой входной дуге поступил токен <35, <FP, IP>>.

Если токен <v2,<FP, IP>> приходит в вершину, для которой уже хранится значение v1, команда, представляющая данную вершину, может быть активирована и выполнена с операндами v1 и v2. В этот момент значение v1 извлекается из памяти кадров, бит наличия сбрасывается, и на функциональный блок, предназначенный для выполнения операции, передается пакет команды <v1, v2, FP, IP, IP.opc, IP.dis>, содержащий операнды (v1 и v2), код операции (IP.opc) и адресат ее результата (IP.dis). Входящие в этот пакет значения FP и IP нужны, чтобы вместе с IP.dis вычислить исполнительный адрес адресата. После выполнения операции функциональный блок пересылает результат в блок формирования токенов. Рисунок 15.8, *б* демонстрирует ситуацию, когда токен уже пришел и на второй вход вершины SUB. Операция становится активируемой, и после ее выполнения результат передается на вершины ADD и MUL, которые ожидают входных токенов в ячейках FP+3 и FP+4 соответственно.

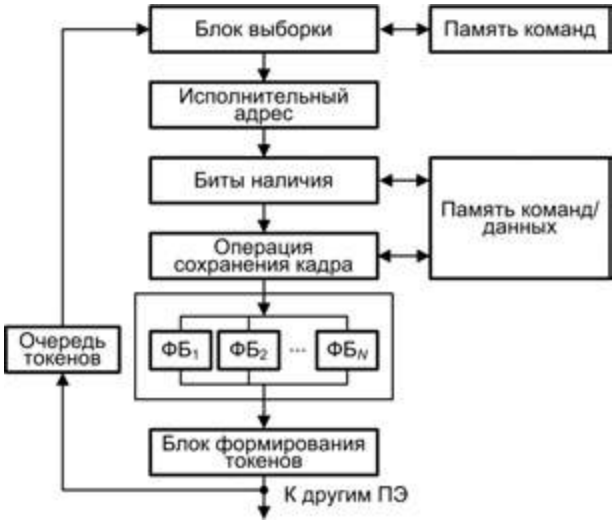


Рис. 15.9. Структура процессорного элемента типовой потоковой системы с явной адресацией токенов

Типовая структура системы с явной адресацией токенов показана на рис. 15.9. Отметим, что функция согласования токенов стала достаточно короткой операцией, что позволяет внедрить ее в виде нескольких ступеней процессорного конвейера.

Мультипотоковые вычислительные системы

Рассмотренный ранее механизм обработки с управлением от потока данных функционирует на уровне команд и его относят к *потоковой обработке низкого уровня* (fine-grain dataflow). Данному подходу сопутствуют большие издержки при пересылке операндов, но, главное, не сохраняется принцип локальности (порядок

использования команд и данных — произвольный), из-за чего теряются преимущества использования кэш-памяти. Справиться с этими проблемами помогает потоковая обработка на процедурном уровне, так называемая *укрупненная потоковая* или *мультипотоковая* обработка (multithreading). Буквальный перевод английского термина означает потоковую обработку множества нитей.

Мультипотоковая модель совмещает локальность программы, характерную для фон-неймановской модели, с толерантностью к задержкам на переключение задач, характерным для потоковой архитектуры. Данная возможность обеспечивается тем, что вершина графа представляет собой не одну команду, а последовательность из нескольких команд, называемую *нитью* (thread). Именно поэтому мультипотоковую организацию часто называют *крупнозернистой потоковой обработкой* (coarse-grained dataflow). Мультипотоковая обработка сводится к потоковому выполнению нитей, в то время как внутри отдельной нити характер выполнения фон-неймановский. Порядок обработки нитей меняется динамически в процессе вычислений, а последовательность команд в пределах нити определена при компиляции статически. Структура мультипотоковой ВС представлена на рис. 15.10.

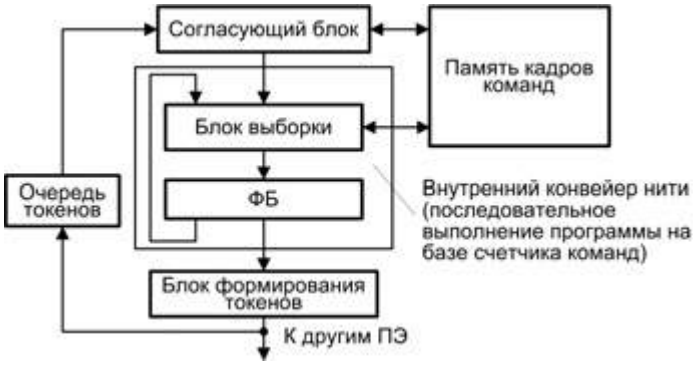


Рис. 15.10. Структура процессорного элемента типовой мультипотоковой системы

Существенное отличие мультипотоковой системы от обычной потоковой состоит в организации внутреннего управляющего конвейера, где последовательность выполнения команд задается счетчиком команд, как в фон-неймановских машинах. Иными словами, этот конвейер идентичен обычному конвейеру команд.

Вернемся к иллюстрации возможных вычислительных моделей (см. рис.15.1). В мультипотоковой архитектуре (см. рис. 15.1, *в*) каждая закрашенная область представляет вершину графа, в которой находится последовательность команд — нить. Если команда приостанавливается, останавливается и соответствующая нить, в то время как выполнение других нитей может продолжаться.

Существует две формы мультипотоковой обработки: *без блокирования* и *с блокированием*. В модели без блокирования выполнение нити не может быть начато, пока не получены все необходимые данные. Будучи запущенной, нить выполняется до конца без приостановки. В варианте с блокированием запуск нити может быть произведен до получения всех операндов. Когда требуется отсутствующий операнд,

нить приостанавливается (блокируется), а возобновление выполнения откладывается на некоторое время. Процессор запоминает всю необходимую информацию о состоянии и загружает на выполнение другую готовую нить. Модель с блокированием обеспечивает более мягкий подход к формированию нитей (часто это выражается в возможности использования более длинных нитей) за счет дополнительной аппаратуры для хранения блокированных нитей.

Возможна также и потоковая обработка переменного уровня, когда вершины соответствуют как простым операциям, так и сложным последовательным процедурам. Последний случай иногда называют *комбинированной обработкой с потоками данных и потоками управления* (combined dataflow/control flow).

Вычислительные системы волнового фронта

Интересной разновидностью систолических структур являются *матричные процессоры волнового фронта* (wavefront array processor), иногда называемые также *волновыми* или *фронтальными*.

Как уже отмечалось, в основе построения систолических ВС лежит глобальная синхронизация массива процессоров, предусматривающая наличие сети распределения синхронизирующих сигналов по всей структуре. В системах с очень большим числом ПЭ начинает сказываться запаздывание тактовых сигналов. Последнее обстоятельство особенно ощутимо при исполнении массива на базе большой интегральной микросхемы, где связи между ПЭ очень тонкие физически, вследствие чего обладают повышенной емкостью. В итоге возникают серьезные проблемы с синхронизацией, устраняют которые самосинхронизирующиеся схемы управления процессорными элементами. *Самосинхронизация* заключается в том, что моменты начала очередной операции каждый ПЭ определяет автоматически, по наличию соответствующих операндов, из чего следует, что ВС волнового фронта — это еще одна разновидность потоковых ВС, хотя и узко специализированная. В итоге отпадает необходимость глобальной синхронизации, исчезают непроизводительные временные издержки и повышается общая производительность всей структуры, хотя и усложняется аппаратная реализация каждого ПЭ [101, 105].

Волновые процессорные массивы сочетают систолическую конвейерную обработку данных с асинхронным характером потока данных. В качестве механизма координации межпроцессорного обмена в волновых системах принята асинхронная *процедура связи с подтверждением* (handshake). Когда какой-либо процессор массива завершает свои вычисления и готов передать данные соседу, он может это сделать лишь при готовности последнего к их приему. Для проверки готовности соседа передающий процессор сначала направляет ему запрос, а данные посылает только после получения подтверждения о готовности их принять. В систолических ВС фронты вычислений продвигаются по структуре синхронно и представляют собой прямые линии. В волновых системах они могут быть изогнутыми кривыми, меняющими свою конфигурацию во времени в зависимости от задержек в отдельных ПЭ и моментов поступления данных на каждый ПЭ.

Концепцию массива процессоров волнового фронта проиллюстрируем на примере матричного умножения $C = A \times B$ (рис. 15.11).

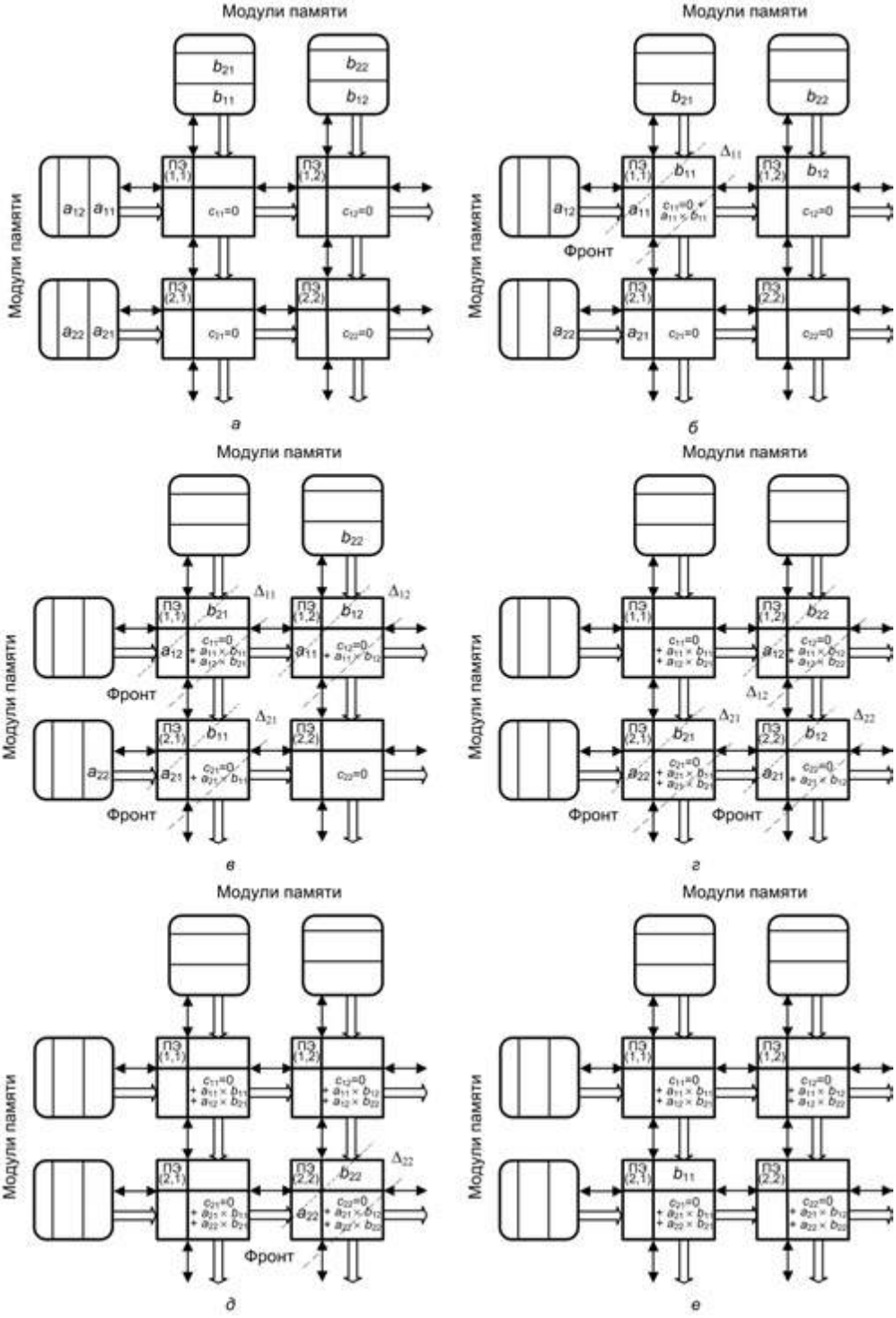


Рис. 15.11. Выполнение матричного умножения на волновой вычислительной системе

Вычислительная система здесь состоит из процессорных элементов, имеющих на каждом входе данных буфер на один операнд. Всякий раз, когда буфер пуст, а в памяти, являющейся источником данных, содержится очередной операнд, производится немедленное его считывание в буфер соответствующего процессора. Операнды из других ПЭ принимаются на основе протокола связи с подтверждением. На рис. 15.11, *а* показано исходное состояние системы. Рисунок 15.11, *б* фиксирует ситуацию после первоначального заполнения входных буферов. Здесь ПЭ(1,1) суммирует произведение $a_{11} \times b_{11}$ с содержимым своего аккумулятора и транслирует операнды a_{11} и b_{11} своим соседям. Таким образом, первый волновой фронт вычислений перемещается в направлении от ПЭ(1,1) к ПЭ(1,2) и ПЭ(2,1). Последующие рис. 15.11, *в–е* иллюстрируют продолжение распространения первого фронта и формирование последующих фронтов вычислений. Фронты на схеме показаны двумя линиями. Первая отражает задержку на получение операндов, а вторая — время вычислений в ПЭ. Общая задержка в ПЭ(i,j) обозначена Δ_{ij}.

По сравнению с систолическими ВС массивы волнового фронта обладают лучшей масштабируемостью, проще в программировании и характеризуются более высокой отказоустойчивостью.

Вычислительные системы с управлением по запросу

В системах с управлением от потока данных каждая команда, для которой имеются все необходимые операнды, немедленно выполняется. Однако для получения окончательного результата многие из этих вычислений оказываются ненужными. Отсюда прагматичным представляется иной подход, когда вычисления инициируются не по готовности данных, а на основе запроса на данные. Такая организация вычислительного процесса носит название *управления вычислениями по запросу* (demand-driven control). В ее основе, как и в потоковой модели (data-driven control), лежит представление вычислительного процесса в виде графа. В потоковой модели вершины вверху графа запускаются раньше, чем нижние. Это — нисходящая обработка. Механизм управления по запросу состоит в обработке вершин потокового графа снизу вверх (вершина запускается лишь когда требуется ее результат). Данный процесс получил название *редукции графа*, а ВС, работающая в режиме снизу вверх (см. рис. 15.1, *г*), называется *редукционной вычислительной системой*.

Математическую основу редукционных ВС составляет *лямбда-исчисление* [1, 51, 144], а для написания программ под такие системы нужны так называемые функциональные языки программирования (FP, Haskell и др.). На функциональном языке все программы представляются в виде выражений[1], а процесс выполнения программы заключается в определении значений последних (это называется *оценкой выражения*). Оценка выражения производится посредством повторения операции выбора и упрощения тех частей выражения, которые можно свести от сложного к простому (такая часть выражения называется *редексом* (от REDuced EXpression — приведенное выражение), причем сам редекс также является отдельным выражением).

[1] В этом контексте выражение представляет собой «сборку» из операций.

Операция упрощения называется *редукцией*. Процесс редукции завершается, когда преобразованное редукцией выражение больше не содержит редекса. Выражение, не содержащее редекса, называется *нормальной формой*.

В редукционной ВС вычисления производятся по запросу на результат операции. Предположим, что вычисляется выражение $a = (b+1) \times c - \frac{d}{c}$. В случае потоковых моделей процесс начинается с самых внутренних операций, а именно с параллельного вычисления $(b + 1)$ и $\frac{d}{c}$. Затем выполняется операция умножения $(b + 1) \times c$ и, наконец, самая внешняя операция — вычитание. Такой род вычислений часто называют *энергичными вычислениями* (eager evaluation).

При вычислениях, управляемых запросами, все начинается с запроса на результат a, который включает в себя запрос на вычисление выражений $(b + 1) \times c$ и $\frac{d}{c}$, а те, в свою очередь, формируют запрос на вычисление $b + 1$, то есть на операцию самого внутреннего уровня. Результат возвращается в порядке, обратном поступлению запросов. Отсюда название *ленивые вычисления* (lazy evaluation), поскольку операции выполняются только тогда, когда их результат требуется другой операции. Редукционные вычисления естественно согласуются с концепцией функционального программирования, упрощающей распараллеливание программ.

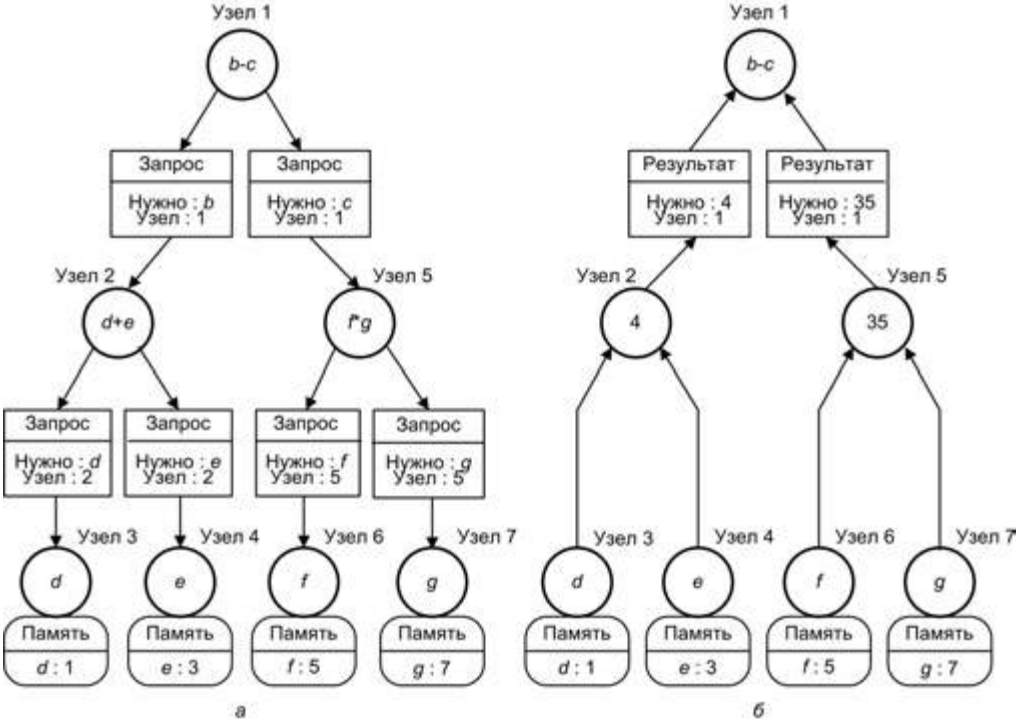


Рис. 15.12. Пример вычисления выражения на редукционной вычислительной системе: *а* — исходное положение; *б* — после первого шага редукции

Известны два типа моделей редукционных систем: строчная и графовая, отличающиеся тем, что именно передается в операцию (функцию) — скопированные значения данных или же только указатели мест хранения данных.

В *строчной редукционной модели* каждая запросившая вершина получает отдельную копию выражения для собственной оценки. Длинное строковое выражение рекурсивным образом сокращается (редуцируется) до единственного значения. Каждый шаг редукции содержит операцию, сопровождаемую ссылкой на требуемые входные операнды. Операция приостанавливается, пока оцениваются входные параметры.

На рис. 15.12 показан процесс вычисления с помощью редукционной ВС значения выражения $a = b - c$ ($b = d + e$, $c = f \times g$) для $d = 1$, $e = 3$, $f = 5$, $g = 7$[1]. Программа редукции состоит из распознавания редексов с последующей заменой их вычисленными значениями. Таким образом, вся программа в конечном итоге редуцируется до результата.

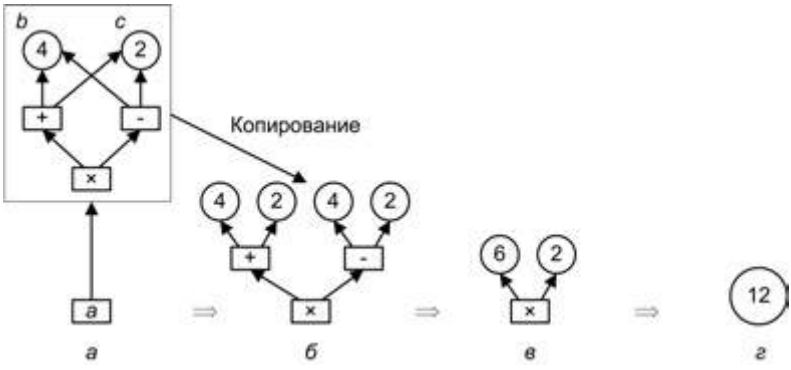


Рис. 15.13. Процесс вычислений в модели со строчной редукцией: *а* — исходный граф; *б*, *в* — последовательно редуцированные графы; *г* — результат редукции

На рис. 15.13 показан процесс вычислений с помощью строчной редукции. Если требуется значение $a = (b + c) \times (b - c)$ (рис. 15.13, *а*), то копируется граф программы, определяющий вычисление *a* (см. рис. 15.13, *б*). При этом запускается операция умножения. Поскольку это вычисление невозможно без предварительного расчета двух параметров, то запускаются вычисления сложения и вычитания, в результате чего образуется редуцированный граф (с ветвями «6» и «2»), показанный на рис. 15.13, *в*. Результат получается путем дальнейшей редукции (см. рис. 15.13, *г*).

В *графовой редукционной модели* выражение представлено как ориентированный граф. Граф сокращается по результатам оценки ветвей и подграфов. В зависимости от запросов возможно параллельное оценивание и редукция различных частей графа или подграфов. Запросившему узлу, который управляет всеми ссылками на граф, возвращается указатель на результат редукции. «Обход» графа и изменение ссылок продолжаются, пока не будет получено значение результата, копия которого возвращается запросившей команде.

[1] Здесь вершины названы узлами.

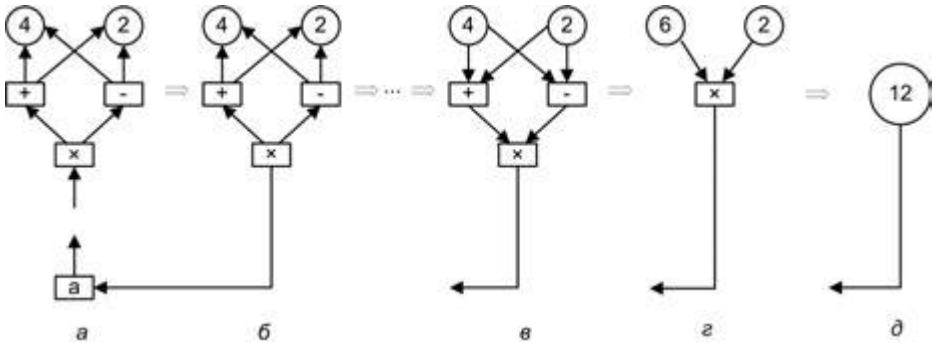


Рис. 15.14. Процесс вычислений в модели с графовой редукцией: *а* — исходный граф; *б*, *в*, *г* — последовательная редукция со сменой направления указателей; *д* — результат редукции

Рисунок 15.14 иллюстрирует пример вычисления $a = (b + c) \times (b - c)$ с помощью графовой редукции. В этой модели для нахождения значения *a* граф программы не копируется, а лишь передается указатель на этот граф (см. рис. 15.14, *а*). При достижении вершины умножения (на графе программы) направление переданного указателя меняется на противоположное, указывая место, куда будет выдаваться итоговой результат вычисления (см. рис. 15.14, *б*). Далее, путем повторения операции смены направления текущего указателя на обратное получается граф, показанный на рис. 15.14, *в*. В нем изменены направления дуг для вершин сложения и вычитания (они указывают: куда подавать операнды и куда посылать результаты сложения и вычитания). Теперь операции сложения и вычитания можно выполнять, граф редуцируется сначала до изображенного на рис. 15.14, *г*, а затем до изображенного на рис. 15.14, *д*.

Контрольные вопросы

1. Перечислите и охарактеризуйте возможные механизмы управления вычислительным процессом.
2. В чем состоит идея управления от потока данных?
3. Какие элементарные операторы могут быть взяты в качестве вершин потокового графа?
4. Каким образом осуществляется передача данных между вершинами потокового графа?
5. В чем состоит принципиальное различие между статической и динамической потоковой архитектурами?
6. Выполнение какого условия, кроме наличия входных данных, требуется для активации операции в статической потоковой ВС?
7. Опишите структуру пакетов действий и пакетов результата в статической потоковой ВС и поясните назначение полей этих пакетов.
8. Какой смысл вкладывается в понятие «окрашенный токен»?

9. Сохраняется ли в потоковых ВС принцип локальности по обращению, свойственный вычислительным системам фон-неймановского типа?
10. В чем состоят сходство и различие между систолическими ВС и вычислительными системами с обработкой по принципу волнового фронта?
11. Как организуется межпроцессорный обмен в массивах волнового фронта?
12. Определите понятие thread применительно к мультипотоковой обработке.
13. В чем заключаются преимущества мультипотоковой обработки над обычной потоковой?
14. Почему вычислительные системы с управлением по запросу называют редукционными?
15. Какой математический аппарат лежит в основе редукционных ВС? Поясните основные положения этого аппарата.
16. Поясните различия между строковой и графовой моделями редукции.

Заключение

Любую работу трудно начинать и еще труднее заканчивать, но приходится...

Наши поздравления уважаемому читателю — надеемся, что вы оказались на этой странице не в силу природного любопытства и нетерпеливости, а в результате изучения всего материала учебника.

Теперь вы вооружены и опасны ☺. Вооружены базовыми знаниями в данной предметной области, а опасны для дилетантов и «незнаек», то есть людей несведущих. И конечно, вы открыты новым знаниям, тому, что у нас впереди. Это очень важно, ведь темпы развития в этой области знаний предельно высоки. Специалист по вычислительным машинам и системам должен быть готов к обучению на протяжении всей профессиональной жизни: поезд новых компьютерных решений движется чрезвычайно стремительно, только успевай впрыгивать на его подножку!

Мы намеренно не употребляли слово «электронные» применительно к вычислительным машинам. Перефразируя известное высказывание, электроника — колыбель вычислительных машин, но нельзя же вечно жить в колыбели! Двадцать первый век принесет массу сюрпризов в области элементной базы ВМ и ВС, а вместе с ее изменениями переменится архитектура и организация вычислительных средств. Вот краткий список тех новаций, которые уже стучатся в дверь: голографическая, твердотельная и протонная память; схемы на базе молекулярных ключей; оптические, квантовые и нанокомпьютеры; электронная цифровая бумага; пластмассовые дисплеи; нейроинформатика, биоинформатика... И это еще далеко не все. Словом, дорога в компьютерный космос открыта, а информационная революция только начинается... Будьте готовы к переменам, и все у вас получится ☺.

Впереди длинный и интересный путь познаний. Удачи Вам, Уважаемый Читатель, на этом пути!

Приложение А

Арифметические основы вычислительных машин

В общем перечне проблем, связанных с разработкой и функционированием ВМ, важное место занимают способы записи и кодирования чисел. Обсуждение этих вопросов составляет содержание данного приложения.

Системы счисления

Система счисления — это совокупность символов и правил для обозначения чисел. Известные системы счисления подразделяются на непозиционные и позиционные.

В *непозиционных системах счисления* вес цифры, то есть вклад, который она вносит в значение числа, не зависит от ее позиции в записи числа.

Наиболее известной из подобных систем является римская система счисления. В ней для записи числа используются цифры, обозначаемые буквами латинского алфавита: I = 1; V = 5; X = 10; L = 50; C = 100; D = 500; M = 1000. Вес каждой цифры в любой позиции записи числа остается неизменным. Так, в числе XXXII (тридцать два) вес цифры X в любой позиции равен просто десяти. Числа формируются в соответствии со следующими правилами.

1. Запись из стоящих подряд одинаковых цифр изображает сумму чисел, обозначаемых этими цифрами. Например, число XXX (тридцать) образуется как X+X+X (10 + 10 + 10).
2. Запись, в которой меньшая по весу цифра стоит слева от цифры с большим весом, изображает разность соответствующих чисел. Так, число IV (четыре) формируется как V–I (5 – 1).
3. Запись, в которой цифра с меньшим весом стоит справа от цифры с большим весом, изображает сумму соответствующих чисел. Например, число VI (шесть) образуется как V+I (5 + 1).

В римской системе отсутствует символ нуля, а также обычные и десятичные дроби. Эти особенности плюс неудобство правил образования чисел обусловили то, что римская система, как, впрочем, и иные непозиционные системы счисления, в вычислительной технике применения не нашли.

Для вычислительной техники характерно использование *позиционных систем счисления*, где числа представляются в виде последовательности цифр, в которой значение каждой цифры зависит от ее позиции в этой последовательности. Количество s различных цифр, употребляемых в позиционной системе, называется основанием, или базой системы счисления. В общем случае положительное число X в позиционной системе с основанием s может быть представлено в виде полинома:

$$X = x_{r-1}...x_1x_0,x_{-1}...x_{-p} = x_{r-1}s^{r-1} + ... + x_0s^0 + x_{-1}s^{-1} + ... + x_{-p}s^{-p},$$

где s — база системы счисления, x_i — цифры, допустимые в данной системе счисления ($0 \leqslant x_i \leqslant s-1$). Последовательность $x_{r-1}x_{r-2}\ ...\ x_0$ образует *целую часть* X, а последовательность $x_{-1}x_{-2}\ ...\ x_{-p}$ — *дробную часть* X. Таким образом, r и p обозначают количество цифр в целой и дробной частях соответственно. Целая и дробная части разделяются запятой[1]. Цифры x_{r-1} и x_{-p} называются соответственно *старшим* и *младшим* разрядами числа.

Чтобы представить в ВМ числа, заданные в позиционной системе с основанием s, необходимо использовать физические элементы, имеющие s устойчивых состояний. Природа электронных устройств накладывает ограничения на применение в машинных вычислениях десятичной системы счисления (DEC — decimal), хотя эта система и более привычна. Электронные элементы с более чем двумя устойчивыми состояниями имеют низкие показатели по надежности, быстродействию, габаритам и стоимости, поэтому в ВМ наибольшее применение нашли двоичная (BIN — binary) и двоично-кодированные системы счисления: восьмеричная (OCT — octal), шестнадцатеричная (HEX — hexadecimal), двоично-кодированная десятичная (BCD — binary coded decimal).

В дальнейшем для обозначения используемой системы счисления число будем заключать в скобки, а в индексе указывать основание системы. Так, число X по основанию s будем обозначать $(X)_s$.

Двоичная система счисления

Основанием системы счисления служит число 2 ($s = 2$), и для записи чисел используются только две цифры: 0 и 1. Чтобы представить любой разряд двоичного числа, достаточно иметь физический элемент с двумя четко различимыми устойчивыми состояниями, одно из которых изображает 1, а другое — 0.

Пример записи числа в двоичной системе:

$$(173{,}625)_{10} = 1\times2^7 + 0\times2^6 + 1\times2^5 + 0\times2^4 + 1\times2^3 + 1\times2^2 + 0\times2^1 + 1\times2^0 + \\ + 1\times2^{-1} + 0\times2^{-2} + 1\times2^{-3} = (10101101{,}101)_2.$$

[1] В некоторых странах — точкой.

Недостатком двоичной системы можно считать ее относительную громоздкость, так как для изображения двоичного числа требуется примерно в 3,3 раза больше разрядов, чем при десятичном представлении.

Восьмеричная и шестнадцатеричная системы счисления

Эти системы счисления относятся к двоично-кодированным, в которых основание системы счисления представляет собой целую степень двойки: 2^3 — для восьмеричной и 2^4 — для шестнадцатеричной.

В восьмеричной системе ($s = 8$) используются 8 цифр — 0, 1, 2, 3, 4, 5, 6, 7.

Пример записи в 8-ричной системе:

$$(173{,}625)_{10} = 2 \times 8^2 + 5 \times 8^1 + 5 \times 8^0 + 5 \times 8^{-1} = (255{,}5)_8.$$

В шестнадцатеричной системе ($s = 16$) используется 16 цифр — 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F.

Пример записи в 16-ричной системе:

$$(173{,}625)_{10} = \mathrm{A} \times 16^1 + \mathrm{D} \times 16^0 + \mathrm{A} \times 16^{-1} = (\mathrm{AD,A})_{16}.$$

Широкое применение 8-ричной и 16-ричной систем счисления обусловлено двумя факторами.

Таблица А.1. Соответствие между цифрами в различных системах счисления

DEC	BIN	OCT	HEX	BCD
0	0000	0	0	0000
1	0001	1	1	0001
2	0010	2	2	0010
3	0011	3	3	0011
4	0100	4	4	0100
5	0101	5	5	0101
6	0110	6	6	0110
7	0111	7	7	0111
8	1000	10	8	1000
9	1001	11	9	1001
10	1010	12	A	0001 0000
11	1011	13	B	0001 0001
12	1100	14	C	0001 0010
13	1101	15	D	0001 0011
14	1110	16	E	0001 0100
15	1111	17	F	0001 0101

Во-первых, эти системы позволяют заменить запись двоичного числа более компактным представлением (запись числа в 8-ричной и 16-ричной системах будет соответственно в 3 и 4 раза короче двоичной записи этого числа). Во-вторых, взаимное преобразование чисел между 2-ичной системой с одной стороны и 8-ричной

и 16-ричной — с другой осуществляется сравнительно просто. Действительно, поскольку для 8-ричного числа каждый разряд представляется группой из трех двоичных разрядов (триад), а для 16-ричного — группой из четырех двоичных разрядов (тетрад), то для преобразования двоичного числа достаточно объединить его цифры в группы по 3 или 4 разряда соответственно, продвигаясь от разделительной запятой вправо и влево. При этом, в случае необходимости, добавляют нули слева от целой части и/или справа от дробной части, и каждую такую группу — триаду или тетраду — заменяют эквивалентной 8-ричной или 16-ричной цифрой (см. табл. А.1)[1]. Для обратного перевода каждая OCT или HEX цифра заменяется соответственно триадой или тетрадой двоичных цифр, причем незначащие нули слева и справа отбрасываются.

Для рассмотренных ранее примеров это выглядит следующим образом:

$$\begin{array}{l} 010\ \ 101\ \ 101\ \ 101 \\ (2\quad 5\quad 5,\quad 5)_8 = (10101101{,}101)_2; \\ 1010\ \ 1101\ \ 1010 \\ (\mathrm{A}\quad \mathrm{D},\quad \mathrm{A})_{16} = (10101101{,}1010)_2. \end{array}$$

Двоично-десятичная система счисления

В двоично-десятичной системе вес каждого разряда равен степени 10, как в десятичной системе, а каждая десятичная цифра кодируется четырьмя двоичными цифрами. Для записи десятичного числа в BCD-системе достаточно заменить каждую десятичную цифру эквивалентной четырехразрядной двоичной комбинацией (см. табл. А.1):

$$\begin{array}{l} 0001\ \ 0111\ \ 0011\ \ 0110\ \ 0010\ \ 0101 \\ (1\quad 7\quad 3,\quad 6\quad 2\quad 5)_{10} = (0001\ 0111\ 0011, 0110\ 0010\ 0101)_{2\text{–}10}. \end{array}$$

Любое десятичное число можно представить в двоично-десятичной записи, но следует помнить, что это не двоичный эквивалент числа. Это видно из следующего примера:

$$(173, 625)_{10} = (10101101{,}101)_2 = (0001\ 0111\ 0011, 0110\ 0010\ 0101)_{2\text{–}10}.$$

Преобразование позиционных систем счисления

Пусть X — число в системе счисления с основанием s, которое требуется представить в системе с основанием h. Удобно различать два случая.

В первом случае $s < h$ и, следовательно, при переходе к основанию h можно использовать арифметику этой системы. Метод преобразования состоит в представлении числа $(X)_s$ в виде многочлена по степеням s, а также в вычислении этого многочлена

[1] В табл. А.1 ячейки с затемненным фоном содержат не цифры, а числа в соответствующих системах счисления.

по правилам арифметики системы счисления с основанием h. Так, например, удобно переходить от двоичной или восьмеричной системы к десятичной. Описанный прием иллюстрируют следующие примеры:

$$(1101{,}01)_2 = 1\times 2^3 + 1\times 2^2 + 0\times 2^1 + 1\times 2^0 + 0\times 2^{-1} + 1\times 2^{-2} = (13{,}25)_{10},$$
$$(432{,}2)_8 = 4\times 8^2 + 3\times 8^1 + 2\times 8^0 + 2\times 8^{-1} = (282{,}25)_{10}.$$

В обоих случаях арифметические действия выполняются по правилам системы с основанием 10.

Во втором случае ($s > h$) удобнее пользоваться арифметикой по основанию s. Здесь следует учитывать, что перевод целых чисел и правильных дробей производится по различным правилам. При переводе смешанных дробей целая и дробная части переводятся каждая по своим правилам, после чего полученные числа записываются через запятую.

Перевод целых чисел

Правило перевода целых чисел становится ясным из общей формулы записи числа в произвольной позиционной системе. Пусть число $(X)_s$ в исходной системе счисления s имеет вид $(x_{r-1}\ldots x_1x_0)_s$. Требуется получить запись числа в системе счисления с основанием h:

$$(X)_h = (y_{w-1}\ldots y_1y_0)_h = y_{w-1}h^{w-1} + y_{w-2}h^{w-2} + \ldots + y_1h^1 + y_0h^0.$$

Для нахождения значений y_i разделим этот многочлен на h:

$$\frac{X}{h} = y_{w-1}h^{w-2} + y_{w-2}h^{w-3} + \ldots + y_1 + \frac{y_0}{h}.$$

Как видно, младший разряд $(X)_h$, то есть y_0, равен первому остатку. Следующий значащий разряд y_1 определяется делением частного $Q_0 = y_{w-1}h^{w-2} + y_{w-2}h^{w-3} + \ldots + y_1$ на h:

$$\frac{Q_0}{h} = y_{w-1}h^{w-3} + y_{w-2}h^{w-4} + \ldots + y_2 + \frac{y_1}{h}.$$

Остальные y_i также вычисляются путем деления полученных частных до тех пор, пока Q_{w-1} не станет равным нулю.

Для перевода целого числа из s-ричной системы в h-ричную необходимо последовательно делить это число и получаемые частные на h (по правилам системы с основанием s) до тех пор, пока частное не станет равным нулю. Старшей цифрой в записи числа в системе с основанием h служит последний остаток, а следующие за ней цифры образуют остатки от предшествующих делений, выписываемые в последовательности, обратной их получению.

В качестве примера рассмотрим последовательность перевода числа 75 из десятичной системы в 2-ичную, 8-ричную и 16-ричную системы счисления (рис. А.1).

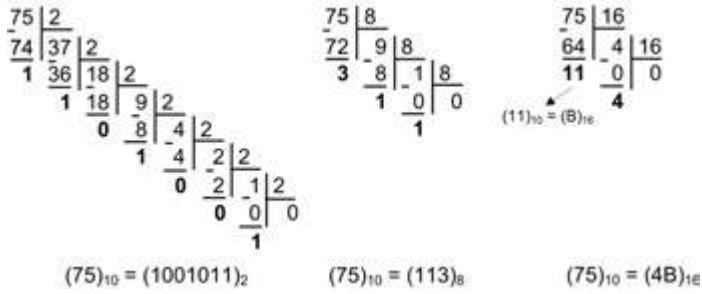


Рис. А.1. Пример перевода целого десятичного числа 75 в 2-ичную, 8-ричную и 16-ричную системы счисления

Перевод правильных дробей

Правильную дробь $(X)_s$, имеющую в системе с основанием s вид $(x_{-1}x_{-2}...x_{-p})_s$, можно выразить в системе с основанием h как многочлен вида:

$$(X)_h = (y_{-1}y_{-2}...y_{-p})_h = y_{-1}h^{-1} + y_{-2}h^{-2} + ... + y_{-p}h^{-p}.$$

Старшая цифра y_{-1} может быть найдена умножением этого многочлена на h, то есть

$$X \times h = y_{-1} + y_{-2}h^{-1} + ... + y_{-p}h^{-p+1}.$$

Если это произведение меньше 1, то цифра y_{-1} равна 0, если же оно больше или равно 1, то цифра y_{-1} равна целой части произведения. Следующая цифра справа y_{-2} определяется путем умножения дробной части указанного выше произведения на h и выделения его целой части и т. д. Процесс может оказаться бесконечным, так как не всегда можно представить дробь по основанию h конечным набором цифр.

Для перевода правильной дроби из системы счисления с основанием s в систему счисления с основанием h нужно умножать исходную дробь и дробные части получающихся произведений на основание h (по правилам «старой» s-системы). Целые части полученных произведений дают последовательность цифр дроби в h-системе.

Описанная процедура продолжается до тех пор, пока дробная часть очередного произведения не станет равной нулю либо не будет достигнута требуемая точность изображения числа X в h-ричной системе. Представлением дробной части числа X в новой системе счисления будет последовательность целых частей полученных произведений, записанных в порядке их получения и изображенных h-ричной цифрой. Абсолютная погрешность перевода числа X при p знаках после запятой равняется $\frac{h^{-(p+1)}}{2}$.

На рис. А.2 приведены примеры перевода правильной дроби 0,453 из десятичной системы в 2-ичную (рис. А.2, *а*), 8-ричную (рис. А.2, *б*) и 16-ричную (рис. А.2, *в*) системы счисления.

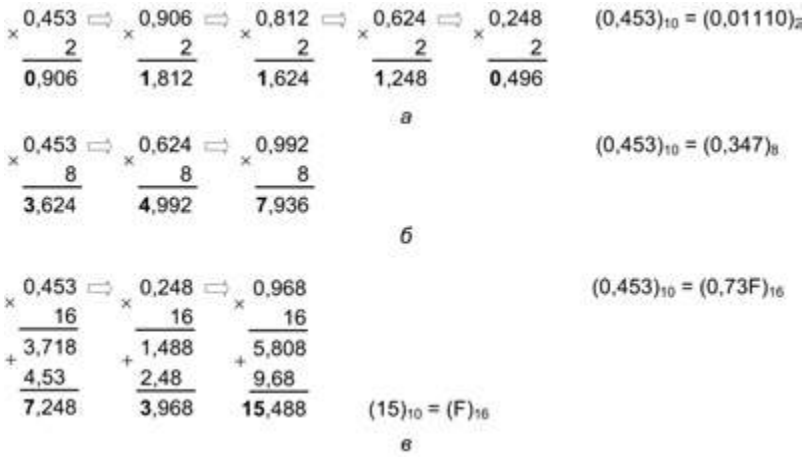


Рис. А.2. Пример перевода десятичной дроби 0,453 в 2-ичную, 8-ричную и 16-ричную системы счисления

Перевод правильных дробей, у которых знаменатель кратен степени нового основания системы счисления

Если преобразуемая дробь имеет вид $(X)_s = \frac{z}{h^m}$, *где m — целое число, то числитель дроби z переводится как целое число, после чего запятая в записи числа перемещается влево на m разрядов.*

Пример перевода из десятичной системы в 2-ичную:

$$X = \frac{12}{256} = \frac{12}{2^8}; (12)_{10} = (1100)_2; X = (0{,}00001100)_2.$$

В отличие от ранее рассмотренного варианта подобный перевод всегда является конечным и точным.

Кодирование отрицательных чисел в ВМ

До сих пор мы рассматривали только положительные числа. При записи двоичных чисел со знаком в их формате необходимо предусмотреть два поля: поле, определяющее знак числа, и поле, характеризующее модуль числа. Под знак числа отводится специальный знаковый бит (двоичный разряд). Остальные разряды определяют модуль числа. Знаковый бит приписывается слева от модуля числа, причем знаку «+» соответствует нулевое значение знакового бита, а «–» – единичное.

В истории развития архитектуры ВМ использовались три основных варианта представления знаковых чисел:

- прямой код или знак и величина;
- обратный код или код с дополнением до единицы;
- дополнительный код или код с дополнением до двух.

Во всех трех кодах (прямом, обратном и дополнительном) положительные числа выглядят одинаково. Для отрицательных чисел различия в форме записи числа касаются только способа представления модуля числа, в то время как способ кодирования и место расположения знакового бита остаются неизменными.

Прямой код двоичного числа

В системе представления в прямом коде число состоит из кода знака и модуля числа, причем обе эти части обрабатываются по отдельности. Формула для образования прямого кода правильной дроби X имеет вид[1]:

$$[X]_{\text{п}} = X, \quad \text{если } X \geqslant 0,$$

$$[X]_{\text{п}} = 1 - X, \quad \text{если } X < 0.$$

Примеры прямого кода для правильных дробей:

$$x_1 = +0{,}0101\ [x_1]_{\text{п}} = 0{,}0101;\ x_2 = -0{,}1011\ [x_2]_{\text{п}} = 1{,}1011.$$

Прямой код целого числа получается по формуле:

$$[X]_{\text{п}} = X, \text{ если } X \geqslant 0,$$

$$[X]_{\text{п}} = 1 \times 2^{n-1} - X, \text{ если } X < 0.$$

Примеры прямого кода для целых чисел:

$$x_1 = +1101\ [x_1]_{\text{п}} = 0{,}1101;\ x_2 = -1011\ [x_2]_{\text{п}} = 1{,}1011.$$

При всей своей наглядности представление чисел в прямом коде имеет существенный недостаток — формальное суммирование чисел с различающимися знаками дает неверный результат. Проиллюстрируем это на примере сложения двух чисел: $x_1 = +5_{10}\ (+101)_2$ и $x_2 = -7\ (-111)_2$. В прямом коде эти числа имеют вид: $[x_1]_{\text{п}} = 0{,}101$ и $[x_2]_{\text{п}} = 1{,}111$. Очевидно, что результат должен быть равен -2, что в прямом коде может быть записано как 1,010. В то же время при непосредственном сложении получаем

$$0{,}101 + 1{,}111 = 10{,}100,$$

то есть значение, существенно отличающееся от ожидаемого.

Для корректного выполнения процедуры суммирования чисел с разными знаками, представленными в прямом коде, необходимо определить больший по модулю операнд, вычесть из него абсолютное значение меньшего операнда и присвоить результату знак большего по модулю операнда.

Отметим также второй недостаток прямого кода — нуль имеет два различных представления, а именно $[+0]_{\text{п}} = 0{,}00..0$ и $[-0]_{\text{п}} = 1{,}00..0$, что математически не имеет смысла.

В силу отмеченных недостатков прямого кода обрабатываемая информация чаще представляется в форме дополнения.

---

[1] В дальнейшем при рассмотрении кодов условимся отделять знаковый разряд от числа точкой (в целых числах) и запятой (в правильных дробях). Кроме того, говоря о n-разрядных числах, будем помнить, что n — количество разрядов, учитывающее также разряд знака.

Представление чисел в форме дополнения

Негативный итог в случае прямого кода является следствием неудачного выбора способа кодирования. Его нужно выбирать исходя из логики использования кода, а не из «напрашивающегося» или «очевидного» на первый взгляд подхода. Основное требование при выборе системы кодирования чисел состоит в том, что полученный код должен удовлетворять правилам выполнения сложения и вычитания в двоичной системе счисления. В связи с этим, код каждого следующего положительного числа должен получаться прибавлением единицы к коду текущего числа, а код каждого следующего отрицательного числа должен получаться вычитанием единицы из кода текущего числа.

При представлении чисел в прямом коде для изменения знака достаточно поменять только значение знакового разряда. В системе *представления чисел в форме дополнения* местоположение знакового разряда и способ кодирования знака остаются теми же, что и при прямом кодировании. В то же время знаковый разряд уже не рассматривается как обособленный, а считается неотъемлемой частью числа и обрабатывается аналогично значащим разрядам и совместно с ними.

Введем понятия поразрядного и точного дополнения цифры. *Поразрядное дополнение цифры x* по основанию *s* определяется выражением

$$x' = (s-1) - x\,.$$

Поразрядное дополнение x' равно разности между наибольшей цифрой в системе счисления *s* и цифрой *x*. В двоичной системе счисления 0' = 1 и 1' = 0, так как *s* = 2, а наибольшая цифра — 1. В десятичной системе счисления наибольшей является цифра 9. Поэтому неполным дополнением цифры 3 будет 6 (9–3).

Точное дополнение цифры x по основанию *s* на единицу больше поразрядного и может быть описано как

$$x'' = s - x\,.$$

Точное дополнение цифры x'' равно разности между числом, равным основанию системы счисления *s*, и цифрой *x*. Так, в десятичной системе счисления точным дополнением цифры 3 будет цифра 7 (10–3).

Взятие дополнения — более сложная процедура, нежели изменение знака, однако два числа, представленные в форме дополнения, можно суммировать и вычитать непосредственно, не проверяя их знаки и величины, как это требуется в прямом коде. В вычислительной технике наибольшее распространение получили две системы представления чисел в форме дополнения, в основе которых лежат соответственно поразрядное дополнение и точное дополнение. Первая из этих систем более известна как обратный код, или дополнение до 1 (1's complement), а вторая — как дополнительный код, или дополнение до 2 (2's complement).

Обратный код двоичного числа

Формула образования обратного кода двоичной дроби *X* имеет вид:

$$[X]_o = 2 - 2^{-(n-1)} + X.$$

Примеры обратного кода правильных дробей:

$$x_1 = +0{,}0101\ [x_1]_о = 0{,}0101;\ x_2 = -0{,}1011\ [x_2]_о = 1{,}0100.$$

Обратный код[1] целого числа формируется в соответствии с выражением:

$$[X]_о = 2^{n-1} + X.$$

Примеры обратного кода целых чисел:

$$x_1 = +1101\ [x_1]_о = 0{,}1101;\ x_2 = -1011\ [x_2]_о = 1{,}0100.$$

Для отрицательных двоичных чисел процедуру образования обратного кода можно свести к следующему формальному правилу: в знаковый разряд записывается единица, а в цифровых разрядах прямого кода единицы заменяются нулями, а нули — единицами.

Напомним, что кодирование распространяется только на отрицательные числа, положительные числа в прямом и обратном коде выглядят идентично[2]:

Диапазон целых чисел, которые могут быть отражены обратным кодом — от $-(2^{n-1} - 1)$ до $+ (2^{n-1} - 1)$.

Хотя обратный код и позволяет решить проблему сложения и вычитания чисел с различными знаками, он имеет и определенные недостатки. Так, процесс суммирования чисел является двухэтапным, что увеличивает время выполнения данной операции. Кроме того, как и в прямом коде, имеют место два представления нуля: $[+0]_о = 0{,}00..0$ и $[-0]_о = 1{,}11..1$, поэтому схема обнаружения нуля в вычислительном устройстве должна либо проверять обе возможности, либо преобразовывать 1,11..1 в 0,00..0.

Дополнительный код двоичного числа

Дополнительный код двоичной дроби X образуется по формуле:

$$[X]_д = 2 + X.$$

Примеры дополнительного кода для правильных дробей:

$$x_1 = 0{,}0101\ [x_1]_д = 0{,}0101;\ x_2 = -0{,}1011\ [x_2]_д = 1{,}0101.$$

Формула для образования дополнительного кода целого числа X:

$$[X]_д = 2^n + X.$$

Примеры дополнительного кода для целых чисел:

$$x_1 = 1101\ [x_1]_д = 0{,}1101;\ x_2 = -1011\ [x_2]_д = 1{,}0101.$$

Дополнительный код отрицательного двоичного числа формируется по следующему правилу: в знаковый разряд записать единицу, в цифровых разрядах прямого кода

---

[1] При рассмотрении формул для обратного и дополнительного кодов следует помнить, что речь идет об отрицательных двоичных числах, то есть X в формулах — это отрицательное двоичное число.

[2] В дальнейшем при рассмотрении кодов условимся отделять знаковый разряд от числа точкой (в целых числах) и запятой (в правильных дробях). Кроме того, говоря о n-разрядных числах, будем помнить, что n — количество разрядов, учитывающее также разряд знака.

единицы заменить нулями, а нули — единицами, после чего к младшему разряду прибавить единицу.

Для примера рассмотрим число X, которое в прямом коде имеет вид: $[X]_п$ = = 1,01101010. Тогда обратный код можно записать как $[X]_о$ = 1,10010101. Для получения дополнительного кода прибавим 1 к младшему разряду обратного кода: $[X]_д$ = 1,10010101 + 0,00000001 = 1,10010110.

Дополнительный код отрицательного числа легко получить непосредственно из прямого кода, воспользовавшись следующим формальным приемом.

Найти в прямом коде числа самую правую единицу (знаковый разряд при этом не учитывается). Эту единицу, а также цифры, расположенные справа от нее и знак числа, оставить неизменными. Во всех остальных позициях поменять единицы на нули, а нули на единицы.

Проиллюстрируем описанный прием примером того же числа X: $[X]_п$ = 1,01101010. Отметим вертикальными линиями область между знаком числа, и самой правой единицей: 1,|011010|10. Внутри этой области поменяем единицы на нули, и наоборот, а вне области все оставим без изменений. Таким образом, получаем дополнительный код числа: 1,|100101|10. Убрав наши условные вертикальные линии, имеем $[X]_д$ = 1,10010110 так же, как и при стандартном способе перевода.

При представлении двоичных чисел дополнительным кодом имеется только одна форма записи нуля 0,0...00, причем ноль считается положительным числом, так как его знаковый бит равен 0. Поскольку возможно только одно двоичное представление нуля, в нижней части диапазона представляемых чисел имеется еще одно отрицательное число -2^{n-1}, у которого нет положительного эквивалента. Таким образом, диапазон целых чисел, которые могут быть представлены дополнительным кодом, составляет от $-(2^{n-1})$ до $+(2^{n-1}-1)$. Положительные числа в дополнительном коде записываются так же, как и в прямом.

Представленное в дополнительном коде n-разрядное двоичное число X можно преобразовать в m-разрядное, учитывая при этом следующие особенности.

Если $m > n$, то необходимо добавить к числу X слева $m - n$ битов, являющихся копиями знакового бита числа X. Иными словами, положительное число дополняется нулями, а отрицательное — единицами. Такое действие называется знаковым расширением. Ниже приводятся примеры знакового расширения при $n = 5$ и $m = 8$.

$$1{,}0101 \Rightarrow 1{,}1110101;\ 0{,}0101 \Rightarrow 0{,}0000101.$$

Если $m < n$, то нужно отбросить $m - n$ битов слева, однако результат будет правильным только в том случае, когда все отбрасываемые биты имеют то же значение, что и знаковый бит остающегося числа.

В большинстве ВМ используется представление чисел в дополнительном коде.

Сложение и вычитание чисел в обратном и дополнительном кодах

Преимущество дополнительного и обратного кодов состоит в том, что алгебраическое сложение этих кодов сводится к арифметическому. Это, в частности, позволяет

заменить операцию вычитания двоичных чисел $x_1 - x_2$ сложением с дополнениями $[x_1]_д + [-x_2]_д$ или $[x_1]_о + [-x_2]_о$.

При выполнении алгебраического сложения знаковый разряд и цифры модуля рассматриваются как единое целое и обрабатываются совместно. Особенность состоит в том, что перенос из старшего (знакового) разряда в обратном и дополнительном кодах учитывается по-разному. В случае обратного кода единица переноса из знакового разряда прибавляется к младшему разряду суммы (осуществляется так называемый циклический перенос). При использовании дополнительного кода единица переноса из знакового разряда отбрасывается.

Пример сложения чисел $x_1 = 0{,}10010001$ и $x_2 = -0{,}01100110$ в обратном коде приведен на рис. А.3, *а*, а в дополнительном коде — на рис А.3, *б*.

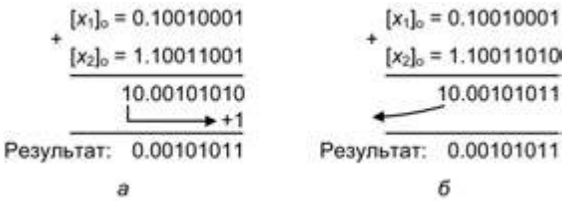


Рис. А.3. Пример сложения чисел: *а* — в обратном коде; *б* — в дополнительном коде

Если знаковый разряд результата равен нулю, это означает, что получено положительное число, которое выглядит так же, как в прямом коде. Единица в знаковом разряде означает, что результат отрицательный и его запись соответствует представлению в том коде, в котором производилась операция.

Чтобы избежать ошибок при выполнении сложения (вычитания), перед переводом чисел в обратные и дополнительные коды необходимо выровнять количество разрядов прямого кода операндов.

При сложении чисел, имеющих одинаковые знаки, могут быть получены числа, представление которых требует еще одного дополнительного разряда. Эта ситуация называется переполнением разрядной сетки. Для обнаружения переполнения в ВМ часто применяются *модифицированные прямой, обратный и дополнительный коды*. В этих кодах знак дублируется, причем знаку «плюс» соответствует комбинация 00, а знаку «минус» — комбинация 11.

Правила сложения для модифицированных кодов те же, что и для обычных. Единица переноса из старшего знакового разряда в модифицированном дополнительном коде отбрасывается, а в модифицированном обратном коде добавляется к младшему цифровому разряду.

Признаком переполнения служит появление в знаковых разрядах результата комбинации 01 при сложении положительных чисел (положительное переполнение) или 10 при сложении отрицательных чисел (отрицательное переполнение). Старший знаковый разряд в этих случаях содержит истинное значение знака суммы, а младший является старшей значащей цифрой числа.

Приложение Б

Логические основы вычислительных машин

Теоретическим фундаментом цифровой техники является *алгебра логики*, или *булева алгебра*, называемая так по имени ее основоположника Джорджа Буля, английского математика и логика середины XIX века.

В алгебре логики переменная может принимать одно из двух значений: *True* (истинно) и *False* (ложно). Эти значения в цифровой технике принято рассматривать как логическую «1» (True) и логический «0» (False), или как двоичные числа 1 и 0, и физически представлять присутствием или отсутствием некоторого сигнала.

Логические функции

Функция $f(x_1, x_2, ..., x_n)$, зависящая от n переменных, называется *функцией алгебры логики* (ФАЛ), если сама функция и любой из ее аргументов могут принимать значения только из множества {0, 1}. Другие названия ФАЛ: *логическая*, *булева* или *переключательная* функция. Совокупность значений аргументов функции алгебры логики называется *набором*, или точкой, и обозначается $< x_1, x_2, ..., x_n >$. Число возможных наборов из n аргументов равно 2^n. Аргументы булевой функции иногда называют булевыми.

Для описания логических функций используют один из нижеперечисленных способов.

Словесный способ. Здесь все случаи, при которых функция принимает значения 0 или 1, описываются словесно. Так, функцию «ИЛИ» со многими аргументами можно описать следующим образом: *функция принимает значение* 1, *если хотя бы один из аргументов принимает значение* 1, *иначе значение функции равно* 0.

Табличный способ. Булева функция $f(x_1,...,x_n)$ задается в виде *таблицы истинности* (соответствия). В левой части таблицы истинности записываются все возможные n-разрядные двоичные комбинации аргументов (табл. Б.1, *а*), а в

правой — значения функции на этих наборах. Таблица истинности содержит 2^n строк (по числу наборов аргументов), n столбцов по числу аргументов и один столбец значений функции.

Иногда вместо двоичных наборов аргументов в таблице истинности указывают их десятичные эквиваленты (табл. Б.1, *б*).

Таблица Б.1. Варианты представления таблицы истинности

x_1	x_2	x_3	f		Номер набора	f
0	0	0	0		0	0
0	0	1	1		1	1
0	1	0	0		2	0
0	1	1	0		3	0
1	0	0	1		4	1
1	0	1	1		5	1
1	1	0	0		6	0
1	1	1	1		7	1

а *б*

Числовой способ. Функция задается в виде последовательности десятичных эквивалентов тех наборов аргументов, на которых функция принимает значение 1. Например, двоичные наборы 010 и 101 имеют десятичные номера 2 и 5 соответственно. При таком подходе логическая функция трех аргументов, представленная в табл. Б.1, может быть записана в виде $f(1,4,5,7) = 1$. Та же булева функция может быть задана и по нулевым значениям: $f(0,2,3,6) = 0$.

Аналитический способ. Предполагает описание ФАЛ в виде алгебраического выражения, получаемого путем применения к аргументам операций булевой алгебры. Способ получения такого описания булевой функции будет рассмотрен в последующих разделах.

x_1	x_2	x_3	f
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

x_3 \ x_1x_2	00	01	11	10
0	0	0	0	1
1	1	0	1	1

Рис. Б.1. Таблица истинности и эквивалентная ей карта Карно

Координатный способ. При этом способе задания таблица истинности заменяется координатной картой состояний, известной под названием карты Карно. Такая карта содержит 2^n клеток по числу возможных наборов из n переменных.

Аргументы функции разбиваются на две группы так, что одна группа определяет координаты столбца, а другая — координаты строки, то есть каждой строке таблицы истинности (каждому набору аргументов) соответствует уникальная клетка карты. Внутри клетки записывается значение функции на данном наборе (рис. Б.1).

Графический, или **геометрический способ.** Булева функция $f(x_1,...,x_n)$ задается с помощью n-мерного куба, поскольку в геометрическом смысле каждый двоичный набор $x_1, x_2,..., x_n$ есть n-мерный вектор, определяющий точку n-мерного пространства. По этой причине любой набор переменных в графическом представлении булевых функций принято называть вектором. Переменные куба называют координатами. Количество переменных в кубе определяет его мерность (3-мерный, ..., n-мерный).

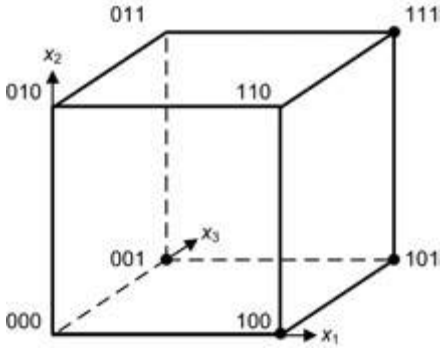


Рис. Б.2. Геометрическое представление булевой функции

Множество наборов, на которых определена функция n переменных, представляется вершинами n-мерного куба. Отметив точками те вершины куба, в которых функция принимает единичное (либо нулевое) значение, получаем геометрическое представление функции. Так, булева функция, приведенная в табл. Б.1, геометрически может быть представлена 3-мерным кубом (рис. Б.2).

Помимо перечисленных форм, иногда используются менее распространенные способы описания ФАЛ: комбинационной схемой, составленной из логических элементов; переключательной схемой; диаграммой Венна; диаграммой двоичного решения и т. д.

Булеву функцию, определенную на всех возможных наборах переменных, называют *полностью определенной*. Пример такой функции был показан в табл. Б.1. Булеву функцию n переменных называют *частично определенной*, или просто *частичной*, если она определена не на всех двоичных наборах длины n.

Наборы, на которых значение ФАЛ не определено, в таблице истинности обычно помечают каким-либо символом, например звездочкой (табл. Б.2).

Таблица Б.2. Таблица истинности для частично определенной булевой функции

x_1	x_2	x_3	f
0	0	0	0
0	0	1	1
0	1	0	*
0	1	1	0
1	0	0	1
1	0	1	*
1	1	0	0
1	1	1	*

Элементарные функции алгебры логики

В случае n переменных возможно не более 2^{2^n} различных булевых функций.

Элементарные функции одной переменной

В случае единственной переменной возможны 4 функции $f_{10i}(x)$, показанные в табл. Б.3 и поясненные в табл. Б.4. При аналитическом описании ФАЛ наибольший интерес представляет функция логического отрицания f_{102}, согласно которой результат равен *True*, если значение аргумента было *False*, и наоборот.

Таблица Б.3. Логические функции одной переменной

Переменная	Логические функции			
x	f_{100}	f_{101}	f_{102}	f_{103}
0	0	0	1	1
1	0	1	0	1

Таблица Б.4. Описание логических функций одной переменной

Функция	Название	Обозначение
f_{100}	Константа нуля	$f_{100}(x) = 0$
f_{101}	Повторение x	$f_{101}(x) = x$
f_{102}	Логическое отрицание, инверсия (от лат. inversio – переворачиваю), «НЕ»	$f_{102}(x) = \overline{x};$ $f_{102}(x) = \neg x$
f_{103}	Константа единицы	$f_{103}(x) = 1$

Элементарные функции двух переменных

Для двух переменных можно сформировать 16 (и только 16) логических функций (табл. Б.5, Б.6).

Таблица Б.5. Логические функции двух переменных

Аргументы		Логические функции															
x_1	x_2	f_{200}	f_{201}	f_{202}	f_{203}	f_{204}	f_{205}	f_{206}	f_{207}	f_{208}	f_{209}	f_{210}	f_{211}	f_{212}	f_{213}	f_{214}	f_{215}
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

Функции двух переменных, рассмотренные в табл. Б.6, играют важную роль в алгебре логики и могут быть названы элементарными.

Таблица Б.6. Описание логических функций двух переменных

Функция	Название	Обозначение
f_{200}	Константа 0	$f_{200}(x_1, x_2) = 0$ или *False*
f_{201}	Логическое умножение, конъюнкция, «И»	$f_{201}(x_1, x_2) = x_1 \wedge x_2$ или $x_1 x_2$ или $x_1 \,\&\, x_2$ или x_1 *and* x_2
f_{202}	Запрет по x_2	$f_{202}(x_1, x_2) = x_1 \Delta x_2$ или $x_1 \wedge \overline{x_2}$
f_{203}	Переменная x_1[1]	$f_{203}(x_1, x_2) = x_1$
f_{204}	Запрет по x_1	$f_{204}(x_1, x_2) = x_2 \Delta x_1$ или $\overline{x_1} \wedge x_2$
f_{205}	Переменная x_2	$f_{205}(x_1, x_2) = x_2$
f_{206}	Сложение по модулю 2, отрицание эквивалентности, «Исключающее ИЛИ»	$f_{206}(x_1, x_2) = x_1 \oplus x_2$ или x_1 *xor* x_2 или $(\overline{x_1} \wedge x_2) \vee (x_1 \wedge \overline{x_2})$
f_{207}	Логическое сложение, дизъюнкция, «ИЛИ»	$f_{207}(x_1, x_2) = x_1 \vee x_2$ или $x_1 @ x_2$ или $x_1 + x_2$ или x_1 *or* x_2
f_{208}	Функция Пирса (Вебба), «ИЛИ-НЕ»	$f_{208}(x_1, x_2) = x_1 \downarrow x_2$ или $\overline{x_1 \vee x_2}$ или $\overline{x_1} \wedge \overline{x_2}$
f_{209}	Логическая равнозначность, эквиваленция	$f_{209}(x_1, x_2) = x_1 \sim x_2$ или $x_1 \leftrightarrow x_2$ или $(\overline{x_1} \wedge \overline{x_2}) \vee (x_1 \wedge x_2)$ или $\overline{x_1 \oplus x_2}$
f_{210}	Отрицание x_1	$f_{210}(x_1, x_2) = \overline{x_2}$

продолжение ⇗

[1] Если значение переменной в функции не влияет на значение функции, как в случае переменной x_2 в функции f_{203}, то такая переменная называется фиктивной (несущественной).

Таблица Б.6 (*продолжение*)

Функция	Название	Обозначение
f_{211}	Правая импликация	$f_{211}(x_1, x_2) = x_1 \vee \overline{x_2}$ 8; 8 $x_2 \rightarrow x_1$
f_{212}	Отрицание x_2	$f_{212}(x_1, x_2) = \overline{x_1}$
f_{213}	Левая импликация	$f_{213}(x_1, x_2) = \overline{x_1} \vee x_2$ 8; 8 $x_1 \rightarrow x_2$
f_{214}	Функция Шеффера, «И-НЕ»	$f_{214}(x_1, x_2) = x_1 \mid x_2$ 8; 8 $\overline{x_1 \wedge x_2}$ 8; 8 $\overline{x_1} \vee \overline{x_2}$
f_{215}	Константа 1	$f_{215}(x_1, x_2) = 1$ 8; 8 *True*

Рассмотренные элементарные функции позволяют строить более сложные булевы функции с помощью операции суперпозиции, заключающейся в подстановке в функцию новых функций вместо аргументов. Так, путем замены в функции $f_1(x_1, x_2)$ аргументов x_1 и x_2 на функции $f_2 = x_6$ и $f_3(x_5, x_7)$ получаем функцию $f_1(x_6, f_3(x_5, x_7))$. Суперпозиция функций двух аргументов дает возможность строить функции любого числа аргументов.

Суперпозиция булевых функций представляется в виде логических формул, причем следует учитывать, что одна и та же функция может быть представлена разными формулами, среди которых имеется и наиболее простая. Нахождение такой формулы составляет предмет минимизации ФАЛ, рассматриваемой ниже.

Правила алгебры логики

Как и любая другая математическая дисциплина, булева алгебра базируется на определенном своде правил, представленных в виде аксиом, теорем и законов (тождеств). Правила эти определяются для двух возможных логических значений «1» (True) и «0» (False), а также трех базовых логических операций: «НЕ», «И» и «ИЛИ», в силу чего основные правила алгебры логики формулируются, главным образом, для этих операций и их сочетаний. Базовые логические операции перечислены в порядке понижения их приоритетности. Учет приоритетности операций позволяет сократить число скобок при записи логических выражений.

Аксиомы алгебры логики

Как известно, аксиомами в математике принято называть предположения, не требующие доказательств.

1. **Аксиомы конъюнкции**

$$0 \wedge 0 = 0;\ 0 \wedge 1 = 0;\ 1 \wedge 0 = 0;\ 1 \wedge 1 = 1.$$

2. **Аксиомы дизъюнкции**

$$0 \vee 0 = 0;\ 0 \vee 1 = 1;\ 1 \vee 0 = 1;\ 1 \vee 1 = 1.$$

3. **Аксиомы отрицания**

$$\text{если } x = 0, \text{ то } \bar{x} = 1; \text{ если } x = 1, \text{ то } \bar{x} = 0.$$

Теоремы алгебры логики

Для доказательства теорем булевой алгебры используется простая подстановка значений булевых переменных. Это обусловлено тем, что переменные могут принимать только два значения — «0» и «1».

4. **Теоремы исключения констант**

$$x \vee 1 = 1; x \vee 0 = x; x \wedge 1 = x; x \wedge 0 = 0.$$

5. **Теоремы идемпотентности (тавтологии, повторения)**

$$x \vee x = x; x \wedge x = x.$$

Для n переменных:

$$x \vee x \vee ... \vee x = x; x \wedge x \wedge ... \wedge x = x.$$

Повторное применение не дает ничего нового.

6. **Теорема противоречия**

$$x \wedge \bar{x} = 0.$$

Невозможно, чтобы противоречащие высказывания были одновременно истинными.

7. **Теорема «исключенного третьего»**

$$x \vee \bar{x} = 1.$$

Из двух противоречивых высказываний об одном и том же предмете одно всегда истинно, а второе — ложно, третьего не дано.

8. **Теорема двойного отрицания (инволюции)**

$$\bar{\bar{x}} = x.$$

Двойное отрицание отменяет инверсию.

Законы алгебры логики

Излагаемые ниже правила обычно называют законами, или тождествами булевой алгебры.

9. **Ассоциативный (сочетательный) закон**

$$x_1 \vee (x_2 \vee x_3) = (x_1 \vee x_2) \vee x_3; x_1 \wedge (x_2 \wedge x_3) = (x_1 \wedge x_2) \wedge x_3.$$

При одинаковых знаках скобки можно ставить произвольно или вообще опускать.

10. **Коммутативный (переместительный) закон**

$$x_1 \vee x_2 = x_2 \vee x_1; x_1 \wedge x_2 = x_2 \wedge x_1.$$

Результат операции над высказываниями не зависит от того, в каком порядке берутся эти высказывания.

11. **Дистрибутивный (распределительный) закон**

$$(x_1 \vee x_2) \wedge x_3 = (x_1 \wedge x_3) \vee (x_2 \wedge x_3);$$
$$(x_1 \wedge x_2) \vee x_3 = (x_1 \vee x_3) \wedge (x_2 \vee x_3).$$

Закон определяет правило выноса общего высказывания за скобку.

12. **Законы де Моргана (законы общей инверсии или дуальности)**

$$\overline{x_1} \vee \overline{x_2} = \overline{x_1 \wedge x_2}\ ;\quad x_1 \vee x_2 = \overline{\overline{x_1} \wedge \overline{x_2}}\ ;$$
$$\overline{x_1} \wedge \overline{x_2} = \overline{x_1 \vee x_2}\ ;\quad x_1 \wedge x_2 = \overline{\overline{x_1} \vee \overline{x_2}}\ .$$

Расширенный закон де Моргана:

$$\overline{x_1} \vee \overline{x_2} \vee \ldots \vee \overline{x_n} = \overline{x_1 \wedge x_2 \wedge \ldots \wedge x_n}\ ;$$
$$\overline{x_1} \wedge \overline{x_2} \wedge \ldots \wedge \overline{x_2} = \overline{x_1 \vee x_2 \vee \ldots \vee x_n}\ .$$

Законы де Моргана можно рассматривать как частный случай теоремы Шеннона, которая утверждает, что инверсия любой функции в алгебре логики получается путем замены каждой переменной ее инверсией и одновременно взаимной заменой символов конъюнкции и дизъюнкции.

13. **Закон поглощения (элиминации)**

$$x_1 \vee (x_1 \wedge x_2) = x_1\ ;\quad x_1 \wedge (x_1 \vee x_2) = x_1\ .$$

14. **Закон склеивания (исключения)**

$$(x_1 \wedge x_2) \vee (x_1 \wedge \overline{x_2}) = x_1\ ;\quad (x_1 \vee x_2) \wedge (x_1 \vee \overline{x_2}) = x_1\ .$$

Справедливость приведенных законов можно доказать табличным способом: выписать все наборы значений x_1 и x_2, вычислить на них значения левой и правой частей доказываемого выражения и убедиться, что результирующие столбцы совпадут.

В дальнейшем для облегчения восприятия приводимых логических выражений знак конъюнкции будем опускать и записывать логическое произведение как обычное, то есть выражение типа $x_1 \wedge x_2 \wedge x_3$ будем писать в виде $x_1 x_2 x_3$. В ряде случаев, чтобы избежать слияния знаков отрицания, между переменными может вставляться точка, например $x_1 \overline{x_2} \cdot \overline{x_3}$.

Дополнительные тождества алгебры логики

В данном разделе без доказательства приводятся некоторые дополнительные тождества алгебры логики, которые могут оказаться полезными при минимизации ФАЛ. Некоторые из них представляют собой лишь иную форму записи ранее рассмотренных тождеств.

$$x_1 \vee (\overline{x_1} \wedge x_2) = x_1 \vee x_2\ ;$$
$$x_1 \wedge (\overline{x_1} \vee x_2) = x_1 \wedge x_2\ ;$$

$$x_1x_2 \vee \overline{x_1}x_3 \vee x_2x_3 = x_1x_2 \vee \overline{x_1}x_3;$$

$$x_1(\overline{x_1} \vee x_2) = x_1x_2;$$

$$(x_1 \vee x_2)(\overline{x_1} \vee x_3) = x_1x_3 \vee \overline{x_1}x_2;$$

$$(x_1 \vee x_2)(\overline{x_1} \vee x_3)(x_2 \vee x_3) = (x_1 \vee x_2)(\overline{x_1} \vee x_3).$$

Логический базис

Любую булеву функцию с произвольным количеством аргументов можно построить через суперпозицию элементарных логических функций одной и двух переменных. Набор простейших функций, с помощью которого можно выразить любые другие, сколь угодно сложные логические функции, называется *функционально полным набором*, или *логическим базисом*. На практике в качестве базисов обычно используются следующие системы логических функций:

- «НЕ», «И», «ИЛИ» — булев базис;
- «И-НЕ» — универсальный базис Шеффера;
- «ИЛИ-НЕ» — универсальный базис Пирса;
- «Исключающее ИЛИ», «И», «Константа 1» — базис Жегалкина;
- «Запрет по X_2», «1».

Логический базис называется *минимальным*, если удаление хотя бы одной из входящих в него функций превращает этот набор в функционально неполный. Логический базис «И», «ИЛИ», «НЕ» не является минимальным, так как с помощью формул де Моргана можно исключить из логических выражений либо функцию «И», либо функцию «ИЛИ». В результате получаются минимальные базисы: «И», «НЕ» и «ИЛИ», «НЕ». Некоторые функции сами по себе представляют собой минимальный логический базис. К таким, например, относятся функции «И-НЕ» и «ИЛИ-НЕ». Чаще всего для записи логических выражений и их последующего преобразования используется базис «И», «ИЛИ», «НЕ».

Аналитическое представление булевых функций

В качестве исходного описания сложных логических функций обычно используется таблица истинности, однако упрощение функций выгоднее производить в аналитической форме. При аналитической записи ФАЛ представляется либо в виде логической суммы элементарных логических произведений (дизъюнкции элементарных конъюнкций), либо в виде логического произведения элементарных логических сумм (конъюнкции элементарных дизъюнкций). Первая форма записи носит название дизъюнктивной нормальной формы (ДНФ), вторая — конъюнктивной нормальной формы (КНФ).

Сначала определим основные понятия и терминологию, используемые при аналитическом представлении ФАЛ.

Элементарной конъюнкцией (элементарным логическим произведением) называют логическое произведение любого количества переменных (аргументов), взятых с отрицанием или без, но каждый из аргументов встречается в этом произведении лишь однократно. Элементарная конъюнкция, включающая все без исключения аргументы ФАЛ, носит название *полной элементарной конъюнкции*.

Дизъюнктивная нормальная форма (ДНФ) — это логическая сумма элементарных логических произведений (дизъюнкция элементарных конъюнкций). Термин «нормальная» означает, что в выражении отсутствуют групповые инверсии, то есть общий знак отрицания над несколькими переменными сразу, например, $\overline{x_1 x_2}$. Примером ДНФ может служить выражение $x_1 x_2 \vee \overline{x_1} x_2 x_3 \vee x_1 \overline{x_2} \cdot \overline{x_3}$.

Минтерм (единичный набор функции) — это логическое произведение всех переменных, взятых с отрицанием или без (полная элементарная конъюнкция), соответствующее набору аргументов, на которых функция принимает значение «1». Каждый минтерм соответствует одной строке таблице истинности, причем только такой, где значение функции равно 1 (True). Множество всех минтермов образует *единичное множество функции*.

Совершенной дизъюнктивной нормальной формой (СДНФ) называется ДНФ, состоящая из всех минтермов булевой функции. Для получения СДНФ на основе таблицы истинности необходимо:

- каждый из входных наборов, на которых булева функция принимает значение «1», представить в виде элементарного произведения (конъюнкции), причем если переменная равна 0, то она входит в конъюнкцию с инверсией, а если 1 — то без инверсии;
- полученные элементарные логические произведения объединить знаками логического сложения (дизъюнкции).

Число элементарных произведений (минтермов) в СДНФ равно числу строк таблицы истинности, на которых функция имеет значение «1». Для получения минтерма в него нужно включить все аргументы, причем те из них, которые имеют в данном наборе нулевое значение, входят в минтерм со знаком отрицания. Так, таблице истинности

x_1	x_2	f
0	0	1
0	1	1
1	0	1
1	1	0

отвечает совершенная дизъюнктивная нормальная форма $f = \overline{x_1} \cdot \overline{x_2} \vee \overline{x_1} x_2 \vee x_1 \overline{x_2}$.

Элементарная дизъюнкция (элементарная логическая сумма) — это логическая сумма (дизъюнкция) любого числа переменных, взятых с отрицанием или без, причем каждый отдельный аргумент встречается лишь однократно. Элементарная дизъюнкция, включающая все без исключения аргументы ФАЛ, называется *полной элементарной дизъюнкцией*.

Конъюнктивная нормальная форма (КНФ) — это логическое произведение элементарных логических сумм (конъюнкция элементарных дизъюнкций). Термин «нормальная» означает то же, что и в случае ДНФ, то есть отсутствие общей инверсии над несколькими переменными сразу, например $\overline{x_1 \vee x_2 \vee x_3}$. В качестве примера КНФ можно привести выражение $(x_1 \vee x_2)(\overline{x_1} \vee x_2 \vee x_3)(x_1 \vee \overline{x_2} \vee \overline{x_3})$.

Макстерм (нулевой набор функции) — это логическая сумма (дизъюнкция) всех аргументов (полная элементарная дизъюнкция), соответствующая набору аргументов, на которых функция принимает значение «0». Множество нулевых наборов называют *нулевым множеством функции*.

Совершенной конъюнктивной нормальной формой (СКНФ) называется КНФ, состоящая из всех макстермов булевой функции. Для получения СКНФ на основе таблицы истинности необходимо:

- каждый из входных наборов, на которых булева функция принимает значения «0», представить в виде элементарной логической суммы (дизъюнкции), причем если переменная равна 1, то она входит в эту сумму с инверсией, а если 0 — то без инверсии;
- полученные элементарные логические суммы объединить знаками логического умножения.

Число элементарных дизъюнкций в СКНФ равно числу строк таблицы истинности, на которых функция имеет значение 0. Для получения макстерма в него нужно включить все аргументы, причем те из них, которые имеют в данном наборе единичное значение, входят в элементарную дизъюнкцию со знаком отрицания. Так, таблице истинности

x_1	x_2	f
0	0	0
0	1	0
1	0	1
1	1	0

отвечает СКНФ $f = (x_1 \vee x_2)(x_1 \vee \overline{x_2})(\overline{x_1} \vee \overline{x_2})$.

Как минтермы, так и макстермы часто определяют общим термином «*терм*». Из вышесказанного следует общее правило установки знака отрицания над переменной терма (как минтерма, так и макстерма):

если значение переменной в таблице истинности отличается от значения логической функции, над такой переменной в терме ставится знак отрицания, в противном же случае знак отрицания не ставится.

Для перехода от таблицы истинности к эквивалентному аналитическому описанию используются совершенные ДНФ и КНФ.

Иногда удобнее пользоваться не самой логической функцией, а ее инверсией. В этом случае для записи СДНФ надо использовать нулевые, а для записи СКНФ — единичные значения функции.

По целому ряду причин более распространенной формой представления ФАЛ является СДНФ, поэтому в дальнейшем основное внимание будет уделено именно этой форме аналитической записи булевых функций.

Минимизация логических функций

Проблема минимизации логических выражений проистекает из практических задач создания логических схем. В качестве исходной аналитической формы обычно рассматривают совершенные ДНФ и КНФ, которые во многих случаях оказываются излишне сложными, из-за чего их техническая либо программная реализация получается избыточной. Для упрощения СДНФ и СКНФ используются различные *методы минимизации* — преобразования логической функции с целью упрощения ее аналитической записи.

К настоящему времени применяются следующие методы минимизации ФАЛ:

1) расчетный метод (метод непосредственных преобразований);
2) расчетно-табличный метод (метод Квайна);
3) метод Квайна–Мак-Класки (развитие метода Квайна);
4) метод Петрика (развитие метода Квайна);
5) табличный метод (метод карт Карно);
6) метод Блейка–Порецкого;
7) метод неопределенных коэффициентов;
8) метод гиперкубов;
9) метод факторизации;
10) метод функциональной декомпозиции и др.

Ниже будут рассмотрены первые пять методов, получившие наиболее широкое распространение.

Минимальный вариант булевой функции обычно ищут применительно к какому-либо логическому базису. Наилучшие результаты получаются с функциями «НЕ», «И», «ИЛИ», в силу чего именно эти функции в дальнейшем будем использовать в качестве основного логического базиса.

Сложность реализации логического выражения обычно характеризуют с помощью *коэффициента сложности* K_c. Для вычисления этого коэффициента нужно сложить количество термов, образующих выражение (слагаемых в ДНФ или сомножителей в КНФ), и сумму рангов всех этих термов. *Ранг терма* равен количеству переменных в терме, например, ранг терма $x_1 x_2 x_3$ равен трем. Следовательно, для функции $f = x_4 x_1 \vee x_3 x_2 \vee \overline{x_3} x_1 \vee \overline{x_4} \overline{x_3} \overline{x_2}$, содержащей 4 терма (три терма с рангом 2 и один терм с рангом 3), коэффициент сложности равен $K_c = 4 + (2 + 2 + 2 + 3) = 13$.

Целью минимизации является получение такой аналитической записи ФАЛ, которая имеет наименьший коэффициент сложности среди всех других эквивалентных вариантов записи данной функции. Подобную аналитическую запись логической функции называют *минимальной*, то есть можно сказать, что целью минимизации является получение *минимальной дизъюнктивной нормальной формы* (МДНФ) или *минимальной конъюнктивной нормальной формы* (МКНФ).

В основе практически всех методов минимизации лежит операция склеивания. Два элементарных произведения одинакового ранга (для ДНФ) или две элементарные суммы одинакового ранга (для КНФ) склеиваются, если они различаются только по одной переменной, а это различие состоит в присутствии знака инверсии над этой переменной в одном из произведений (сумм) и в его отсутствии в другом.

Прежде чем перейти к рассмотрению методов минимизации ФАЛ, приведем используемые в этом случае положения и определения.

Элементарные логические произведения, образовавшиеся в результате склеивания, называют *импликантами*. Склеивания начинаются с минтермов и продолжаются с импликантами, полученными в предшествующих операциях склеивания. Так, если СДНФ описывается выражением $f = \overline{x_1} \cdot \overline{x_2} \cdot \overline{x_3} \vee \overline{x_1} \cdot \overline{x_2} x_3 \vee \overline{x_1} x_2 \overline{x_3} \vee \overline{x_1} x_2 x_3$, то после склеивания минтермов $\overline{x_1} \cdot \overline{x_2} \cdot \overline{x_3}$ и $\overline{x_1} \cdot \overline{x_2} x_3$ получим импликанту $\overline{x_1} \cdot \overline{x_2}$, которая поглощает участвовавшие в склеивании минтермы. Склеивание другой пары минтермов — $\overline{x_1} x_2 \overline{x_3}$ и $\overline{x_1} x_2 x_3$ дает импликанту $\overline{x_1} x_2$, также поглощающую минтермы, из которых она была получена. В результате исходная ДНФ приобретает следующий вид: $f = \overline{x_1} \cdot \overline{x_2} \vee \overline{x_1} x_2$. Конъюнкции в правой части выражения (обратим внимание, что это уже не минтермы, а импликанты) также можно склеить, получив при этом импликанту $\overline{x_1}$. Импликанты, дальнейшее склеивание которых невозможно, называют *простыми*, или *первичными импликантами*. В нашем примере $\overline{x_1}$ является простой импликантой.

Выражение, полученное из совершенной ДНФ и состоящее только из простых импликант, носит название *сокращенной дизъюнктивной нормальной формы*.

Следующим этапом минимизации является нахождение тупиковых ДНФ. *Тупиковой дизъюнктивной нормальной формой* называется ДНФ, полученная из сокращенной ДНФ, в результате исключения из последней всех лишних простых импликант. В зависимости от того, какие из простых импликант были признаны лишними, может получиться несколько вариантов тупиковых ДНФ.

Тупиковая ДНФ, имеющая наименьший коэффициент сложности по сравнению с другими тупиковыми формами данной функции, носит название *минимальной дизъюнктивной нормальной формы* (МДНФ). Если среди возможных тупиковых форм имеется несколько с одинаковым коэффициентом сложности, это означает, что существует и несколько МДНФ.

Для приведенных выше понятий имеются аналоги, относящиеся к конъюнктивным нормальным формам. Приведем их.

Имплицентой называется элементарная логическая сумма, получаемая при склеивании пары макстермов или имплицент, образовавшихся в результате предшествующих склеиваний.

Простая имплицента — это имплицента, которую уже нельзя склеить ни с одной другой.

Сокращенная конъюнктивная нормальная форма — это конъюнкция всех простых имплицент.

Тупиковая конъюнктивная нормальная форма — это логическое произведение простых имплицент, из которых ни одна не является лишней.

Минимальная конъюнктивная нормальная форма (МКНФ) — тупиковая КНФ, имеющая наименьший коэффициент сложности среди других тупиковых форм данной ФАЛ.

Аналитическая минимизация производится в такой последовательности (описывается только применительно к ДНФ, поскольку для КНФ процедура аналогична):

- находят сокращенную дизъюнктивную нормальную форму (любая функция имеет только одну такую форму);
- находят все тупиковые нормальные формы;
- из полученных тупиковых форм выбирают минимальные формы.

Минимизация методом непосредственных преобразований

Исходной формой для этого метода минимизации служит СДНФ или СКНФ. Непосредственные преобразования логических формул служат для упрощения формул или приведения их к определенному виду путем использования основных правил алгебры логики. Некоторые преобразования логических формул похожи на преобразования формул в обычной алгебре (вынесение общего множителя за скобки, использование переместительного и сочетательного законов и т. п.), тогда как другие преобразования основаны на свойствах, которыми не обладают операции обычной алгебры (использование распределительного закона для конъюнкции, законов поглощения, склеивания, де Моргана и др.).

Минимизацию обычно проводят в такой последовательности (описывается только процесс минимизации СДНФ, поскольку минимизация СКНФ производится по аналогичной схеме). Сначала ищутся пары минтермов, отличающихся друг от друга только знаком инверсии и лишь в одном из аргументов. Такие минтермы склеиваются. В результате склеивания из двух минтермов образуется импликанта, ранг которой на единицу меньше, чем у склеиваемых минтермов:

$$x_1\overline{x_2}x_3 \vee \overline{x_1}\cdot\overline{x_2}\cdot\overline{x_3} \vee x_1x_2x_3 = x_1x_3(\overline{x_2} \vee x_2) \vee \overline{x_1}\cdot\overline{x_2}\cdot\overline{x_3} = x_1x_3 \vee \overline{x_1}\cdot\overline{x_2}\cdot\overline{x_3}\,.$$

С импликантами, образовавшимися в результате первого этапа склеивания, по возможности проводится очередной этап склеивания. Процесс продолжают до тех пор, пока дальнейшее склеивание импликант становится невозможным, то есть до получения простых импликант. Выражение, образованное только из простых импликант, как уже отмечалось, носит название сокращенной дизъюнктивной нормальной формы.

Далее предпринимается попытка исключить из сокращенной ДНФ избыточные простые импликанты, используя для этого прочие правила булевой алгебры, например теоремы противоречия, «исключенного третьего», закон поглощения и т. д. В зависимости от применяемых правил и порядка их использования может получиться несколько вариантов нормальных форм, в которых ни одну из входящих в них простых импликант уже нельзя исключить (тупиковых ДНФ).

Наконец, из тупиковых форм выбирается та, которая имеет наименьший коэффициент сложности (содержит минимальное суммарное количество букв и термов). Это и будет минимальная дизъюнктивная нормальная форма. Если минимальный коэффициент сложности одновременно имеет несколько тупиковых форм, то имеют место и несколько МДНФ.

Пример 1. Функция трех аргументов f задана таблицей истинности:

x_1	x_2	x_3	f
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

$$f = \overline{x_1} \cdot \overline{x_2} \cdot \overline{x_3} \vee \overline{x_1} \cdot \overline{x_2} x_3 \vee \overline{x_1} x_2 \overline{x_3} \vee \overline{x_1} x_2 x_3 = \overline{x_1} \cdot \overline{x_2} (\overline{x_3} \vee x_3) \vee \overline{x_1} x_2 (\overline{x_3} \vee x_3) =$$
$$= \overline{x_1} \cdot \overline{x_2}(1) \vee \overline{x_1} x_2 (1) = \overline{x_1} \cdot \overline{x_2} \vee \overline{x_1} x_2 = \overline{x_1} (\overline{x_2} \vee x_2) = \overline{x_1}(1) = \overline{x_1}.$$

Очевидно, что полученное выражение является минимальной дизъюнктивной нормальной формой.

Пример 2. Функция четырех аргументов представлена в табл. Б.7.

Таблица Б.7. Таблица истинности к примеру

Номер набора	x_1	x_2	x_3	x_4	f
0	0	0	0	0	1
1	0	0	0	1	1
2	0	0	1	0	1
3	0	0	1	1	0
4	0	1	0	0	0
5	0	1	0	1	1
6	0	1	1	0	1
7	0	1	1	1	1
8	1	0	0	0	1
9	1	0	0	1	1
10	1	0	1	0	1
11	1	0	1	1	0
12	1	1	0	0	0
13	1	1	0	1	0
14	1	1	1	0	1
15	1	1	1	1	0

Минимизация функции может быть осуществлена следующим образом:

$$
\begin{aligned}
f = {} & \overline{x_1}\cdot\overline{x_2}\cdot\overline{x_3}\cdot\overline{x_4} \vee \overline{x_1}\cdot\overline{x_2}\cdot\overline{x_3}x_4 \vee \overline{x_1}\cdot\overline{x_2}x_3\overline{x_4} \vee \overline{x_1}x_2\overline{x_3}\overline{x_4} \vee \overline{x_1}x_2x_3\overline{x_4} \vee \overline{x_1}x_2x_3x_4 \vee \\
& \vee x_1\overline{x_2}\cdot\overline{x_3}\cdot\overline{x_4} \vee x_1\overline{x_2}\cdot\overline{x_3}x_4 \vee x_1\overline{x_2}x_3\overline{x_4} \vee x_1x_2x_3\overline{x_4} = \\
= {} & (\overline{x_1}\cdot\overline{x_2}\cdot\overline{x_3}\cdot\overline{x_4} \vee \overline{x_1}\cdot\overline{x_2}\cdot\overline{x_3}x_4) \vee (\overline{x_1}\cdot\overline{x_2}x_3\overline{x_4} \vee \overline{x_1}x_2x_3\overline{x_4}) \vee \\
& \vee (\overline{x_1}x_2\overline{x_3}x_4 \vee \overline{x_1}x_2x_3x_4) \vee (x_1\overline{x_2}\cdot\overline{x_3}\cdot\overline{x_4} \vee x_1\overline{x_2}\cdot\overline{x_3}x_4) \vee (x_1\overline{x_2}x_3\overline{x_4} \vee x_1x_2x_3\overline{x_4}) = \\
= {} & \overline{x_1}\cdot\overline{x_2}\cdot\overline{x_3}(\overline{x_4} \vee x_4) \vee \overline{x_1}x_3\overline{x_4}(\overline{x_2} \vee x_2) \vee \overline{x_1}x_2x_4(\overline{x_3} \vee x_3) \vee \\
& \vee x_1\overline{x_2}\cdot\overline{x_3}(\overline{x_4} \vee x_4) \vee x_1x_3\overline{x_4}(\overline{x_2} \vee x_2) = \\
= {} & \overline{x_1}\cdot\overline{x_2}\cdot\overline{x_3} \vee \overline{x_1}x_3\overline{x_4} \vee \overline{x_1}x_2x_4 \vee x_1\overline{x_2}\cdot\overline{x_3} \vee x_1x_3\overline{x_4} = \\
= {} & \overline{x_2}\cdot\overline{x_3}(\overline{x_1} \vee x_1) \vee \overline{x_1}x_2x_4 \vee x_3\overline{x_4}(\overline{x_1} \vee x_1) = \\
= {} & \overline{x_2}\cdot\overline{x_3} \vee \overline{x_1}x_2x_4 \vee x_3\overline{x_4} = \overline{x_2}\cdot\overline{x_3} \vee x_3\overline{x_4} \vee \overline{x_1}x_2x_4.
\end{aligned}
$$

Получена тупиковая ДНФ, которая в данном случае совпадает с минимальной.

Ниже приводятся некоторые приемы и способы, которые могут быть полезными при упрощении логических выражений:

$$\overline{x_0 \vee x_1}(x_0 \vee \overline{x_1}) = 0;$$

$$\overline{x_0}x_1 \vee \overline{x_0 \vee x_1} \vee x_0 = 1;$$

$$(x_0 \vee x_1)(\overline{x_0} \vee x_1)(\overline{x_0} \vee \overline{x_1}) = \overline{x_0}x_1;$$

$$x_0\overline{x_1} \vee \overline{x_0}x_1x_2 \vee x_0x_2 = x_0\overline{x_1} \vee x_1x_2;$$

$$\overline{x_0x_1 \vee \overline{x_2}} = (\overline{x_0} \vee \overline{x_1})x_2$$

$$x_0x_1 \vee x_0x_1x_2 \vee x_0x_2x_3 = x_0(x_1 \vee x_2x_3);$$

$$x_0 \vee \overline{x_1\overline{x_2}} \vee \overline{\overline{x_0} \vee x_1 \vee \overline{x_2}} = x_0 \vee x_2 \vee \overline{x_1};$$

$$x_0x_1 \vee x_0x_1x_2 \vee x_0x_1x_2 \vee x_0x_1x_2 = x_0;$$

$$(x_0x_1 \vee x_2)(x_0 \vee x_1) \vee x_2 = x_0 \vee x_2 \vee x_1$$

$$x_0x_1(x_0x_2 \vee x_0x_1x_2 \vee x_2x_3) = x_0x_1.$$

При упрощении ФАЛ не всегда ясно, какое из правил алгебры логики следует применить на том или ином шаге. Навыки приходят с опытом.

Минимизация по методу Квайна

Метод Квайна (Willard van Orman Quine 1908–2000) по своей сути идентичен как по отношению к ДНФ, так и к КНФ, поэтому рассмотрим его на примере дизъюнктивной формы. В основе метода лежит использование двух основных законов алгебры логики — закона склеивания и закона поглощения. Процедура минимизации

проводится в несколько этапов (рассмотрим их на примере функции, приведенной в табл. Б.7):

1. **Нахождение простых импликант**

Перебирают все пары минтермов исходной СДНФ, склеивая те из них, для которых эта операция возможна. При составлении пар любой из минтермов может быть использован многократно. Процедура повторяется и над полученными импликантами — опять проводятся операции склеивания. Процесс повторяется до тех пор, пока не остается ни одной импликанты, допускающей склеивания с другими. Напомним, что такие импликанты называются простыми. Участвовавшие в склеивании элементарные конъюнкции помечаются каким-либо символом, например «*». Наличие такого символа означает, что данная элементарная конъюнкция уже учтена в какой-то из импликант, полученных в результате склеивания, и поглощается последней. Если какой-либо из минтермов изначально не удалось склеить ни с одним другим, то он уже сам по себе является простой импликантой. Дизъюнктивная нормальная форма, составленная из всех простых импликант, полученных в результате описанной процедуры, представляет собой сокращенную ДНФ, эквивалентную исходной СДНФ. Данный этап иллюстрирует табл. Б.8 и Б.9.

В табл. Б.8 показаны минтермы исходной совершенной ДНФ. Анализируются все возможные пары минтермов и, если это возможно, производится их склеивание. Минтермы, участвовавшие в операции склеивания, помечаются символом «*». Результаты склеивания с указанием номеров «склеенных» минтермов показаны в табл. Б.9.

Таблица Б.8. Минтермы совершенной ДНФ

Номер набора	Минтермы	Номер набора	Минтермы
0	$\overline{x_1} \cdot \overline{x_2} \cdot \overline{x_3} \cdot \overline{x_4}$ *	7	$\overline{x_1} x_2 x_3 x_4$ *
1	$\overline{x_1} \cdot \overline{x_2} \cdot x_3 \overline{x_4}$ *	8	$x_1 \overline{x_2} \cdot \overline{x_3} \cdot \overline{x_4}$ *
2	$\overline{x_1} \cdot x_2 \overline{x_3} \overline{x_4}$ *	9	$x_1 \overline{x_2} \cdot x_3 \overline{x_4}$ *
5	$\overline{x_1} x_2 \overline{x_3} x_4$ *	10	$x_1 \overline{x_2} x_3 \overline{x_4}$ *
6	$\overline{x_1} x_2 x_3 \overline{x_4}$ *	14	$x_1 x_2 x_3 \overline{x_4}$ *

Таблица Б.9. Склеивание минтермов совершенной ДНФ

Склеиваемые наборы	Импликанты	Склеиваемые наборы	Импликанты
0,1	$\overline{x_1} \cdot \overline{x_2} \cdot \overline{x_3}$ *	5,7	$\overline{x_1} x_2 x_4$
0,2	$\overline{x_1} \cdot \overline{x_2} \cdot \overline{x_4}$ *	6,7	$\overline{x_1} x_2 x_3$
0,8	$\overline{x_2} \cdot \overline{x_3} \cdot \overline{x_4}$ *	6,14	$x_2 x_3 \overline{x_4}$ *

продолжение ⇩

Таблица Б.9 (*продолжение*)

Склеиваемые наборы	Импликанты	Склеиваемые наборы	Импликанты
1,5	$\overline{x_1} \cdot \overline{x_3} x_4$	8,9	$x_1 \overline{x_2} \cdot \overline{x_3}$ *
1,9	$\overline{x_2} \cdot \overline{x_3} x_4$ *	8,10	$x_1 \overline{x_2} \cdot \overline{x_4}$ *
2,6	$\overline{x_1} x_3 \overline{x_4}$ *	10,14	$x_1 x_3 \overline{x_4}$ *
2,10	$\overline{x_2} x_3 \overline{x_4}$ *		

В результате первого этапа склеивания получены 13 импликант ранга 3. Поскольку помеченными оказались все минтермы, среди них простых импликант нет, тем самым в 13 импликантах содержится вся исходная информация, и ДНФ, составленная из этих импликант, полностью эквивалентна исходной совершенной ДНФ. Далее повторим процедуру попарного склеивания применительно к полученным 13 импликантам (табл. Б.10).

Таблица Б.10. Склеивание импликант ранга 3

Склеиваемые наборы	Импликанты
0,1,8,9	$\overline{x_2} \cdot \overline{x_3}$
0,2,8,10	$\overline{x_2} \cdot \overline{x_4}$
0,8,1,9	$\overline{x_2} \cdot \overline{x_3}$
0,8,2,10	$\overline{x_2} \cdot \overline{x_4}$
2,6,10,14	$x_3 \overline{x_4}$
2,10,6,14	$x_3 \overline{x_4}$

Из таблицы видно, что импликанты, обозначенные как 1,5; 5,7 и 6,7, остались непомеченными. Это означает, что они не были склеены ни с одной другой импликантой, поэтому являются простыми и должны учитываться на последующих этапах минимизации. В ходе склеивания образовались три пары импликант ранга 2. В соответствии с теоремой идемпотентности из нескольких одинаковых импликант можно оставить только одну. Нетрудно заметить, что дальнейшее склеивание трех оставшихся импликант ранга 2 невозможно, то есть эти импликанты — простые. Таким образом, в дополнение к трем простым импликантам ранга 3: $\overline{x_1} \cdot \overline{x_3} x_4$, $\overline{x_1} x_2 x_4$ и $\overline{x_1} x_2 x_3$ получены еще три простые импликанты ранга 2: $\overline{x_2} \cdot \overline{x_3}$, $\overline{x_2} \cdot \overline{x_4}$ и $x_3 \overline{x_4}$. Логическая сумма перечисленных простых импликант представляет собой сокращенную ДНФ: $f = \overline{x_1} \cdot \overline{x_3} x_4 \vee \overline{x_1} x_2 x_4 \vee \overline{x_1} x_2 x_3 \vee \overline{x_2} \cdot \overline{x_3} \vee \overline{x_2} \cdot \overline{x_4} \vee x_3 \overline{x_4}$.

2. **Составление импликантной матрицы и расстановка меток избыточности**

Этот и последующие шаги имеют целью убрать из сокращенной ДНФ все лишние простые импликанты, то есть перейти от сокращенной ДНФ к тупиковым формам, а затем и к минимальным. Задача решается с помощью специальной *импликантной матрицы*. Каждая строка такой матрицы соответствует одной из простых импликант, входящих в сокращенную ДНФ, иными словами, количество строк в матрице равно числу простых импликант в сокращенной ДНФ. Столбцы матрицы представляют минтермы исходной СДНФ, при этом каждому минтерму соответствует свой столбец.

Если минтерм в столбце импликантной матрицы содержит в себе простую импликанту из какой-либо строки матрицы, то на пересечении данного столбца и данной строки ставится метка избыточности. Это означает, что данная простая импликанта поглощает соответствующий минтерм и способна заменить его в окончательном логическом выражении.

Таблица Б.11. Импликантная матрица Квайна

		0	1	2	5	6	7	8	9	10	14
0,1,8,9	$\overline{x_2} \cdot \overline{x_3}$	√	√					√	√		
0,2,8,10	$\overline{x_2} \cdot \overline{x_4}$	√		√				√		√	
2,6,10,14	$x_3\overline{x_4}$			√		√				√	√
1,5	$\overline{x_1} \cdot \overline{x_3}x_4$		√		√						
5,7	$\overline{x_1}x_2x_4$				√		√				
6,7	$\overline{x_1}x_2x_3$					√	√				

Импликантная матрица для рассматриваемого примера (табл. Б.11) содержит 6 строк (по числу простых импликант) и 10 столбцов (по числу минтермов исходной СДНФ). В матрице минтермы обозначены своими номерами, а слева от простых импликант перечислены номера минтермов, из которых эти импликанты были получены. Расставим в ней метки в тех позициях, где простая импликанта, указанная в левом столбце, покрывает минтерм, записанный в верхней строке.

3. **Нахождение существенных импликант и исключение связанных с ними строк и столбцов**

Присутствие в столбце только одной метки означает, что простая импликанта строки, где стоит эта метка, является *существенной*, или *базисной импликантой*, то есть обязательно войдет в минимальную ДНФ. Строка, содержащая существенную импликанту, а также столбцы, на пересечении с которыми в этой строке стоит метка

избыточности, вычеркиваются. Это позволяет упростить последующие шаги минимизации. Если после упомянутого вычеркивания в оставшейся части таблицы появятся строки, не содержащие меток или содержащие идентично расположенные метки, то такие строки также вычеркиваются. В последнем случае оставляют одну — ту, в которой простая импликанта имеет наименьший ранг среди остальных вычеркиваемых импликант.

Таблица Б.12. Удаление из импликантной матрицы существенных импликант и покрываемых ими минтермов

		0	1	2	5	6	7	8	9	10	14
0,1,8,9	$\overline{x_2} \cdot \overline{x_3}$	√	√					√	√		
0,2,8,10	$\overline{x_2} \cdot \overline{x_4}$	√		√				√		√	
2,6,10,14	$x_3\overline{x_4}$			√		√				√	√
1,5	$\overline{x_1} \cdot x_3x_4$		√		√						
5,7	$\overline{x_1}x_2x_4$				√		√				
6,7	$\overline{x_1}x_2x_3$					√	√				

В нашей функции по одной метке имеют столбцы 9 и 14, следовательно, имеют место две существенные импликанты: $\overline{x_2} \cdot \overline{x_3}$ и $x_3\overline{x_4}$. С учетом этого поиск остальных импликант минимальной ДНФ можно упростить, исключив строки с существенными импликантами, а также перекрываемые ими столбцы. Это показано в табл. Б.12 (удаляемые строки и столбцы закрашены).

После вычеркивания существенных импликант $\overline{x_2} \cdot \overline{x_3}$ и $x_3\overline{x_4}$, а также столбцов с минтермами, которые поглощаются этими импликантами, получим сокращенную матрицу (табл. Б.13).

Таблица Б.13. Сокращенная импликантная матрица

		5	7
0,2,8,10	$\overline{x_2} \cdot \overline{x_4}$		
1,5	$\overline{x_1} \cdot x_3x_4$	√	
5,7	$\overline{x_1}x_2x_4$	√	√
6,7	$\overline{x_1}x_2x_3$		√

Первая строка не содержит меток избыточности, поэтому ее можно удалить (табл. Б.14).

Таблица Б.14. Сокращенная импликантная матрица после исключения пустых строк

		5	7
1,5	$\overline{x_1} \cdot \overline{x_3} x_4$	√	
5,7	$\overline{x_1} x_2 x_4$	√	√
6,7	$\overline{x_1} x_2 x_3$		√

4. **Выбор минимального покрытия**

В сокращенной импликантной матрице (табл. Б.14) нужно выбрать такую минимально возможную совокупность строк, которая включает метки во всех столбцах («покрывает» все оставшиеся в таблице минтермы). Дизъюнкция простых импликант, соответствующих строкам этой совокупности, а также ранее вычеркнутых существенных импликант, образует тупиковую ДНФ. В общем случае полных покрытий с одинаковым числом строк, а значит, и тупиковых ДНФ может быть несколько.

Из матрицы (табл. Б.14) видно, что минимальное покрытие не исключенных ранее минтермов обеспечивает простая импликанта $\overline{x_1} x_2 x_4$ либо пара импликант $\overline{x_1} \cdot \overline{x_3} x_4$ и $\overline{x_1} x_2 x_3$. С учетом ранее выявленных существенных импликант получаем две тупиковые ДНФ:

$$f = \overline{x_1} x_2 x_4 \vee \overline{x_2} \cdot \overline{x_3} \vee x_3 \overline{x_4};$$

$$f = \overline{x_1} \cdot \overline{x_3} x_4 \vee \overline{x_1} x_2 x_3 \vee \overline{x_2} \cdot \overline{x_3} \vee x_3 \overline{x_4}.$$

5. **Определение и запись минимальной нормальной формы**

В случае нескольких тупиковых форм предпочтение отдается той из них, которая имеет наименьший коэффициент сложности. Если получилась лишь одна тупиковая ДНФ, то она одновременно является и минимальной.

Коэффициент сложности первой из двух получившихся тупиковых форм равен 10, а второй — 14. По этой причине минимальной ДНФ следует признать первое выражение:

$$f = \overline{x_1} x_2 x_4 \vee \overline{x_2} \cdot \overline{x_3} \vee x_3 \overline{x_4}.$$

Минимизация по методу Квайна–Мак-Класски

Метод Квайна–Мак-Класски отличается от метода Квайна только в той части, которая связана со способом нахождения простых импликант. Взамен громоздкой

процедуры склеивания всех возможных пар импликант Мак-Класски (Edward J. McCluskey) предложил следующую процедуру.

1. Все минтермы представляются соответствующими двоичными комбинациями — наборами.
2. Наборы разбиваются на группы. В *i*-ю группу объединяются наборы, содержащие *i* единиц.
3. Производятся все возможные операции склеивания, но только между наборами, входящими в соседние группы. В получаемых импликантах переменная, по которой происходило склеивание, заменяется каким-либо символом, например «–». Наборы, участвовавшие в склеивании, помечаются.
4. Процедуры, описанные в пунктах 2 и 3, повторяются применительно к импликантам, полученным на предыдущем этапе склеивания, до тех пор пока дальнейшее склеивание становится невозможным. Неотмеченные после склеивания наборы являются простыми импликантами, образующими сокращенную ДНФ.

Описанная модификация заменяет лишь первый шаг метода Квайна, при этом все последующие шаги производятся аналогично методу Квайна.

Проиллюстрируем описанную процедуру на примере ранее рассмотренной ФАЛ (см. табл. Б.7). Все наборы, на которых функция равна единице, распределим по группам (табл. Б.15). В *i*-ю группу включаются те наборы, которые содержат *i* единиц.

Таблица Б.15. Распределение минтермов по группам

Номер группы	Номер набора	Двоичный набор
0	0*	0000
1	1* 2* 8*	0001 0010 1000
2	5* 6* 9* 10*	0101 0110 1001 1010
3	7* 14*	0111 1110

Таблица Б.16. Первый этап склеивания членов групп

Группы	Склеиваемый наборы	Результат
0–1	0,1* 0,2* 0,8*	000– 00–0 –000
1–2	1,5 1,9* 2,6* 2,10* 8,9* 8,10*	0–01 –001 0–10 –010 100– 10–0

Группы	Склеиваемый наборы	Результат
2–3	5,7 6,7 6,14* 10,14*	01–1 011– –110 1–10

Теперь проведем все возможные операции склеивания между парами наборов, входящими в соседние группы, при этом переменную, по которой производилось склеивание, заменим символом «–» (табл. Б.16).

Теперь аналогично произведем операцию над элементами новых соседних групп (табл. Б.17). В колонке «Группы» указаны номера новых групп, образовавшихся после первого этапа склеивания.

Таблица Б.17. Второй этап склеивания

Группы	Склеиваемые наборы	Результат
0–1	0,1,8,9 0,2,8,10 0,8,1,9 0,8,2,10	–00– –0–0 –00– –0–0
1–2	2,6,10,14 2,10,6,14	––10 ––10

Из таблицы видно, дальнейшее склеивание невозможно, вследствие чего первый этап минимизации завершается.

Исключим повторяющиеся наборы (это мы можем сделать на основании теоремы идемпотентности). Оставшиеся, а также непомеченные наборы соответствуют простым импликантам, которые можно представить в привычной записи, заменив единицы и нули обозначениями переменных соответственно без знака инверсии и со знаком инверсии и исключив переменные, обозначенные знаком «–» (табл. Б.18).

Таблица Б.18. Соответствие двоичных наборов простым импликантам

Двоичный набор	Простая импликанта
0–01	$\overline{x_1} \cdot \overline{x_3} x_4$
01–1	$\overline{x_1} x_2 x_4$
011–	$\overline{x_1} x_2 x_3$
–00–	$\overline{x_2} \cdot \overline{x_3}$
–0–0	$\overline{x_2} \cdot \overline{x_4}$
–10	$x_3 \overline{x_4}$

Полученные простые импликанты образуют сокращенную ДНФ:

$$f = \overline{x_1} \cdot \overline{x_3} x_4 \vee \overline{x_1} x_2 x_4 \vee \overline{x_1} x_2 x_3 \vee \overline{x_2} \cdot \overline{x_3} \vee \overline{x_2} \cdot \overline{x_4} \vee x_3 \overline{x_4}.$$

Нетрудно заметить, что получены те же простые импликанты, что и в методе Квайна. Дальнейшие действия совпадают с соответствующими этапами процедуры Квайна.

Минимизация по методу Петрика

Метод Петрика (Petrick) также имеет целью упрощение метода Квайна, но в части нахождения всех тупиковых форм по импликантной матрице.

Сначала стандартный вид импликантной матрицы заменяется конъюнктивным ее представлением, для чего все простые импликанты обозначаются прописными латинскими буквами, то есть теперь каждой строке импликантной матрицы соответствует какая-то буква. Для каждого i-го столбца импликантной матрицы записывается дизъюнкция, элементами которой служат буквы, обозначающие строки, где на пересечении с i-м столбцом стоит метка избыточности. Конъюнктивное представление импликантной матрицы — это конъюнкция вышеупомянутых дизъюнкций, составленных для всех столбцов матрицы. Далее выполняется упрощение конъюнктивного представления импликантной матрицы. Для этого могут применяться все известные правила булевой алгебры. После упрощения (раскрытия скобок, поглощений и т. п.) получается дизъюнкция конъюнкций, соответствующая тупиковой ДНФ.

В качестве иллюстрации метода рассмотрим импликантную матрицу, представленную в табл. Б.11.

Требуется найти все тупиковые ДНФ.

Сначала обозначим все имеющиеся простые импликанты прописными латинскими буквами:

$$\overline{x_2} \cdot \overline{x_3} = A; \quad \overline{x_2} \cdot \overline{x_4} = B; \quad x_3 \overline{x_4} = C;$$
$$\overline{x_1} \cdot \overline{x_3} x_4 = D; \quad \overline{x_1} x_2 x_4 = E; \quad \overline{x_1} x_2 x_3 = F.$$

		Простые импликанты	Минтермы									
			0	1	2	5	6	7	8	9	10	14
A	**0,1,8,9**	$\overline{x_2} \cdot \overline{x_3}$	√	√					√	√		
B	**0,2,8,10**	$\overline{x_2} \cdot \overline{x_4}$	√		√				√		√	
C	**2,6,10,14**	$x_3 \overline{x_4}$			√		√				√	√
D	**1,5**	$\overline{x_1} \cdot \overline{x_3} x_4$		√		√						
E	**5,7**	$\overline{x_1} x_2 x_4$				√		√				
F	**6,7**	$\overline{x_1} x_2 x_3$					√	√				

Теперь запишем конъюнктивное представление импликантной матрицы w:

$$w = (A \vee B)(A \vee D)(B \vee C)(D \vee E)(C \vee F)(E \vee F)(A \vee B)A(B \vee C)C.$$

После удаления одинаковых членов получаем:

$$w = (A \vee B)(A \vee D)(B \vee C)(D \vee E)(C \vee F)(E \vee F)AC.$$

Далее произведем возможные поглощения, в результате чего выражение принимает вид:

$$w = (D \vee E)(E \vee F)AC = (DE \vee EE \vee DF \vee EF)AC = (DE \vee E \vee DF \vee EF)AC =$$
$$= (E \vee DF)AC = ACE \vee ACDF.$$

Поскольку окончательное выражение содержит два слагаемых, имеют место две тупиковые ДНФ, представленные как *ACE* и *ACDF*. Первая содержит три простых импликанты $A = \overline{x_2} \cdot \overline{x_3}$, $C = x_3\overline{x_4}$ и $E = \overline{x_1}x_2x_4$. Таким образом, первая тупиковая ДНФ имеет вид $f = \overline{x_2} \cdot \overline{x_3} \vee x_3\overline{x_4} \vee \overline{x_1}x_2x_4$. Вторая тупиковая ДНФ состоит из 4 простых импликант — $A = \overline{x_2} \cdot \overline{x_3}$, $C = x_3\overline{x_4}$, $D = \overline{x_1} \cdot \overline{x_3}x_4$ и $F = \overline{x_1}x_2x_3$ — и может быть записана как $f = \overline{x_2} \cdot \overline{x_3} \vee x_3\overline{x_4} \vee x_1 \cdot \overline{x_3}x_4 \vee \overline{x_1}x_2x_3$. Как видно, коэффициент сложности первой тупиковой формы равен 10, а второй — 14, то есть минимальная ДНФ совпадает с первой тупиковой ДНФ.

Минимизация табличным методом

Метод, предложенный в 1952 году Е. Вейчем и усовершенствованный в 1953 году М. Карно, являет собой один из наглядных методов минимизации ФАЛ. В обоих случаях минимизация выполняется с помощью специальных карт, известных как диаграмма Вейча и карта Карно соответственно. Диаграмма Вейча отличается от карты Карно нумерацией строк и столбцов. Если в диаграмме Вейча они нумеруются в порядке возрастания двоичных чисел, например 00, 01, 10, 11, то в карте Карно используется код Грея, предполагающий циклический порядок следования номеров: 00, 01, 11, 10. В результате матрица Карно более удобна в обращении, что станет понятным из последующего изложения.

Карта Карно реализует *координатный способ* представления ФАЛ, при котором таблица истинности заменяется прямоугольной координатной картой состояний, содержащей 2^n клеток (по числу возможных наборов аргументов). Аргументы разбиваются на две группы так, что одна группа определяет координаты столбца карты, а другая — координаты строки. При таком способе каждая клетка соответствует определенному набору аргументов ФАЛ. Внутрь клетки ставится значение функции на данном наборе. Если предполагается получение минимальной ДНФ, то заполняются лишь те клетки, для которых значение функции равно 1. В случае минимальной КНФ — клетки, соответствующие наборам, где значение ФАЛ равно 0.

Двоичные числа, записываемые вокруг карты, характеризуют значения аргументов. В отличие от таблиц истинности, где значения аргументов обычно следуют в естественной последовательности (00, 01, 10, 11), нумерация ячеек в карте Карно выполняется согласно коду Грея, и это является определяющим моментом. Особенность

кода Грея в том, что двоичные коды номеров соседних столбцов и строк отличаются только в одном бите. Существуют карты Карно на 2, 3, 4, 5 и 6 переменных, причем последние стали использоваться достаточно недавно. На рис. Б.3 представлены карты Карно для 2, 3, 4, 5 и 6 аргументов.

Единица в ячейке карты Карно соответствует единичному значению функции в таблице истинности. Благодаря системе нумерации ячеек минтермы, отличающиеся только в одном бите (такие минтермы называются *смежными*, или соседними), располагаются в соседних ячейках по горизонтали или по вертикали. Отметим также, что код Грея является циклическим, то есть с точки зрения «соседства» карта Карно представляет собой пространственную фигуру. Так, карта для 4 аргументов — это тор. Это означает, что одинаково расположенные клетки в левой и правой крайних колонках карты, а также в крайних верхнем и нижнем рядах карты следует рассматривать как смежные. Так как смежные клетки соответствуют наборам, различающимся только в одном бите, к ним можно применить операцию склеивания, что на карте представляется в виде группировки соответствующих клеток.

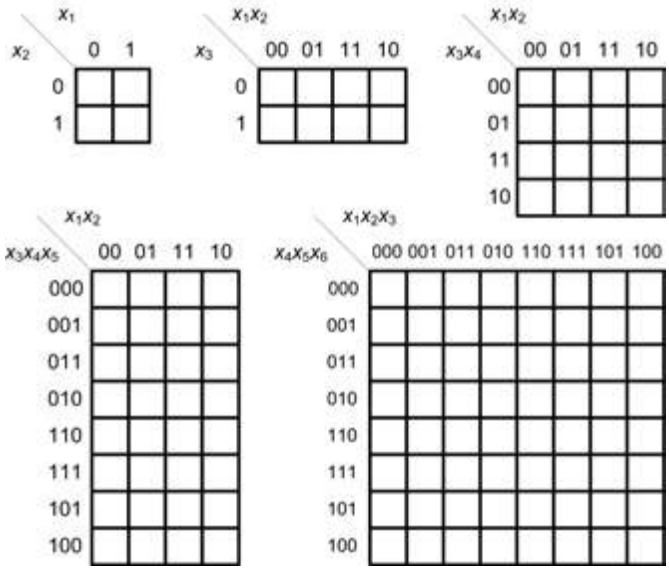


Рис. Б.3. Карты Карно для 2, 3, 4, 5 и 6 аргументов

Процедуру минимизации ФАЛ с помощью карт Карно можно описать следующим образом.

1. Заполнить единицами те клетки карты Карно, для которых значение функции равно 1 (для получения ДНФ), или же нулями те клетки, которые соответствуют нулевым значениям функции (для получения КНФ).
2. В карте Карно группы единиц (для получения ДНФ) или группы нулей (для получения КНФ) необходимо обвести четырехугольными контурами. Этот процесс соответствует операции склеивания или нахождения импликант данной функции.

3. Количество клеток внутри контура должно быть целой степенью двойки (1, 2, 4, 8, 16, ...).
4. При проведении контуров крайние строки карты (верхние и нижние, левые и правые), а также угловые клетки считаются соседними (для карт до 4-х переменных).
5. Каждый контур должен включать максимально возможное количество клеток, тогда он будет отвечать простой импликанте.
6. Все единицы (нули) в карте (даже одиночные) должны быть охвачены контурами. Любая единица (нуль) может входить в контуры произвольное количество раз.
7. Множество контуров, покрывающих все единицы (нули) в карте, образуют тупиковую ДНФ (КНФ). Целью минимизации является нахождение минимальной из множества тупиковых форм.
8. В элементарной конъюнкции (дизъюнкции), которая соответствует одному контуру, остаются только те переменные, значение которых не изменяется внутри обведенного контура. Переменные булевой функции входят в элементарную конъюнкцию (для значений функции 1) без инверсии, если их значение на соответствующих координатах равно 1, и с инверсией — если равно 0. Для значений булевой функции, равных 0, записываются элементарные дизъюнкции, куда переменные входят без инверсии, если их значение на соответствующих координатах равно 0, и с инверсией — если оно равно 1.

На рис. Б.4 приведен пример минимизации булевой функции трех аргументов с получением минимальной ДНФ. В примере все клетки, содержащие единицы, удалось охватить двумя контурами. В горизонтально ориентированном контуре значения переменных x_1 и x_3 совпадают, а значения x_2 различны. Таким образом, переменная x_2 является несущественной (фиктивной) и поэтому ее можно отбросить. Аналогично в вертикальном контуре можно избавиться от переменной x_3.

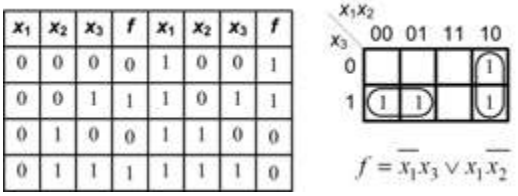


Рис. Б.4. Пример минимизации ФАЛ трех аргументов

Пример на рис. Б.5 иллюстрирует использование одной и той же клетки в нескольких контурах.

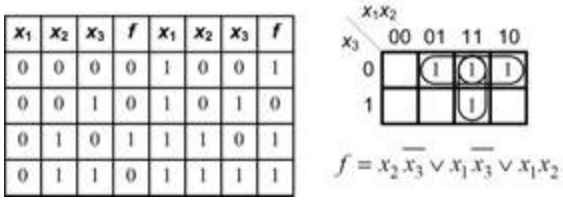


Рис. Б.5. Пример использования общей клетки в нескольких контурах

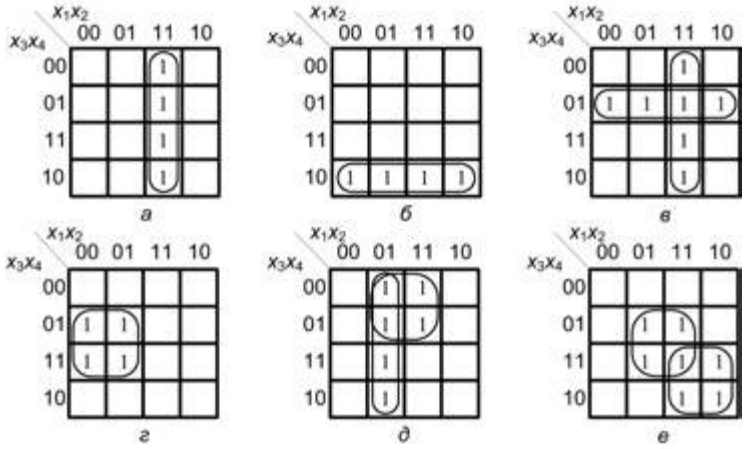


Рис. Б.6. Варианты группировки клеток для функции 4-х аргументов: *а* — $f = x_1x_2$; *б* — $f = x_3\overline{x_4}$; *в* — $f = x_1x_2 \vee \overline{x_3}x_4$; *г* — $f = \overline{x_1}x_4$; *д* — $f = x_2\overline{x_3} \vee \overline{x_1}x_2$; *е* — $f = x_2x_4 \vee x_1x_3$

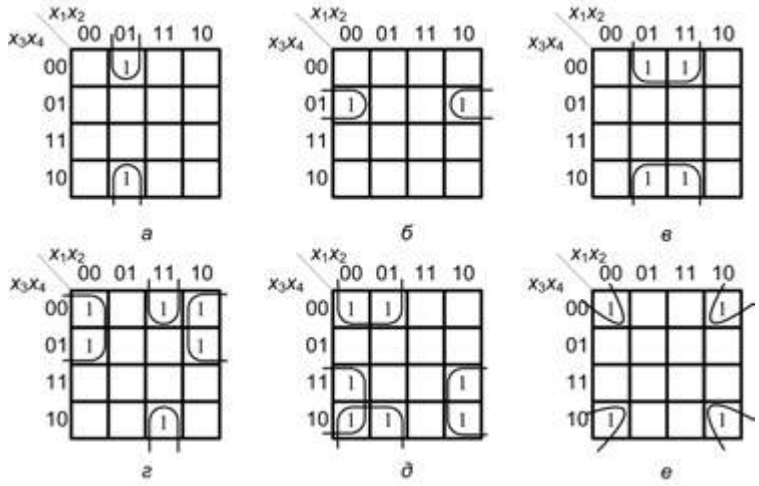


Рис. Б.7. Варианты группировки клеток, использующие циклический характер карт: *а* — $f = \overline{x_1}x_2\overline{x_4}$; *б* — $f = \overline{x_2} \cdot \overline{x_3}x_4$; *в* — $f = x_2\overline{x_4}$; *г* — $f = \overline{x_2} \cdot \overline{x_3} \vee x_1x_2\overline{x_4}$; *д* — $f = \overline{x_1} \cdot x_4 \vee \overline{x_2}x_3$; *е* — $f = \overline{x_2} \cdot \overline{x_4}$

Возможные варианты группировки клеток в картах Карно, в том числе и использующие циклический характер кода Грея, приведены на рис. Б.6 и Б.7.

Когда карту Карно заполняют единицами из таблицы истинности, результат записывают в виде дизъюнктивной нормальной формы. В качестве альтернативы возможно заполнение нулями клеток, соответствующих нулевым значениям ФАЛ. В этом случае группируют нули, а результат записывается в виде конъюнктивной нормальной формы. На рис. Б.8 показаны варианты минимизации логической функции трех аргументов с получением минимальных ДНФ (рис. Б.8, *а*) и КНФ (рис. Б.8, *б*).

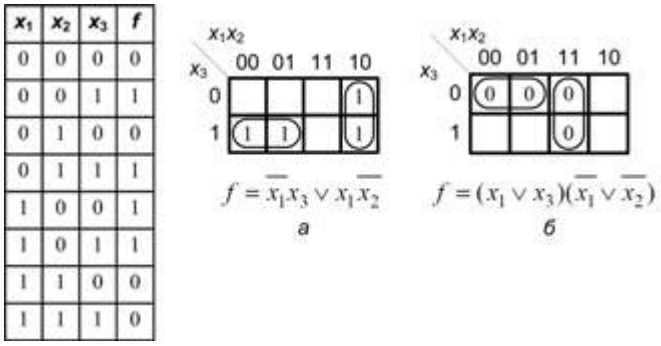


Рис. Б.8. Примеры минимизации ФАЛ с получением минимальной: *а* — ДНФ; *б* — КНФ

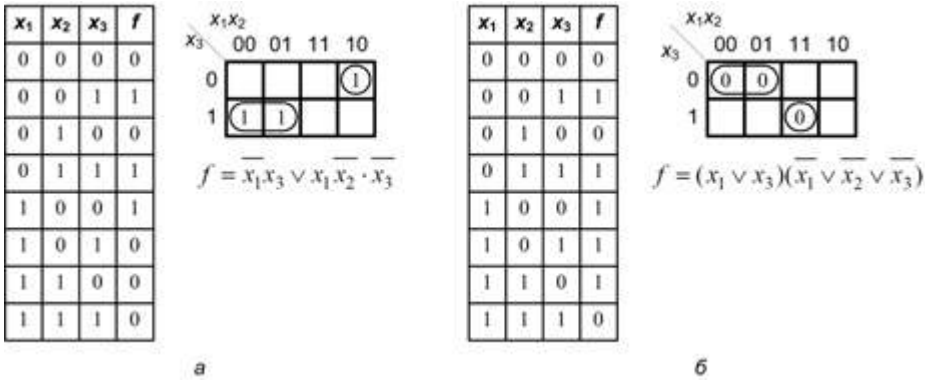


Рис. Б.9. Учет клеток, не имеющих в карте Карно соседних клеток: *а* — ДНФ; *б* — КНФ

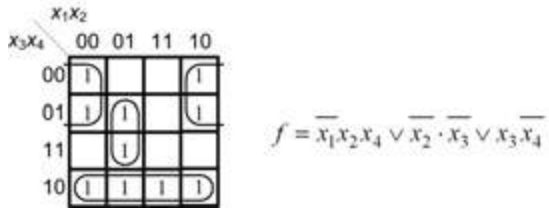


Рис. Б.10. Минимизация функции из табл. Б.7

Как в случае ДНФ, так и в случае КНФ элементы, не попавшие ни в одну из групп, необходимо добавить в результирующее выражение без каких-либо изменений (рис. Б.9).

В заключение в качестве примера рассмотрим минимизацию функции, которая приведена в табл. Б.7. Соответствующая карта Карно и результат минимизации показаны на рис. Б.10. Отметим, что метод карт Карно применим к минимизации булевых функций до 6-ти переменных: до 4-х переменных на плоскости и до 6-ти — в трехмерной интерпретации.

Минимизация частично определенных функций

Функция алгебры логики с n аргументами, значение которой определено на всех 2^n входных наборах, называется полностью определенной. Для реальных задач более характерен вариант, когда значения функции задаются только на части входных наборов. Логическая функция n аргументов, которая определена не на всех 2^n наборах, называется частично определенной. При задании частично определенной ФАЛ указываются номера лишь тех наборов, где функция равна нулю и единице. Остальные наборы считаются безразличными, или запрещенными. В таблице истинности вместо значения ФАЛ в строках с безразличными наборами записывается символ «*». Значение логической функции на безразличных наборах может доопределяться произвольно, причем независимо для каждого набора. Если число запрещенных комбинаций равно m, то путем доопределения можно получить 2^m различных функций. Естественно, что доопределение целесообразно производить таким образом, чтобы после минимизации ФАЛ имела наименьший возможный коэффициент сложности. Наиболее удобно это производить с помощью карт Карно.

Варианты доопределения частично определенных функций проиллюстрируем на примерах.

Пример 3. Функция трех аргументов задана таблицей истинности.

x_1	x_2	x_3	f	x_1	x_2	x_3	f
0	0	0	0	1	0	0	1
0	0	1	*	1	0	1	1
0	1	0	1	1	1	0	0
0	1	1	*	1	1	1	*

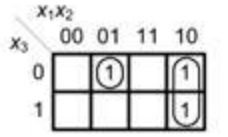
На рис. Б.11 показаны варианты доопределения данной функции.

Если значения функции на всех безразличных наборах принять равным 0, то получаем функцию, приведенную на рис. Б.11, *а*. МДНФ имеет вид $f = \overline{x_1}x_2\overline{x_3} \vee x_1\overline{x_2}$. МДНФ функции, в которой все *=1 (рис. Б.11, *б*) описывается выражением $f = x_3 \vee x_1x_2 \vee x_1\overline{x_2}$. Карта Карно позволяет найти вариант доопределения ФАЛ, при котором МДНФ будет иметь минимальный коэффициент сложности (рис. Б.11, *в*). Соответствующая этому варианту МДНФ имеет вид: $f = x_1\overline{x_2} \vee \overline{x_1}x_2$.

Пример 4. Найти минимальную форму для функции.

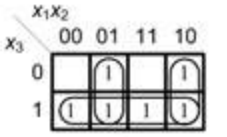
x_1	x_2	x_3	x_4	f	x_1	x_2	x_3	x_4	f
0	0	0	0	1	1	0	0	0	*
0	0	0	1	*	1	0	0	1	1
0	0	1	0	*	1	0	1	0	0
0	0	1	1	0	1	0	1	1	*
0	1	0	0	*	1	1	0	0	0
0	1	0	1	0	1	1	0	1	*
0	1	1	0	1	1	1	1	0	1
0	1	1	1	*	1	1	1	1	*

x_1	x_2	x_3	f
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0



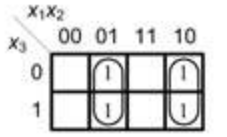
а

x_1	x_2	x_3	f
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1



б

x_1	x_2	x_3	f
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0



в

Рис. Б.11. Варианты доопределения частично определенной функции: *а* — нулями; *б* — единицами; *в* — оптимальным образом

Оптимальное доопределение функций можно проводить с помощью метода Квайна. Сначала примем условие, при котором значение функции на всех неопределенных наборах равно 1. Проведя первые этапы процедуры минимизации Квайна, получим сокращенную ДНФ:

$$f = \overline{x_1} \cdot \overline{x_4} \vee \overline{x_2} \cdot \overline{x_3} \vee x_2 x_3 \vee x_1 x_4.$$

Составим импликантную матрицу.

	$\overline{x_1} \cdot \overline{x_2} \cdot \overline{x_3} \cdot \overline{x_4}$	$\overline{x_1} x_2 x_3 \overline{x_4}$	$x_1 \overline{x_2} \cdot \overline{x_3} x_4$	$x_1 x_2 x_3 \overline{x_4}$
$\overline{x_1} \cdot \overline{x_4}$	√	√		
$\overline{x_2} \cdot \overline{x_3}$	√		√	
$x_2 x_3$		√		√
$x_1 x_4$			√	

Матрица позволяет получить минимальный вид результирующей функции $f = \overline{x_2} \cdot \overline{x_3} \vee x_2 x_3$. Этот результат получается при следующем доопределении исходной функции.

x_1	x_2	x_3	x_4	f	x_1	x_2	x_3	x_4	f
0	0	0	0	1	1	0	0	0	1
0	0	0	1	1	1	0	0	1	1
0	0	1	0	0	1	0	1	0	0
0	0	1	1	0	1	0	1	1	0
0	1	0	0	0	1	1	0	0	0
0	1	0	1	0	1	1	0	1	0
0	1	1	0	1	1	1	1	0	1
0	1	1	1	1	1	1	1	1	1

Тот же результат, как уже отмечалось, может быть достигнут с помощью карты Карно (рис. Б.12).

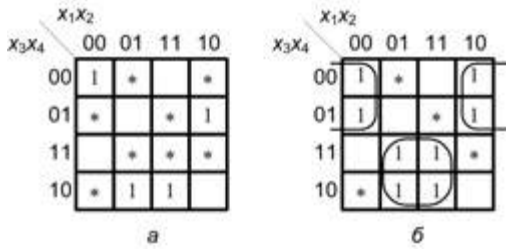


Рис. Б.12. Частично определенная функция: *а* — перед доопределением; *б* — после доопределения $f = \overline{x_2} \cdot \overline{x_3} \vee x_2 x_3$

Минимизация совокупности логических функций

В практике проектирования логических схем очень частым является случай, когда устройство имеет несколько выходов. Состояние каждого выхода в этом случае описывается с помощью отдельной ФАЛ, а общее описание устройства представляется совокупностью (системой) булевых функций. Если каждую ФАЛ этой совокупности минимизировать по отдельности, то общая схема устройства будет состоять из изолированных подсхем, в которых могут присутствовать одинаковые участки. Объединение подобных участков ведет к упрощению схемы в целом, но это возможно, только если уравнения совокупности ФАЛ будут минимизироваться не по отдельности, а совместно. Общая идея минимизации совокупности функций алгебры логики сводится к получению таких выражений, в которых оптимально используются члены, общие для нескольких функций.

Минимизация систем ФАЛ может производиться по алгоритму, похожему на метод Квайна, но с небольшими отличиями. Рассмотрим этот алгоритм на примере совокупности двух ФАЛ:

$$f_1 = x_1 \overline{x_2} x_3 \vee x_1 x_2 \overline{x_3} \vee x_1 \overline{x_2} \cdot \overline{x_3}$$

$$f_2 = x_1 x_2 x_3 \vee x_1 x_2 \overline{x_3} \vee x_1 \overline{x_2} x_3.$$

Сначала сформируем множество минтермов всех уравнений системы. Каждый минтерм снабдим признаком, приписав его в виде индекса. В качестве признака используем название функции, в которую входит минтерм: $\{x_1\overline{x_2}x_{3f_1}, x_1x_2\overline{x_3}_{f_1}, x_1\overline{x_2}\cdot\overline{x_3}_{f_1}, x_1x_2x_{3f_2}, x_1x_2\overline{x_3}_{f_2}, x_1\overline{x_2}x_{3f_2}\}$.

Из полученного множества выделим подмножество неодинаковых элементов. Из одинаковых элементов оставим по одному, изменив в них индекс так, чтобы он указывал на все логические выражения, в которые входит данный элемент (такое подмножество называется полным подмножеством членов совокупности ФАЛ). Проделав описанную процедуру, получим: $\{x_1\overline{x_2}x_{3f_1f_2}, x_1x_2\overline{x_3}_{f_1f_2}, x_1\overline{x_2}\cdot\overline{x_3}_{f_1}, x_1x_2x_{3f_2}\}$. Именно это подмножество используется при минимизации совокупности ФАЛ.

Итак, задача заключается в нахождении *минимальной совокупности дизъюнктивных нормальных форм* ФАЛ, под которой понимается та, где полное подмножество членов совокупности содержит минимальное количество букв, а каждая ФАЛ включает минимальное число дизъюнктивных членов. Иными словами, минимальная совокупность должна иметь наименьший возможный суммарный ранг. Следует отметить, что форма представления каждой отдельной ФАЛ, входящей в минимальную совокупность, может быть отлична от минимальной для данной отдельной ФАЛ.

Выпишем и пронумеруем все минтермы с их признаками:

$$1-\ x_1\overline{x_2}x_{3f_1f_2};\ 2-\ x_1x_2\overline{x_3}_{f_1f_2};\ 3-\ x_1\overline{x_2}\cdot\overline{x_3}_{f_1};\ 4-\ x_1x_2x_{3f_2}.$$

Далее, как и в методе Квайна, начинается процедура получения простых импликант. Для получения простых импликант проводятся все возможные склеивания членов подмножества. Полученным после склеивания произведениям приписывается признак, состоящий из общих букв, содержащихся в признаках обоих склеивающихся членов. Если ни одна буква в признаке склеиваемых членов не совпадает, то произведению признак не присваивается:

$$1,3-\ x_1\overline{x_2}_{f_1};\ 1,4-\ x_1x_{3f_2};\ 2,3-\ x_1\overline{x_3}_{f_1};\ 2,4-\ x_1x_{2f_2}.$$

После проведения склеиваний выполняются поглощения, причем операции поглощения можно проводить только между членами с одинаковыми признаками. Минтермы, которые не поглощаются ни одним произведением, являются *простыми импликантами совокупности функций*.

Поглощенными оказываются $x_1\overline{x_2}\cdot\overline{x_3}_{f_1}$ и $x_1x_2x_{3f_2}$. Минтермы $x_1\overline{x_2}x_{3f_1f_2}$ и $x_1x_2\overline{x_3}_{f_1f_2}$ являются простыми импликантами.

Для получения всех простых импликант заданной совокупности функций операции склеивания выполняются над произведениями, полученными в результате склеивания минтермов. При этом произведения, не содержащие признака, в склеивании не участвуют.

Дальнейшее склеивание дает $1,3,2,4-x_1$; $1,4,2,3-x_1$.

Поскольку последние импликанты не имеют признака, то они исключаются.

В нашем примере первый этап на этом завершается, но в общем случае склеивания и поглощения повторяются, пока очередной цикл склеивания станет невозможным.

Таким образом, простыми импликантами совокупности будут: $x_1\overline{x_2}x_{3\,f_1f_2}$, $x_1x_2\overline{x_3}_{\,f_1f_2}$, $x_1\overline{x_2}_{\,f_1}$, $x_1x_{3\,f_2}$, $x_1\overline{x_3}_{\,f_1}$, $x_1x_{2\,f_2}$.

Для нахождения минимальной совокупности, как и в методе Квайна, используется импликантная матрица. В заголовки столбцов матрицы записываются все минтермы, а в горизонтальные входы — все простые импликанты совокупности функций. Каждому минтерму приписывается признак, указывающий, в какие функции этот минтерм входит.

	$x_1\overline{x_2}x_3$		$x_1x_2\overline{x_3}$		$x_1\overline{x_2}\cdot\overline{x_3}$	$x_1x_2x_3$
	f_1	f_2	f_1	f_2	f_1	f_2
$x_1\overline{x_2}x_{3\,f_1f_2}$	√	√				
$x_1x_2\overline{x_3}_{\,f_1f_2}$			√	√		
$x_1\overline{x_2}_{\,f_1}$	√				√	
$x_1x_{3\,f_2}$		√				√
$x_1\overline{x_3}_{\,f_1}$			√		√	
$x_1x_{2\,f_2}$				√		√

Заключительный этап минимизации совокупности ФАЛ состоит в выборе подмножества импликант с минимальным числом букв, покрывающих все столбцы импликантной матрицы. Для рассматриваемого примера — это простые импликанты $x_1\overline{x_2}x_{3\,f_1f_2}$, $x_1\overline{x_3}_{\,f_1}$, $x_1x_{2\,f_2}$. Выделив для функции f_i импликанты с признаком f_i, получим искомую минимальную совокупность ФАЛ:

$$f_1 = x_1\overline{x_2}x_3 \vee x_1\overline{x_3};$$
$$f_2 = x_1\overline{x_2}x_3 \vee x_1x_2.$$

Учитывая, что первый член входит в оба выражения, при вычислении коэффициента сложности совокупности этот элемент нужно учитывать лишь однократно. Поэтому суммарный ранг совокупности равен 7, а коэффициент сложности — 10.

Если бы мы минимизировали каждую из ФАЛ совокупности отдельно, то получили бы систему:

$$f_1 = x_1\overline{x_2} \vee x_1\overline{x_3};$$
$$f_2 = x_1x_3 \vee x_1x_2,$$

в которой ранги первых членов выражений меньше, однако, суммарный ранг системы равен 8, а коэффициент сложности — 12, то есть больше, чем при минимизации совокупности ФАЛ.

Следует заметить, что если упростить хотя бы одну ФАЛ, входящую в минимальную совокупность, то количество букв в полном подмножестве увеличится.

Приложение В

Схемотехнические основы вычислительных машин

Сигналы в цифровой схемотехнике

Цифровой сигнал — это сигнал, который может принимать два значения, рассматриваемые как логическая «1» и логический «0». Устройства, работающие только с цифровыми сигналами, называются *цифровыми устройствами*. При описании цифровых сигналов используются определенные термины (рис. В.1):

- активный уровень сигнала — уровень, порождающий выполнение устройством соответствующей функции;
- пассивный уровень сигнала — уровень, при котором устройство не выполняет никакой функции;
- положительный сигнал (сигнал положительной полярности) — сигнал, активный уровень которого — логическая «1» («0» соответствует отсутствию сигнала);
- отрицательный сигнал (сигнал отрицательной полярности) — сигнал, активный уровень которого — логический «0» («1» соответствует отсутствию сигнала);

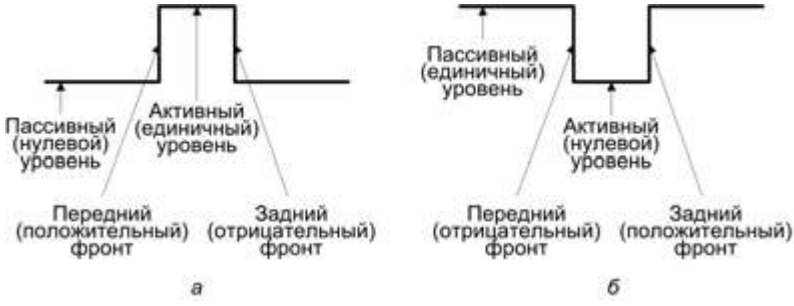


Рис. В.1. Основные параметры цифровых сигналов: *а* — в положительной логике; *б* — в отрицательной логике

- передний фронт сигнала — переход сигнала из пассивного уровня в активный;
- задний фронт сигнала — переход сигнала из активного уровня в пассивный;
- положительный фронт сигнала — переход сигнала из «0» в «1»;
- отрицательный фронт сигнала — переход сигнала из «1» в «0»;
- тактовый сигнал (строб) — управляющий сигнал, определяющий момент выполнения элементом или узлом его функции.

Логические элементы

Между аналитической формой представления булевых функций и их схемной реализацией существует взаимно однозначное соответствие: каждой элементарной ФАЛ соответствует схемный аналог. Электронные схемы, выполняющие простейшие логические операции, называются *логическими элементами*. Для реализации разнообразных логических функций достаточно иметь логические элементы, обеспечивающие минимальный логический базис.

Основные обозначения на схемах

В настоящее время для обозначения логических элементов используются несколько стандартов. Наиболее распространенными являются международный (IEC), российский (ГОСТ), американский (ANSI) и европейский (DIN). ГОСТ на уровне логических элементов практически идентичен международному стандарту. Здесь логические элементы изображаются в виде прямоугольников с соответствующими надписями внутри. В двух последних стандартах каждый вид логического элемента представляется особым символом. Отметим, что стандарт DIN фактически является стандартом, принятым в Германии, однако иногда он используется и в других европейских странах, хотя широкого распространения пока не получил.

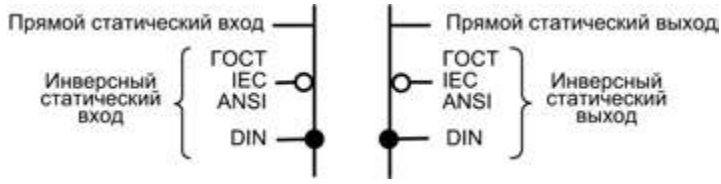


Рис. В.2. Типовое изображение входов и выходов на условном обозначении логических элементов

Возможное обозначение входов и выходов логических элементов показано на рис. В.2.

Логический элемент «НЕ»

Логический элемент «НЕ» (инвертор) имеет всего один вход и один выход. Выходной сигнал инвертора принимает всегда противоположное значение по отношению к значениям входного сигнала. Схема «НЕ» реализует операцию $F = \overline{x}$ (читается как «не x») и на электрических схемах изображается в одном из вариантов, приведенных на рис. В.3.

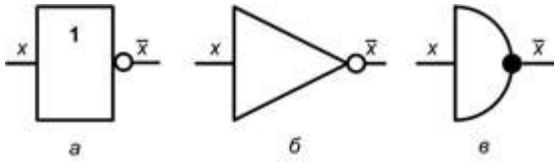


Рис. В.3. Условные обозначения логического элемента «НЕ» в стандартах:
а — ГОСТ и IEC; *б* — ANSI; *в* — DIN

Кружок служит указателем инверсии[1].

Логический элемент «И»

Логические элементы «И» реализуют функцию конъюнкции (логического умножения). Минимальное число входов равно двум. В общем случае такие элементы описываются логическим выражением вида $f = x_1 \wedge x_2 \wedge ... \wedge x_n$ и изображаются, как показано на рис. В.4.

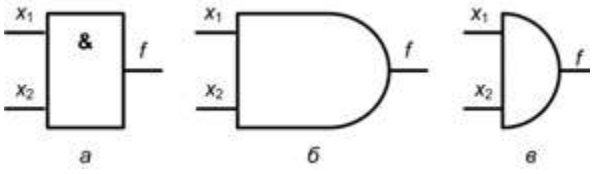


Рис. В.4. Условные обозначения логического элемента «И» в стандартах:
а — ГОСТ и IEC; *б* — ANSI; *в* — DIN

Сигнал на выходе логического элемента «И» соответствует логической единице, когда на все n входов ($n \geq 2$) поданы сигналы логической единицы. По этой причине такие элементы называют схемами совпадения. Иногда используется еще одно название — «конъюнкторы».

Переместительный и сочетательный законы булевой алгебры предопределяют возможность построения схемы «И» на большое число входов с использованием конъюнкторов, имеющих меньшее количество входов. Так, 6-входовую схему «И» можно описать выражением $f = x_1 \wedge x_2 \wedge x_3 \wedge x_4 \wedge x_5 \wedge x_6$ и реализовать на базе двухвходовых схем «И», если согласно упомянутым законам представить данное выражение в виде $f = ((((x_1 \wedge x_2) \wedge x_3) \wedge x_4) \wedge x_5) \wedge x_6$ либо $f = ((x_1 \wedge x_2) \wedge (x_3 \wedge x_4)) \wedge (x_5 \wedge x_6)$. Соответственно получаем два варианта схемы (рис. В.5).

Логически оба варианта эквивалентны. Следует, однако, учитывать, что сигнал на выходе логической схемы появляется с некоторой задержкой по отношению к моменту подачи входных сигналов. По этой причине второй вариант (см. рис. В.5, *б*) предпочтителен в плане быстродействия, так как в нем входные сигналы проходят через меньшее число элементов.

[1] ГОСТ разрешает и иные варианты размещения указателя инверсии. Здесь же будем придерживаться варианта, совпадающего с международном стандартом IEC.

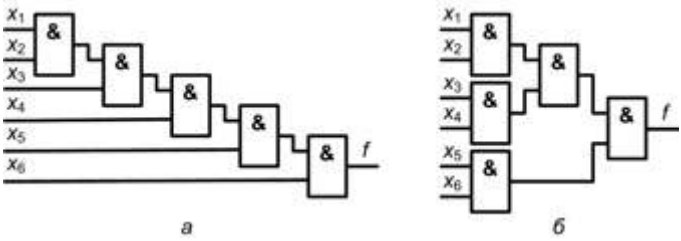


Рис. В.5. Варианты построения 6-входовой схемы «И» на базе двухвходовых схем: *а* — каскадный; *б* — параллельный

Логический элемент «ИЛИ»

Логический элемент «ИЛИ» — это электронная логическая схема, выходной сигнал которой соответствует логическому нулю, когда логическому нулю равны сигналы на всех входах схемы. Схема обеспечивает логическое сложение входных сигналов ($f = x_1 \vee x_2 \vee ... \vee x_n$).

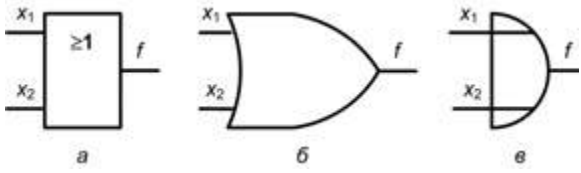


Рис. В.6. Условные обозначения логического элемента «ИЛИ» в стандартах: *а* — ГОСТ и IEC; *б* — ANSI; *в* — DIN

Возможные графические обозначения показаны на рис. В.6.

Логический элемент «И-НЕ»

Электронная логическая схема, в которой выходной сигнал соответствует логическому «0», когда сигналы на всех входах равны логической «1». Схема реализует инверсию логического произведения всех входных сигналов, то есть логическую операцию $f = \overline{x_1 \wedge x_2 \wedge ... \wedge x_n}$. Условные графические обозначения логического элемента «И-НЕ» показаны на рис. В.7.

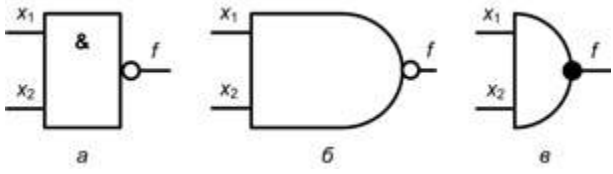


Рис. В.7. Условные обозначения логического элемента «И-НЕ»: *а* — ГОСТ и IEC; *б* — ANSI; *в* — DIN

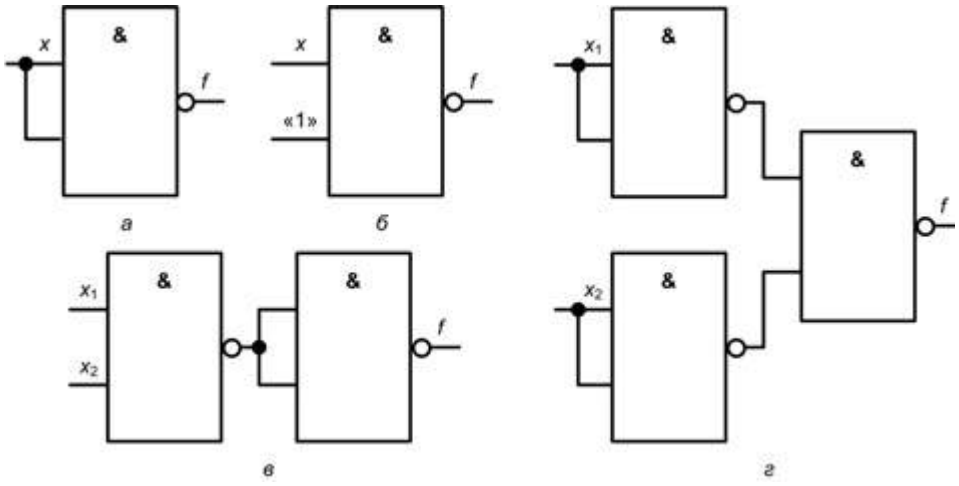


Рис. В.8. Реализация на базе логического элемента «И-НЕ» функций: *а, б* — «НЕ»; *в* — «И»; *г* — «ИЛИ»

Имея элемент «И-НЕ», можно реализовать элементы «НЕ», «И», «ИЛИ», как это показано на рис. В.8.

Элемент «ИЛИ-НЕ»

Электронная логическая схема, в которой выходной сигнал соответствует логической «1», когда сигналы на всех входах равны логическому «0». Схема реализует инверсию логической суммы, то есть логическую операцию «ИЛИ-НЕ» ($f = \overline{x_1 \vee x_2 \vee ... \vee x_n}$), и обозначается, как показано на рис. В.9.

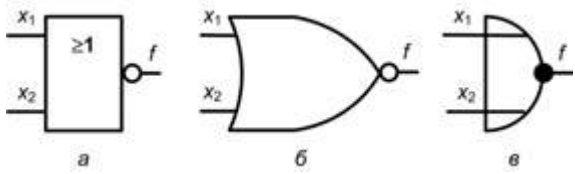


Рис. В.9. Условные обозначения логического элемента «ИЛИ-НЕ»: *а* — ГОСТ и IEC; *б* — ANSI; *в* — DIN

Элемент «ИЛИ-НЕ» можно трансформировать в элементы «НЕ», «И», «ИЛИ», как это показано на рис. В.10.

Набор логических элементов, реализующий операции того или иного минимального базиса, называется *минимальным элементным базисом*. В современной микроэлектронике таким базисом служат элементы «И-НЕ» либо «ИЛИ-НЕ».

Использование только элементов минимального базиса часто приводит к увеличению сложности устройств и ухудшает их основные эксплуатационные параметры. Поэтому во многих случаях используются расширенные (избыточные) элементы

базиса, в которых кроме элементов «И-НЕ», «ИЛИ-НЕ» используются схемы, выполняющие функции «И-ИЛИ-НЕ», «И», «ИЛИ», «Исключающее ИЛИ».

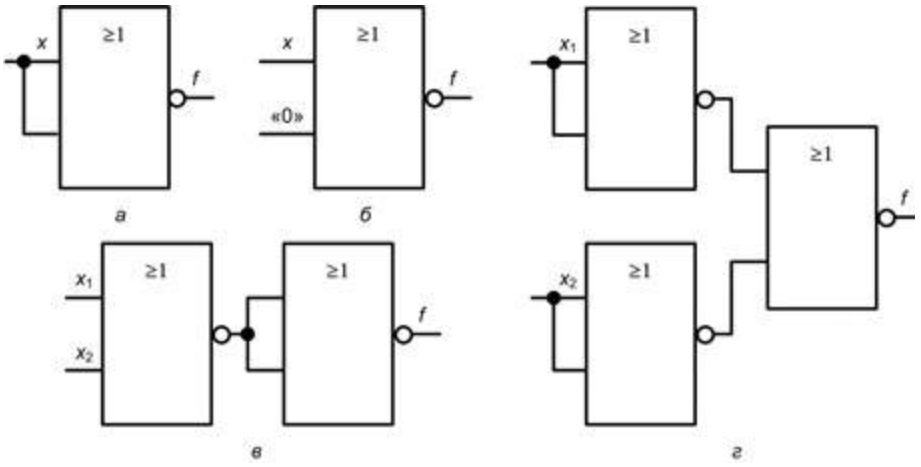


Рис. В.10. Реализация на базе логического элемента «ИЛИ-НЕ» функций: *а, б* — «НЕ»; *в* — «ИЛИ»; *г* — «И»

Логический элемент «Исключающее ИЛИ»

В общем случае речь идет о схеме, на выходе которой сигнал соответствует логической единице, когда значение логической единицы имеет нечетное число аргументов. Под логическим элементом «Исключающее ИЛИ» понимают схему с двумя входами. Схемы с большим числом входов обычно рассматриваются как самостоятельные логические устройства, известные под названием «схем контроля четности». Приведенному описанию соответствует логическая функция «неравнозначность». Функция равносильна операции сложения по модулю 2. Для двух аргументов она описывается выражением $f = x_1 \oplus x_2 = x_1 \wedge \overline{x_2} \vee \overline{x_1} \wedge x_2$. Графические изображения логического элемента «Исключающее ИЛИ» показаны на рис. В.11.

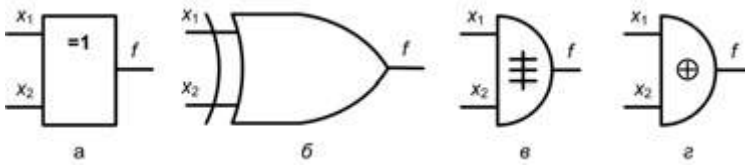


Рис. В.11. Условные обозначения логического элемента «Исключающее ИЛИ» в стандартах: *а* — ГОСТ и IEC; *б* — ANSI; *в*, *г* — DIN

Логический элемент «Эквивалентность»

В некоторых микросхемах встречается логический элемент, реализующий функцию эквивалентности или логической равнозначности ($f = \overline{x_1 \oplus x_2} = x_1 \wedge x_2 \vee \overline{x_1} \wedge \overline{x_2}$).

Условные графические обозначения подобных логических элементов показаны на рис. В.12.

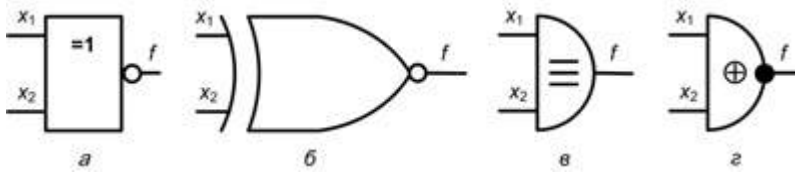


Рис. В.12. Условные обозначения логического элемента «Исключающее ИЛИ» с инверсным выходом в стандартах: *а* — ГОСТ и IEC; *б* — ANSI; *в*, *г* — DIN

Смысловое значение этой функции в том, что она принимает значение логической «1» при четном и значение логического «0» при нечетном числе единичных значений ее аргументов. В многовходовом варианте ($n > 2$) реализующие ее схемы получили название «схем контроля нечетности».

Положительная и отрицательная логика

Напряжения на входах и выходах логических элементов могут принимать два уровня: высокий (H — high) и низкий (L — low). Если высокий уровень соответствует логической «1», а низкий уровень — логическому «0», то принято считать, что логический элемент работает с *положительной логикой*. Если высокий уровень соответствует логическому «0», а низкий уровень — логической «1», то элемент работает с *отрицательной логикой*.

Пусть имеется таблица истинности логического элемента «ИЛИ».

x_1	x_2	f
0	0	0
0	1	1
1	0	1
1	1	1

Для элемента с положительной логикой эту таблицу можно переписать в следующем виде:

x_1	x_2	f
L	L	L
L	H	H
H	L	H
H	H	H

Однако этот же элемент реализует также и определенную логическую функцию для отрицательной логики. Для отрицательной логики таблицу истинности логического элемента можно переписать в виде:

x_1	x_2	f_1
1	1	1
1	0	0
0	1	0
0	0	0

Эта таблица истинности соответствует логическому элементу «И». Докажем это алгебраически: $f = x_1 \vee x_2$. Перейдем от положительной логики к отрицательной: $f_1 = \overline{f} = \overline{x_1} \vee \overline{x_2} = \overline{x_1 \wedge x_2} = x_1 \wedge x_2$. Таким образом, один и тот же элемент для положительной логики является элементом «ИЛИ», а для отрицательной логики — элементом «И». Очевидно, что элемент, являющийся элементом «И» для положительной логики, будет являться элементом «ИЛИ» — для отрицательной логики. Аналогично элемент «И-НЕ» трансформируется в «ИЛИ-НЕ», а «ИЛИ-НЕ» — в «И-НЕ». Таким образом, логическую функцию элемента можно изменить, не затрагивая его структуры, а лишь поменяв логику сигналов на входах и выходах.

Чтобы явно указать использование отрицательной логики, несколько видоизменяют условные графические изображения логических элементов (рис. В.13).

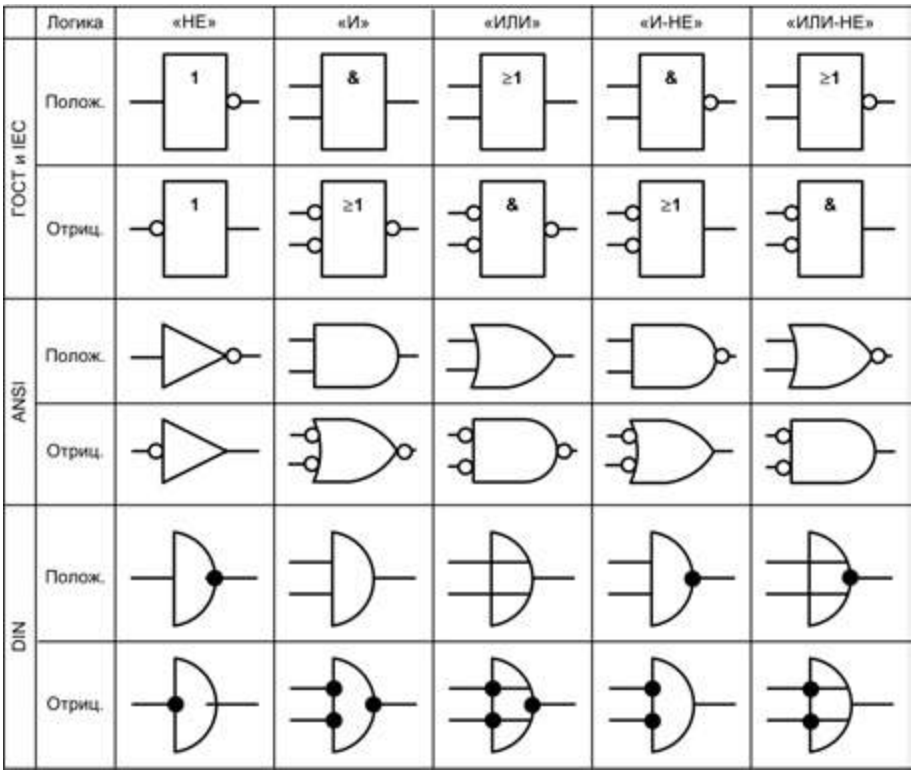


Рис. В.13. Условные графические изображения логических элементов в положительной и отрицательной логике

Элементы памяти

Состояние выходных сигналов рассмотренных логических элементов определяется лишь текущим состоянием входов и не зависит от предыдущего состояния схемы. Иными словами, логические элементы не обладают свойством памяти и не могут быть использованы для запоминания информации. Функцию элемента памяти в вычислительной техники выполняют *последовательностные логические устройства*, в которых выходной сигнал зависит не только от текущих входных логических значений, но и от тех, которые действовали на входе в предыдущие моменты времени.

Простейшая последовательностная схема состоит из пары схем «НЕ», охваченных петлей обратной связи (рис. В.14).

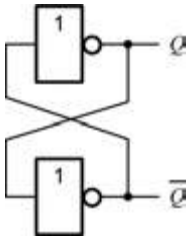


Рис. В.14. Простейший бистабильный элемент

Схему часто называют *бистабильной*, поскольку у нее есть два устойчивых состояния. Так как элемент не имеет свободных входов, им невозможно управлять и изменять его состояние. В реальных последовательных устройствах бистабильный элемент строится не на схемах «НЕ», а на схемах «И-НЕ» либо «ИЛИ-НЕ». Появляющиеся при этом дополнительные входы позволяют реализовать логику изменения состояния элемента. Простейшим последовательностным элементом является триггер.

Триггеры

Триггер представляет собой устройство, которое находится в одном из двух устойчивых состояний и скачкообразно переходит из одного состояния в другое под воздействием внешнего управляющего сигнала. Триггер может служить элементом памяти, способным хранить 1 бит информации. В основе любого триггера лежит схема из двух логических элементов, которые охвачены положительными обратными связями (см. рис. В.14). Триггеры являются основными составными элементами в большинстве последовательностных устройств.

Выходы триггеров

Основной выход триггера, определяющий его состояние, обозначают Q. Обычно у триггеров есть и инверсный выход — $\overline{Q}$. В том случае, когда в триггере хранится

логическая 1 ($Q = 1$), говорят, что триггер установлен. Если триггер хранит логический 0 ($Q = 0$), считается, что триггер сброшен.

Для описания работы триггера аналогично комбинационным схемам могут быть использованы таблицы истинности или логические выражения. Особенностью такого описания является использование в качестве дополнительной входной переменной предыдущего значения выходного сигнала триггера.

Входы триггеров

Состояние триггера может меняться только при воздействии внешних сигналов. Для обеспечения возможности изменения состояния триггера организуют цепи записи информации. В настоящее время разработано большое количество типов триггеров, которые различаются способами записи информации. Способ записи можно определить по обозначению информационных входов:

- *R* (Reset — сброс) — вход установки триггера в нулевое состояние ($Q = 0$);
- *S* (Set — установка) — вход установки триггера в единичное состояние ($Q = 1$);
- *K* (Kill — аннулирование) — вход сброса универсального триггера ($Q = 0$);
- *J* (Jerk — толчок) — вход установки универсального триггера ($Q = 1$);
- *T* (Toggle — переключатель) — вход триггера, по которому он меняет свое текущее состояние на противоположное;
- *D* (Delay — задержка) — информационный вход переключения триггера в состояние, соответствующее логическому уровню на этом входе;
- *C* (Clock — такт) — управляющий или синхронизирующий вход.

Кроме этих основных входов некоторые триггеры могут снабжаться входом *V*. Вход *V* блокирует работу триггера, при этом триггер продолжает сохранять ранее записанную в него информацию.

Входы триггеров могут быть прямыми (статическими или динамическими) и инверсными (статическими или динамическими). *Статические входы* реагируют на уровень входного сигнала, а *динамические* — на его перепад. Типовое изображение статических входов на условном обозначении логических элементов приводилось на рис. В.2. Изображение динамических входов в разных стандартах практически совпадает и показано на рис. В.15.

Рис. В.15. Типовое изображение динамических входов на условном обозначении триггеров

Классификация триггеров

Триггеры классифицируют по логическому функционированию и способу записи информации.

По логическому функционированию различают *RS*-, *JK*-, *D*-, *DV*-, *T*- и *TV*-триггеры. Название типа триггера отражает вид используемых входов. Кроме того, используются комбинированные триггеры, где совмещается одновременно несколько типов, и триггеры со сложной логикой, в которых группы входов связаны между собой логическими зависимостями.

При классификации *по способу записи информации* выделяют несколько признаков.

Первый признак — момент реакции на входной сигнал. Здесь различают *асинхронные* (нетактируемые) и *синхронные* (тактируемые) триггеры. Асинхронный триггер изменяет свое состояние непосредственно в момент изменения сигнала на его информационных входах, то есть его реакция на изменение входного сигнала подобна реакции комбинационного элемента. Синхронный триггер изменяет свое состояние лишь в строго определенные моменты времени, соответствующие действию на его синхронизирующем входе *C* активного сигнала. При пассивном значении сигнала *C* триггер на изменение информационных сигналов не реагирует.

Вторым признаком служит способ восприятия тактовых сигналов. По этому признаку различают триггеры, *управляемые уровнем* (со статическим управлением, или стробируемые) и *управляемые фронтом* (с динамическим управлением, или тактируемые). Управление уровнем означает, что при одном уровне тактового сигнала триггер «прозрачен» — воспринимает входные сигналы и реагирует на них, а при другом уровне — остается в неизменном состоянии, не реагируя на входные сигналы. При управлении фронтом разрешение на переключение триггера дается только в момент перепада тактового сигнала (на фронте или спаде). В остальное время триггер не воспринимает входные сигналы и остается в неизменном состоянии.

Наконец, третий признак — характер процесса переключения триггера. Здесь различают одноступенчатые и двухступенчатые триггеры. В одноступенчатом триггере переключение в новое состояние происходит сразу, а в двухступенчатом — по этапам. Двухступенчатые триггеры содержат два триггера — входной и выходной. На первом этапе информация заносится во входной триггер, а на втором этапе — переносится в выходной. Подробнее двухступенчатые триггеры рассматриваются ниже.

RS-триггеры

RS-триггер — это элемент с двумя входами (*S* и *R*) и двумя выходами — прямым (*Q*) и инверсным ($\overline{Q}$). Если на вход *S* такого триггера подать логический сигнал «1» (на входе $R = 0$), то выходной сигнал *Q* примет значение «1». Если подать «1» на вход *R* (на входе $S = 0$), выходной сигнал *Q* примет значение «0». При $Q = 1$ и $\overline{Q} = 0$ считается, что триггер находится в единичном состоянии, а при $Q = 0$ и $\overline{Q} = 1$ — в нулевом. Одновременная подача сигналов $R = 1$ и $S = 1$ запрещена, поскольку в этом случае устройство утрачивает свойства триггера (на выходах *Q* и $\overline{Q}$ будут одинаковые значения, что невозможно по определению).

Асинхронный RS-триггер снабжен только двумя информационными входами: входом сброса *R* и входом установки *S*. По сути это простейший элемент памяти,

который может быть реализован на элементах «ИЛИ-НЕ» либо «И-НЕ». Условное обозначение триггера и временная диаграмма приведены на рис. В.16.

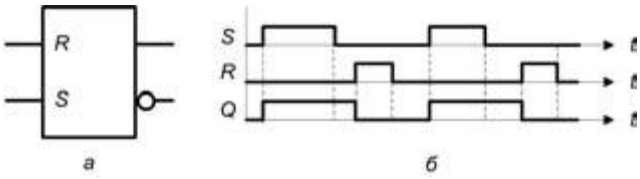


Рис. В.16. Асинхронный *RS*-триггер: *а* — условное графическое обозначение; *б* — временная диаграмма работы

Основной недостаток асинхронных триггеров — незащищенность перед опасными состязаниями сигналов, когда сигналы, поступающие на разные информационные входы триггера, проходят по разным цепям через различное число элементов. Из-за этого между сигналами возможны временные сдвиги, что может привести к ложным срабатываниям триггеров.

Синхронный RS-триггер срабатывает лишь при наличии дополнительного сигнала синхронизации на входе С. Срабатывание триггера может происходить по уровню сигнала (статическое управление) либо по фронту или срезу сигнала (динамическое управление). Условное графическое обозначение и временная диаграмма для синхронного RS-триггера приведены на рис. В.17.

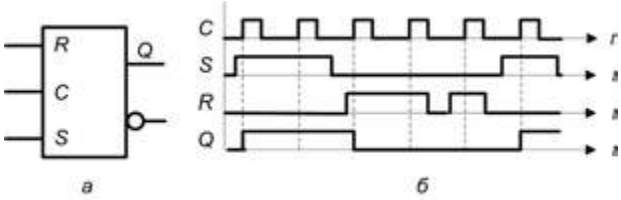


Рис. В.17. Синхронный *RS*-триггер: *а* — условное графическое обозначение; *б* — временная диаграмма работы

Синхронный триггер является более помехозащищенным, чем асинхронный, однако полностью исключить возможность возникновения на входах недопустимой кодовой комбинации данный триггер не позволяет.

JK-триггеры

JK-триггер по своей структуре сложнее *RS*-триггера. Прежде всего, это всегда синхронный (тактируемый) триггер. Как и в *RS*-триггере, *JK*-триггер имеет отдельные входы установки *J* (аналог *S*) и сброса *K* (аналог *R*). Если $J = 1$ и $K = 0$, то тактовым импульсом можно добиться переключения из 0 в 1. Если $K = 1$, $J = 0$, то тактовым импульсом триггер переключается из 1 в 0. В отличие от *RS*-триггера комбинация $J = K = 1$ не просто разрешена, но специально предусмотрена. При этой комбинации входных сигналов триггер превращается в *T*-триггер, переключаясь по каждому импульсу на входе *C*. *JK*-триггер часто называют универсальным триггером.

Реальные микросхемы строятся в виде триггеров с инверсным динамическим управлением, то есть все переключения в триггере происходят по заднему фронту тактирующего импульса. Графическое обозначение *JK*-триггера с динамическим управлением и временные диаграммы в режиме *T*-триггера приведены на рис. В.18.

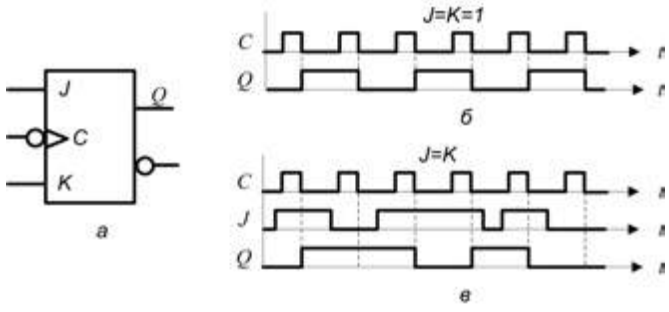


Рис. В.18. *JK*-триггер с динамическим управлением: *а* — условное графическое обозначение; *б* — временная диаграмма работы в асинхронном режиме *T*-триггера; *в* — временная диаграмма работы в синхронном режиме *T*-триггера

D-триггеры

Одним из наиболее распространенных видов триггеров является D-триггер. Он имеет один информационный вход D и один тактирующий вход C. По сигналу синхронизации в D-триггер переписывается информация, которая в данный момент присутствует на входе D. По определению такой триггер может быть только синхронным. Информация на выходе остается неизменной вплоть до прихода следующего импульса синхронизации. Как правило, в интегральном исполнении выпускаются D-триггеры с динамическим управлением, в которых перезапись информации с входа D происходит по фронту тактирующего сигнала. На рис. В.19 приведено условное обозначение и временная диаграмма работы *D*-триггера.

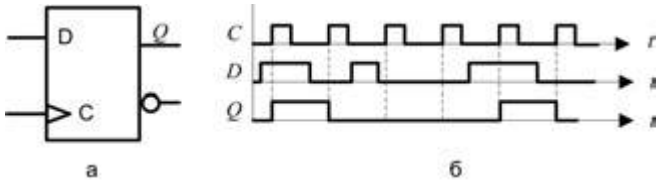


Рис. В.19. *D*-триггер с динамическим управлением: *а* — условное графическое обозначение; *б* — временная диаграмма работы

DV-триггеры

DV-триггер представляет собой модификацию *D*-триггера (рис. В.20). В *D*-триггере записанная информация не может храниться более одного периода синхронизации. *DV*-триггер при $V = 1$ функционирует как обычный *D*-триггер, а при $V = 0$ — переходит в режим хранения информации независимо от смены сигналов на входе *D*.

Наличие входа *V* расширяет функциональные возможности *D*-триггера, позволяя в нужные моменты сохранять информацию на выходах в течение требуемого числа тактовых периодов.

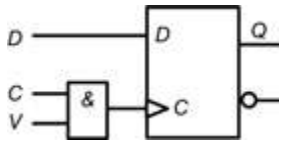


Рис. В.20. Логическая схема *DV*-триггера

Т-триггеры

При построении счетчиков возникает необходимость в триггере, меняющем свое состояние с каждым сигналом на входе Т. Т-триггер, или счетный триггер, имеет один информационный вход Т. Диаграммы функционирования Т-триггера в асинхронном и синхронном режимах приводились при рассмотрении JK-триггеров.

Несколько вариантов построения *T*-триггера показаны на рис. В.21.

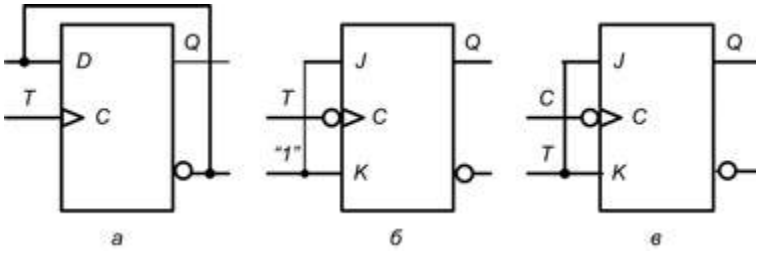


Рис. В.21. Варианты реализации *T*-триггеров: *а* — асинхронный на базе *D*-триггера; *б* — асинхронный на базе *JK*-триггера; *в* — синхронный на базе *JK*-триггера

TV-триггеры

Триггер *TV*-типа кроме счетного входа *T* имеет управляющий вход *V* для разрешения приема информации. Асинхронные и синхронные *TV*-триггеры могут быть получены на базе *JK*-триггера. Ниже показаны схемы асинхронного (рис. В.22, *а*) и синхронного (рис. В.22, *б*) *TV*-триггеров.

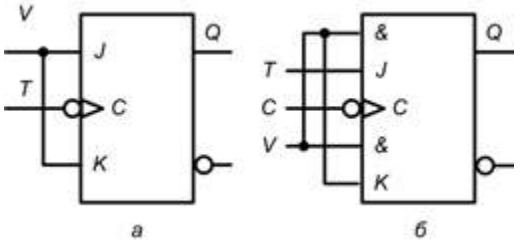


Рис. В.22. Варианты реализации *TV*-триггеров: а — асинхронного; б — синхронного

Двухступенчатые триггеры

При построении схем, представляющих собой цепочку взаимосвязанных триггеров, например регистров сдвига, выходные сигналы предшествующего триггера являются входными сигналами для последующего триггера. В этих условиях необходимо, чтобы значения выходных сигналов триггера на то время, пока производится их запись в другой триггер, не изменялись. Данная проблема решается с помощью двухступенчатых триггеров.

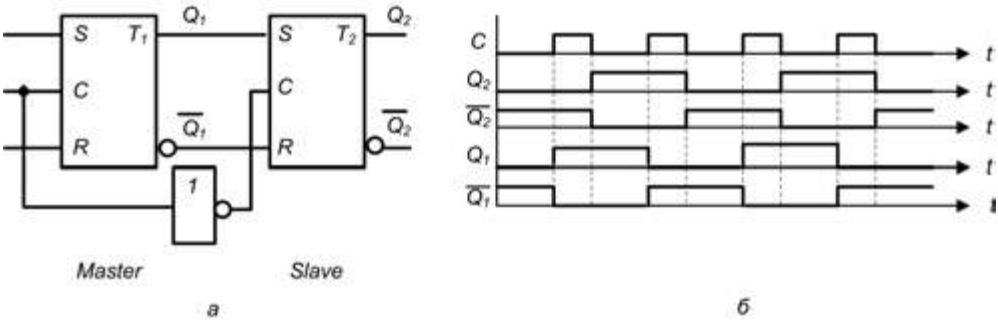


Рис. В.23. Двухступенчатый триггер: *а* — логическая схема; *б* — диаграмма работы

Двухступенчатый триггер состоит из двух последовательно соединенных ступеней, причем каждая ступень содержит по синхронному *RS*-триггеру (рис. В.23, *а*). Первая ступень — ведущая, или М-секция (М происходит от английского Master — хозяин) принимает информацию с входных линий *S* и *R*. Состояние выходов ведущей ступени подается на вторую ступень — ведомую, или *S*-секцию (*S* происходит от английского Slave — раб).

Состояние выходов ведущего триггера изменяется в момент появления положительного импульса синхронизации, и эти изменения будут переданы на входы ведомого триггера. Однако никакие изменения на выходе ведомого триггера не будут происходить до тех пор, пока не появится положительный сигнал инвертированного импульса синхронизации, то есть отрицательный (задний) фронт исходного синхроимпульса. Следовательно, изменения на выходах Q_2 и $\overline{Q_2}$ не произойдут до тех пор, пока не завершится импульс синхронизации. На рис. В.23, *б* показаны временные диаграммы работы триггера.

Двухступенчатый триггер в стандарте ГОСТ обычно обозначают двумя буквами *ТТ*.

Приложение Г

Синтез и анализ комбинационных схем

Комбинационными схемами называются схемные реализации логических функций любого вида, логическое состояние выходов которых зависит только от комбинации логических сигналов на входах в данный момент времени. Таким образом, к комбинационным относят схемы, не обладающие свойством памяти.

Обычно комбинационные схемы реализуются на основе ДНФ и КНФ. В качестве базиса может быть использован любой функционально полный базис, но для удобства изложения мы будем в дальнейшем ориентироваться на булев базис.

При работе с цифровыми устройствами обычно решают две задачи: создание логической схемы по аналитическому описанию соответствующей ФАЛ; получение аналитического описания ФАЛ по логической схеме устройства. Первая задача носит название *задачи синтеза*, вторая — *задачи анализа*.

Синтез комбинационных схем

Обычно исходным пунктом для синтеза служит описание ФАЛ в виде минимальной ДНФ или минимальной КНФ.

Синтез логических устройств в булевом базисе

Булевым называют базис, образуемый элементами «И», «ИЛИ», «НЕ».

При схемной реализации ДНФ каждой элементарной конъюнкции соответствует элемент «И», число входов которого определяется количеством переменных в данной конъюнкции. Все выходы элементов «И» объединяются на элементе «ИЛИ», причем число входов этого элемента равно числу элементарных конъюнкций в ДНФ.

При схемной реализации КНФ каждой элементарной дизъюнкции соответствует элемент «ИЛИ», число входов которого определяется количеством переменных в данной дизъюнкции. Все выходы элементов «ИЛИ» объединяются на элементе «И», причем число входов этого элемента равно числу элементарных дизъюнкций в КНФ.

Инверсия переменных реализуется путем подключения логических элементов «НЕ» на те входы логических элементов «И» («ИЛИ»), куда должны быть поданы переменные с инверсией.

Таким образом, задача синтеза сводится к построению логической схемы, реализующей заданную функцию алгебры логики. Для построения логической схемы необходимо элементы, реализующие логические операции, указанные в ФАЛ, располагать в порядке, заданном булевым выражением. В качестве примера рассмотрим функцию четырех переменных, приведенную в табл. Б.7. Напомним, что минимальная ДНФ для этой функции имеет вид $f = \overline{x_1}x_2x_4 \vee \overline{x_2} \cdot \overline{x_3} \vee x_3\overline{x_4}$. Логическую схему будем строить в базисе «И», «ИЛИ», «НЕ». Из выражения видно, что понадобятся 4 схемы «НЕ» (для получения инверсных значений аргументов), одна трехвходовая схема «И» (для первой конъюнкции), две двухвходовых схемы «И» (для получения второй и третьей конъюнкций) и одна трехвходовая схема «ИЛИ». В соответствии с МДНФ получаем логическую схему, приведенную на рис. Г.1.

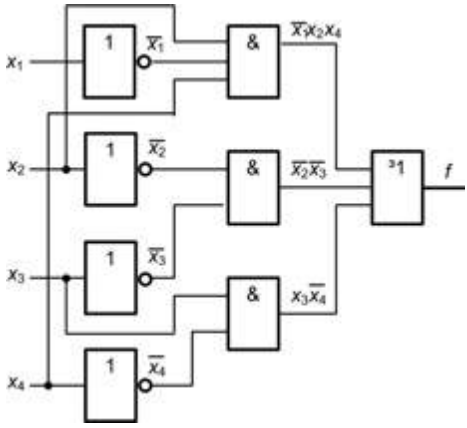


Рис. Г.1. Логическая схема устройства в базисе «И», «ИЛИ», «НЕ»

Синтез логических устройств в заданном базисе

Для сокращения номенклатуры используемых микросхем вместо булева базиса часто используют функционально полные наборы «И-НЕ» либо «ИЛИ-НЕ». Для этого ДНФ или КНФ записывают в заданном базисе.

Если задан базис «И-НЕ», то путем двойного инвертирования исходного выражения или его части (с последующим применением теоремы де Моргана) логическая функция приводится к виду, содержащему только операции логического умножения и инвертирования. В качестве примера синтезируем устройство, реализующее логическую функцию из предшествующего примера. Сначала переведем минимальную ДНФ в базис «И-НЕ».

$$f = \overline{x_1}x_2x_4 \vee \overline{x_2} \cdot \overline{x_3} \vee x_3\overline{x_4} = \overline{\overline{x_1}x_2x_4 \vee \overline{x_2} \cdot \overline{x_3} \vee x_3\overline{x_4}} = \overline{\overline{x_1}x_2x_4} \cdot \overline{\overline{x_2} \cdot \overline{x_3}} \cdot \overline{x_3\overline{x_4}}.$$

Теперь, используя полученное представление функции, изобразим логическую схему устройства (рис. Г.2). Приведем ее в стандарте ANSI.

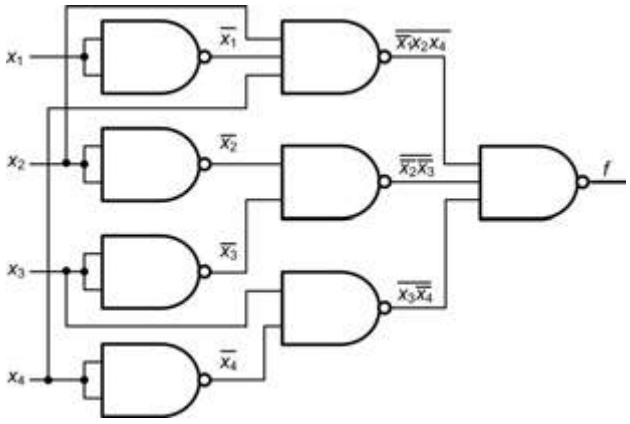


Рис. Г.2. Логическая схема устройства в базисе «И-НЕ»

Поступая аналогично, получим выражение в базисе «ИЛИ-НЕ»:

$$f = \overline{x_1}x_2x_4 \vee \overline{x_2}\cdot\overline{x_3} \vee x_3\overline{x_4} = \overline{\overline{\overline{x_1}x_2x_4 \vee \overline{x_2}\cdot\overline{x_3} \vee x_3\overline{x_4}}} =$$

$$= \overline{\overline{x_1 \vee \overline{x_2} \vee \overline{x_4}} \vee \overline{x_2 \vee x_3} \vee \overline{\overline{x_3} \vee x_4}}.$$

Соответствующая логическая схема, изображенная в стандарте DIN, показана на рис. Г.3.

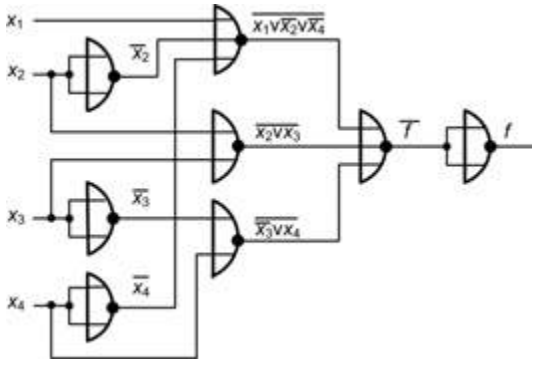


Рис. Г.3. Логическая схема устройства в базисе «ИЛИ-НЕ»

Анализ комбинационных схем

Задача анализа заключается в определении функции f, реализуемой заданной логической схемой. При решении данной задачи обычно придерживаются следующей последовательности действий.

1. Схема разбивается на ярусы. Ярусам присваиваются последовательные номера.
2. Выходы каждого логического элемента обозначаются названием искомой функции, снабженным цифровым индексом, где первая цифра — номер яруса, а остальные цифры — порядковый номер элемента в ярусе.
3. Для каждого элемента записывается аналитическое выражение, связывающее его выходную функцию с входными переменными. Выражение определяется логической функцией, реализуемой данным элементом.
4. Производится подстановка одних функций в другие, пока не получится булева функция, выраженная через входные переменные.

Используя приведенную последовательность действий, произведем анализ ранее синтезированной схемы (рис. Г.1). Сначала разобьем ее на ярусы. Пронумеровав получившиеся ярусы, введем обозначения для каждой выходной функции (рис. Г.4).

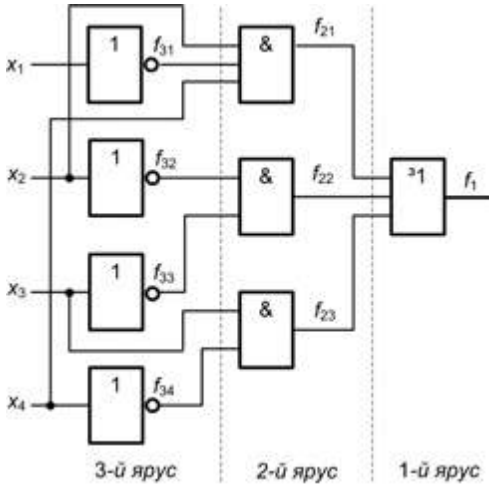


Рис. Г.4. Логическая схема устройства с разбивкой на ярусы

Запишем все функции, начиная с 1-го яруса:

$$f_1 = f_{21} \vee f_{22} \vee f_{23};$$
$$f_{21} = f_{31}x_2x_4, \quad f_{22} = f_{32}f_{33}, \quad f_{23} = x_3f_{34};$$
$$f_{31} = \overline{x_1}, \quad f_{32} = \overline{x_2}, \quad f_{33} = \overline{x_3}, \quad f_{34} = \overline{x_4}.$$

Теперь запишем все функции, подставляя входные переменные x_1, x_2, x_3 и x_4:

$$f_{21} = \overline{x_1}x_2x_4, \quad f_{22} = \overline{x_2} \cdot \overline{x_3}, \quad f_{23} = x_3\overline{x_4}.$$

В итоге получим выходную функцию:

$$f = f_1 = \overline{x_1}x_2x_4 \vee \overline{x_2} \cdot \overline{x_3} \vee x_3\overline{x_4}.$$

Список литературы

1. *Амамия М., Танака Ю.* Архитектура ЭВМ и искусственный интеллект. М.: Мир, 1993. 400 с.
2. *Антонов А. С.* Введение в параллельные вычисления. М.: НИВЦ МГУ, 2002. 69 с.
3. *Баранов С. И., Баркалов А. А.* Микропрограммирование: принципы, методы, применения. Зарубежная радиоэлектроника, 1984, № 5. С. 3–29.
4. *Буза М. К.* Архитектура компьютеров. Минск: Новое знание, 2006. 559 с.
5. *Воеводин В. В., Воеводин Вл. В.* Параллельные вычисления. СПб.: БХВ-Петербург, 2002. 608 с.
6. *Выхованец В. С.* Организация ЭВМ и систем. Тирасполь: РИО ПГУ, 2002. 193 с.
7. *Горнец Н. Н.* Организация ЭВМ и систем. М.: Издательский центр «Академия», 2008. 320 с.
8. *Гук М. Ю.* Шины PCI, USB и FireWire. СПб.: Питер, 2005. 540 с.
9. *Дубова Н.* Суперкомпьютеры nCube. Открытые системы, 1995, № 2. С. 42–47.
10. *Каган Б. М.* Электронные вычислительные машины и системы. М.: Энергоатомиздат, 1991. 592 с.
11. *Колосов В. Г., Мелехин В. Ф.* Проектирование узлов и систем автоматики и вычислительной техники. Л.: Энергоатомиздат, 1983. 256 с.
12. *Королев Л. Н.* Архитектура ЭВМ. М.: Научный мир, 2005. 272 с.
13. *Крайзмер Л. П., Бородаев Д. А., Гутенмахер Л. И., Кузьмин Б. Н., Смелянский И. Л.* Ассоциативные запоминающие устройства. Л.: Энергия, 1967.
14. *Крейгон Х.* Архитектура компьютеров и ее реализация. М.: Мир, 2004. 416 с.
15. *Майоров С. А., Новиков Г. И.* Структура электронных вычислительных машин. Л.: Машиностроение, 1979. 384 с.
16. *Максимов Н. В., Партыка Т. Л., Попов И. И.* Архитектура ЭВМ и вычислительных систем: учебник. М.: Форум, 2008. 512 с.
17. *Новиков Г. И., Павлов В. П.* Способ определения оптимального набора микроопераций и логических условий. УСИМ, 1979, № 4. С. 90–95.
18. *Опадчий Ю. Ф., Глудкин О. П., Гуров А. И.* Аналоговая и цифровая электроника. М.: Горячая линия — Телеком, 2002. 768 с.
19. *Орлов С. А.* Управляющие ЭВМ. М.: Изд-во МО, 1981. 241 с.

20. *Орлов С. А.* Организация и проектирование цифровых управляющих микроЭВМ и микроВС. М.: Изд-во МО, 1985. 475 с.
21. *Орлов С. А.* Основы организации микропроцессорных средств автоматизированных систем. Учебное пособие. М.: Изд-во МО, 1986. 207 с.
22. *Павлов Р. В.* Принципы организации многопроцессорных и многомашинных вычислительных систем. Рыбинск: РГАТА, 2002. 89 с.
23. *Палагин А. В., Иванов В. А., Кургаев А. Ф., Денисенко В. П.* МиниЭВМ: Принципы построения и проектирования. Киев: Наукова думка, 1975. 200 с.
24. *Панфилов И. В., Половко А. М.* Вычислительные системы. М.: Советское радио, 1990. 304 с.
25. *Пятибратов А. П., Гудыно Л. П., Кириченко А. А.* Вычислительные системы, сети и телекоммуникации. М.: Финансы и статистика, 1998. 400 с.
26. *Степанов А. Н.* Архитектура вычислительных систем и компьютерных сетей. СПб.: Питер, 2007. 509 с.
27. *Столлингс У.* Структурная организация и архитектура компьютерных систем, 5-е изд. М.: Изд. дом Вильямс, 2002. 896 с.
28. *Таненбаум Э.* Архитектура компьютера, 5-е изд. СПб.: Питер, 2007. 843 с.
29. *Хамахер К., Вранешич З., Заки С.* Организация ЭВМ, 5-е изд. СПб.: Питер, 2003. 848 с.
30. *Харкевич А. А.* Борьба с помехами. М.: Госиздат физико-математической литературы, 1963. 276 с.
31. *Хокни Р., Джессхоуп К.* Параллельные ЭВМ: Архитектура, программирование и алгоритмы. М.: Радио и связь. 1986. 392 с.
32. *Хорошевский В. Г.* Архитектура вычислительных систем. М.: Изд-во МГТУ им. Н.Э. Баумана, 2005. 512 с.
33. *Цилькер Б. Я., Макеев В. Я.* Архитектура вычислительных машин. Рига: TSI, 2000. 213 с.
34. *Цилькер Б. Я.* Архитектура вычислительных машин и систем. Рига: TSI, 2008. 208 с.
35. *Цилькер Б. Я., Орлов С. А.* Организация ЭВМ и систем. СПб.: Питер, 2004. 668 с.
36. *Цилькер Б. Я., Пятков В. П.* Архитектура вычислительных систем. Рига: TSI, 2001. 249 с.
37. *Abd-El-Bar, M.,* Design and Analysis of Reliable and Fault-Tolerant Computer Systems, Imperial College Press, 2007.
38. *Abd-El-Bar, M., El-Rewini, H.,* Fundamentals of Computer Organization and Architecture, John Wiley & Sons, 2005.
39. *Agarwal, A., Bianchini, R., Chaiken, D., Johnson, K. L., Kranz, D., Kubiatowicz, J., Lim, B-H., Mackenzie, K., Yeung, D.,* The MIT Alewife Machine: Architecture and Performance, Proceedings of the 22nd Annual International Symposium on Computer Architecture, Jun. 1995, pp. 2–13.
40. *Agerwala, T., Cocke, J.* High Performance Reduced Instruction Set Processors, Technical Report RC12434 (#55845), Yorktown, New York: IBM Thomas J. Watson Research Center, Jan. 1987.
41. *Almasi, G. S., Gottlieb, A.,* Highly Parallel Computing, 2nd Edition, Addison-Wesley, 1994.
42. *Altnether, J.,* Error Detecting and Correcting Codes, Intel Application Note AP-46, 1979.

43. *Amdahl G. M.,* Validity of the Single-Processor Approach to Achieving Large Scale Computing Capabilities, Proceedings AFIPS Conference, Vol. 30 (Atlantic City, New Jersey, Apr. 18–20), AFIPS Press, Reston, Va., 1967, pp. 483–485.
44. *Andrews, M.,* Principles of Firmware Engineering in Microprogram Control. Silver Spring, MD, Computer Science Press, 1980.
45. *Andrews, W.,* Futurebus+ Spec Completed, Almost Computer Design, Feb. 1, 1990?, pp. 22–28.
46. *Archibald, J. A.,* The Cache Coherence Problems in Shared-Memory Multi-processors, Technical Report, University of Washington, Feb. 1987.
47. *Archibald, J. A.,* Cache Coherence Approach for Large Multiprocessor Systems, Proceedings of the 2nd International Conference on Supercomputing, ACM, New York, 1988, pp. 337–345.
48. *Arvind, Bic, L., Ungerer, T.,* Evolution of Dataflow Computers. Advanced Topics in Data-Flow Computing (Gaudiot J-L., and Bic L., eds.), Prentice-Hall, 1991, pp. 3–33.
49. *Avizienis, A.,* Signed-digit number representations for fast parallel arithmetic, IRE Transactions on Electronic computers, EC-10, Sept 1961, pp. 389–400.
50. *Backus, J. W.,* Can Programming be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs, CACM, Vol. 21, 1978, pp. 613–641.
51. *Barendregt, H. P.,* The Lambda Calculus — Its Syntax and Semantics, 2nd Edition, North-Holland, 1984.
52. *Batcher, K. E.,* Sorting Networks and their Applications, Proceedings SJCC, 1968, pp. 307–314.
53. *Baugh, C. R., Wooley, B. A.,* A Two's Complement Parallel Array Multiplication Algorithm, IEEE Transactions on Computers, C-22, Dec. 1973, pp. 1045–1047.
54. *Benes, V. E.,* Mathematical Theory of Connecting Networks and Telephone Traffic, Academic Press, 1965, 53 pp.
55. *Berger, A.,* Hardware and Computer Organization: The Software Perspective, Elsevier, 2005.
56. *Booth, A. D.,* A Signed Binary Multiplication Technique, Quart. J. Mech. Appl. Math. 4, part 2, 1951, pp. 236–240.
57. *Brown, A., Patterson, D. A.,* Towards availability benchmarks: A case study of software RAID systems, Proceedings of the 2000 USENIX Annual Technical Conference, San Diego, CA, June 2000.
58. *Braun, E.,* Digital Computer Design, Logic Circuitry, Synthesis. Academic Press, New York, 1963.
59. *Brewer, E.,* Clustering: Multiply and Conquer, Data Communications, Jul. 1997.
60. *Budruk, R., Anderson, D., Shanley, T.,* PCI Express System Architecture, Addison-Wesley, 2003.
61. *Calder, B., Grunwald, D.,* Fast & Accurate Instruction Fetch and Branch Prediction, ACM SIGARCH Computer Architecture News, Proceedings of the 21st Annual International Symposium on Computer Architecture, Vol. 24, issue 2, Apr. 1994, pp. 2–11.
62. *Chang, P. Y., Hao, E., Yeh, T. Y., Patt, Y.,* Branch Classification: a New Mechanism for Improving Branch Predictor Performance, Proceedings of the 27th ACM/IEEE International Symposium on Microarchitecture, Dec. 1994, pp. 22–31.
63. *Chen, P.,* Interconnection Networks Using Shuffles, IEEE Computer, Vol. 14, 1981, pp. 55–64.

64. *Chen, T. C., Ho, I. T.,* Storage-Efficient Representation of Decimal Data, CACM, 18 (1), January 1975, pp 49–52.
65. *Cheong, H., Veldenbaum, A.,* Software-directed Cache Management in Multiprocessors, Cache and Interconnect Architectures in Multiprocessors, 1990, pp. 259-276.
66. *Chong, Y. M.,* Data Flow Chip Optimizes Image Processing, Computer Design, Oct. 1984, pp. 97–103.
67. *Clos, C.*, A Study of Non-Blocking Switching Networks, Bell Systems Technical Journal, Mar. 1953, pp. 406–424.
68. *Cocke, J., Sweeney D. W.*, High Speed Arithmetic in a Parallel Device, Technical Report IBM, Feb. 1957.
69. *Comte, D., Hifdi, N.,* LAU Multiprocessor: Microfunctional Description and Technologic Choice, Proceedings of the 1st Europe Conference on Parallel and Distributed Processing, Feb. 1979, pp. 8–15.
70. *Comte, D., Hifdi, N., Syre, J. C.*, The Data Driven LAU Multiprocessor System: Results and Perspectives, Proceedings World Comp. Congress IFIP '80, Oct. 1980, pp. 175–180.
71. *Cornish, M.*, The TI Dataflow Architecture: The Power of Concurrency for Avionics, Proceedings of the 3rd Conference on Digital Avionics Systems, Nov. 1979, pp. 19–25.
72. *Cowlishaw, M. F.*, Densely Packed Decimal Encoding, IEEE Procedings — Computers and Digital Techniques, 149 (3), May 2002, pp. 102–104.
73. *Dasgupta, S.*, A Hierarchical Taxonomic System for Computer, Computer, Vol. 23, № 3, 1990, pp. 64–74.
74. *Dandamudi, S. P.*, Fundamentals of Computer Organization and Design, Springer, 2003.
75. *Davis, A. L.*, The Architecture and System Method of DMMI: A Recursively Structured Data Driven Machine, Proceedings 5th ISCA, Apr. 1978, pp. 210–215.
76. *Denning, P.*, The Working Set Model for Program Behaviour, Communications of the ACM, May 1968, pp. 323–333.
77. *Dennis, J. B., Misunas, D. P.,* A Preliminary Architecture for a Basic Dataflow Processor, Proceedings of the 2nd Annual Symposium on Computer Architecture, 1975, pp. 126–132.
78. *Dennis, J. B.*, Dataflow Supercomputers, IEEE Computer, № 13, 1980, pp. 48–56.
79. *Dubois, M., Scheurich, C., Briggs, F. A.*, Synchronization, Coherence, and Event Ordering in Multiprocessors, Computer, Vol. 21, № 2, Feb. 1988, pp. 9–21.
80. *Duncan, R.*, A Survey of Parallel Computer Architectures, Computer, Vol. 23, № 2, 1990, pp. 5–16.
81. *El-Rewini, H., Abd-El-Bar, M.,* Advanced Computer Architecture and Parallel Processing, John Wiley & Sons, 2005.
82. *Evers, P., Chang, C., Patt, Y.*, Using Hybrid Branch Predictors to Improve Branch Prediction Accuracy in the Presence of Context Switches, Proceedings of the 23rd International Symposium on Computer Architecture, May 1996, pp. 3–11.
83. *Flood, J. E.*, Telecommunications Switching, Traffic and Networks, Prentice-Hall, 1995.
84. *Flynn, M. J.*, Very High-Speed Computing System, Proceedings IEEE, № 54, 1966, pp. 1901–1909.
85. *Flynn, M. J.*, Some Computer Organizations and their Effectiveness, IEEE Transactions on Computers, Vol. 24, Sep. 1972, pp. 948–960.
86. *Flynn, M. J.*, Parallel Processors Were the Future... and May Yet Be, IEEE Computer, Vol. 29, №. 12, Dec. 1996, pp. 151–152.

87. *Gloy, N.*, et al., An Analysis of Dynamic Branch Prediction Schemes on System Workloads, ACM SIGARCH Computer Architecture News, Proceedings of the 23rd Annual International Symposium on Computer Architecture, May 1996, pp. 12–21.
88. *Goodman, J. R.*, Using Cache Memory to Reduce Processor-Memory Traffic, Proceedings of the 10th International Symposium on Computer Architecture, 1983.
89. *Gurd, J. R.* The Manchester dataflow machine. Future Generations Computer Systems 1, 1985, pp. 201–212.
90. *Gustafson, J. L.*, Reevaluating Amdahl's Law, CACM, 31(5), 1988, pp. 532–533.
91. *Hayes, J. P.*, Computer Architecture and Organization, 2nd International Edition, McGraw-Hill Book Company, Singapore, 1988.
92. *Handler, W.*, On Classification Schemes for Computer Systems in the Post von Neumann Era, Lecture Notes in Computer Science, 1975.
93. *Hennessy, J. L., Patterson, D. A.*, Computer Architecture: A Quantitative Approach, 4th Edition, Morgan Kaufmann Publishers, San Francisco, CA, USA, 2007.
94. *Hennessy, J. L., Patterson, D. A.*, Computer Organization and Design: The Hardware/Software Interface, 4th Edition, Morgan Kaufmann Publishers, San Francisco, CA, USA, 2009.
95. *Higbie, L. C.*, Supercomputer architecture, Computer, Vol. 6, № 12, 1973, pp. 48–56.
96. *Hill, M.*, Evaluating Associativity in CPU Caches, IEEE Transactions on Computers, Dec. 1989.
97. *Hiraki, K., Shimada, T., Nishida, K.*, A hardware design of the SIGMA-1, a dataflow computer for scientific computations, Proceedings 1984 ICCP, Aug. 1984, pp. 524–531.
98. *Hiraki, K., Sekiguchi, S., Shimada, T.*, Status report of SIGMA-1: A Dataflow Supercomputer, Advanced Topics in Data-Flow Computing (Gaudiot J-L., and Bic L., eds.), Prentice-Hall, 1991, pp. 207–223.
99. *Huck, T.*, Comparative Analysis of Computer Architectures, Stanford University Technical Report, № 83–243, May 1983.
100. IEC 60027–2, Letter Symbols to be Used in Electrical Technology — Part2, Telecommunication and Electronics, Nov. 2000.
101. *Iwashita, M.*, Modular Dataflow Image Processor, Proceedings COMPCON Fall '83, 1983, pp. 464–467.
102. *James, D. V.*, SCI (Scalable Coherent Interface) Cache Coherence, Cache and Interconnect Architectures in Multiprocessors, 1990, pp. 189–208.
103. *Katz, R. H., Eggers, S. J., Wood, D. A., Perkins, C. L., Sheldon, R. G.*, Implementing a Cache Consistency Protocol, Proceedings of the 12th International Symposium on Computer Architecture, 1985.
104. *Kishi, M., Yasuhara, H., Kawamura, Y.*, DDDP: A Distributed Data Driven Processor, Proceedings 10th ISCA, Jun. 1983, pp. 236–242.
105. *Kung, S. Y.*, On Supercomputing with Systolic/Wavefront Array Processors, Proceedings IEEE, Vol. 72, № 47, 1984, pp. 867–884.
106. *Lawrie, D. H.*, Access and Alignment of Data in an Array Processor, IEEE Transactions on Computers, C-24, № 12, 1975, pp. 1145–1155.
107. *Lee, J., Smith, A.*, Branch Prediction Strategies and Branch Target Buffer Design, IEEE Computer, Vol. 17 (1), Jan. 1984, pp. 6–21.
108. *Lehman, M.*, High-speed Digital Multiplication, IRE Transaction on Electronic Computers, Vol. EC-6–6, № 3, 1957.

109. *Lenfant, J.*, Parallel Permutations of Data: A Benes Network Control Algorithms for Frequently Used Permutations, IEEE Transactions on Computers, C-27, № 7, 1978, pp. 637–647.
110. *Lilja, D.*, Reducing the Branch Penalty in Pipelined Processors, Computer, Jul. 1988.
111. *Lilja, D. J.*, Cache Coherence in Large-Scale Shared-Memory Multiprocessors: Issues and Comparison, ACM Computing Surveys, 25 (3), Sep. 1993.
112. *Lipovski, G. L.*, Banyan Networks for Partitioning Multiprocessor Systems, Proceedings of the 1st International Symposium on Computer Architecture, 1973, pp. 21–28.
113. *Lovett, T., Clapp, R.*, Implementation and Performance of a CC-NUMA System, Proceedings of the 23rd Annual International Symposium on Computer Architecture, May 1996.
114. *Lu, M.*, Arithmetic and Logic in Computer Systems, John Wiley & Sons, 2004.
115. *Lunde, A.*, Empirical Evaluation of Some Features of Instruction Set Processor Architectures, Communications of the ACM, Mar. 1977.
116. *Mano, M. M.*, Computer System Architecture, 3rd Edition. L.A: Prentice-Hall, 1995.
117. *Manu, T.*, Cache Coherence for Scalable Shared Memory Multiprocessors, Technical Report, Computer System Laboratory, Stanford University, May 1992.
118. *Mayberry, W., Efland, G.*, Cache Boosts Multiprocessor Performance, Computer Design, Nov. 1984.
119. *McFarling*, Combining Branch Predictors, WRL Technical Note TN-36, Digital Equipment Corporation, Jun., 1993.
120. *Moor, G.*, Cramming More Components onto Integrated Circuits, Electronics, Vol. 38, N7, Apr, 19, 1965, pp. 114–117.
121. *Mou, Z., Jutand, F.*, Overturned Stairs Adder Trees and Multiplier Design, IEEE Transactions on Computers, C-41, Apr. 1992, pp. 940–948.
122. *Murdocca, M. J., Heuring V. P.*, Computer Architecture and Organization: An Integrated Approach, John Wiley & Sons, 2007.
123. *Nassimi, D.*, A Self-Routing Benes Network and Parallel Permutation Algorithms, IEEE Transactions on Computers, C-30, № 5, 1981, pp. 332–340.
124. *Null, L., Lobur, J.*, The Essentials of Computer Organization and Architecture, Jones and Barlett Publishers, 2006.
125. An Overview Of Computational Science, http://csep1.phy.ornl.gov/ov/ov.html.
126. *Parhami, B.*, Computer Arithmetic: Algorithms and Hardware Design, Oxford University Press, 2000.
127. *Patel, J. H.*, Performance of Processor-Memory Interconnection for Multiprocessors, IEEE Transactions on Computers, Vol. 30, № 10, Oct. 1981, pp. 881–780.
128. *Patterson, D., Ditzel, D.*, The case for the reduced instruction set computer, Computer Architecture News Vol. 8, № 6, Oct. 1980, pp. 25–33.
129. *Patterson, D. A., Garth, R., Katz, R.*, A Case for Redundant Arrays of Inexpensive Disks (RAID), University of California, Berkeley, Report № UCB SCD/87/391, Dec. 1987.
130. *Pezaris, S. D.*, A 40-ns 17b by 17b Array Multiplier, IEEE Transactions on Computers, C-20, Apr. 1971, pp. 442–447.
131. *Przybylski, S.*, The Performance Impact of Block Size and Fetch Strategies, Proceedings of the 17th Annual International Symposium on Computer Architecture, May 1990.
132. *Robertson, J. E.*, A New Class of Digital Division Methods, IEEE Transactions on Computers, Electronic Computers, EC-7, Sep. 1958, pp. 218–222.
133. International Roadmap for Semiconductors, http://public.itrs.net.

134. *Shen, J. P., Lipasti, M. H.*, Modern Processor Design: Fundamentals of Superscalar Processors , Tata McGraw-Hill Publishing Company, 2005.
135. *Shiva, S. G.*, Computer Organization, Design, and Architecture, 4th edition, CRC Press, 2007.
136. *Sima, D., Fountain, T., Kacsuk, P.*, Advanced Computer Architectures — a Design Space Approach, Addison-Wesley, 1997, 766 pp.
137. *Skillicorn, D. A.*, Taxonomy for Computer Architectures, Computer, Vol. 21, № 11, 1988, pp. 46–57.
138. *Smith, J.*, A Study of Branch Prediction Strategies, ISCA, Proceedings of the 8th Annual International Symposium on Computer Architecture (selected papers), May 1981.
139. *Smith, A.*, Cache Memories, ACM Computing Surveys, Sep. 1982.
140. *Smith, A.*, Line (Block) Size Choice for CPU Cache Memories, IEEE Transactions on Communications, Sep. 1987.
141. *Smith, J. E.*, A Study of Branch Prediction Strategies, ISCA, ACM, 25 Years of the International Symposium on Computer Architecture (selected papers), Aug. 1998.
142. *Snelling, D. F.*, The Design and Analysis of a Stateless Data-Flow Architectures, Technical Report UMCS-93-7-2, University of Manchester, Department of Computer Science, 1993.
143. *Stallings, W.*, Computer Organization and Architecture, 8th Edition, Prentice-Hall, 2009.
144. *Stoy, J. E.*, Denotational Semantics, MIT Press, 1981.
145. *Stone, H. S.*, Parallel Processing with the Perfect Shuffle, IEEE Transactions on Computers, C-20, № 2, 1971, pp. 153–161.
146. *Sun, X. H., Ni, L. M.*, Another view on parallel speedup, NY. Proc. of Conference on High Performance Networking and Computing, 1990, pp. 324–333.
147. *Sun, X. H., Ni, L. M.*, Scalable Problems and Memory-Bounded Speedup, Journal of Parallel and Distributed Computing, № 19, 1993, pp. 27–37.
148. *Takahashi, N.*, Amamija, M., A Data Flow Processor Array System: Design and Analysis, Proceedings 10th ISCA, Jun. 1983, pp. 243–250.
149. *Talcott, A. R., Yamamoto, W., Serrano, M. J., Wood, R. C., Nemirovsky, V.*, The Impact of Unresolved Branches on Branch Prediction Scheme Performance, ACM SIGARCH Computer Architecture News, Proceedings of the 21st Annual International Symposium on Computer Architecture, Vol. 24, issue 2, Apr. 1994, pp. 12–21.
150. *Tamic, Y., Sequin, C.*, Strategies for Managing the Register File in RISC, IEEE Transactions on Computers, Nov. 1983.
151. *Tanenbaum, A.*, Implications of Structured Programming for Machine Architecture, Communications of the ACM, Mar. 1978.
152. *Tanenbaum, A.*, Structured Computer Organization, 5th edition, Prentice-Hall International, 2006.
153. *Tarnoff, D.*, Computer Organization and Design Fundamentals, e-Book, David Tarnoff, 2007, 432 pp.
154. *Taylor, F. J., Ma, G-K.*, Multiplier policies for digital signal processing, IEEE ASSP Magazine, vol. 7, Jan. 1990, pp. 6–20.
155. *Thacker, C. P., Stewart, L. C., Satterthwaite, E. H.*, Firefly: A Multiprocessor Workstation, IEEE Transactions on Computers, Vol. 37, № 8, Aug. 1988, pp. 909–920.
156. *Thurber, K. J.*, Large Scale Computer Architecture, Hayden Book Company, Rochelle Park, New Jersey, 1976.

157. *Tocher, K. D.*, Techniques of Multiplication and Division for Automatic Binary Computers, Quart. J. Mech. Appl. Math., 11, Jul./Sep. 1958, pp. 364–384.
158. *Tomasevic, M., Milutinovic, V.*, The Cache Coherence Problem in Shared-Memory Multiprocessors, IEEE Computer Society Press, Los Alamitos, CA. 1993.
159. *Treleaven, P. C., Brownbridge, D. R., Hopkins, R. P.*, Data-Driven and Demand-Driven Computer Architecture, ACM Computing Surveys, 14 (1), Mar. 1982.
160. *Vedder, R., Campbell, M., Tucker, G.*, The Hudges Data Flow Multiprocessor, Proceedings of the 5th International Conference on Distributed Computing Systems, May 1985, pp. 324–332.
161. *Von Neumann, J.*, First Draft of a Report on the EDVAC, Moore School, University of Pennsylvania, 1945.
162. *Whitney, S., et al.*, The SGI Origin Software Environment and Application Performance, Proceedings COMPCON Spring '97, Feb. 1997.
163. *Wilkes, M.*, The Best Way to Design an Automatic Calculating Machine, Proceedings Manchester University Computer Inaugural Conference, Jul. 1951.
164. *Wilkinson, B.*, Computer Architecture: Design and Performance, New York: Prentice-Hall, 1996.
165. *Williams, R.*, Computer Systems Architecture: A Networking Approach, 2nd edition, Prentice-Hall International, 2006.
166. *Xie, J., Guo, B.*, An Evaluation and Comparative Analysis of Branch Prediction Schemes on Alpha Processors, Duke University, Department of Computer Science, Dec. 5, 2000.
167. *Yeh, T., Patt, Y. N.*, Alternative Implementation of Two-Level Adaptive Branch Prediction, Proceedings of the International Symposium on Computer Architecture, 1992, pp. 124–134.
168. *Yeh, T., Patt, Y. N.*, Comparison of Dynamic Branch Predictors that use Two Levels of Branch History, Proceedings of the International Symposium on Computer Architecture, 1993, pp. 257–265.
169. *Young, C., Gloy, N., Smith, N.*, A Comparative Analysis of Schemes for Correlated Branch Prediction, Proceedings of the International Symposium on Computer Architecture, 1995, pp. 287–295.
170. *Zhou, M., Su, Z.*, A Comparative Analysis of Branch Prediction Schemes, Technical Report, University of California, Berkeley, 1995.
171. *Zomaya, Y.*, Parallel and Distributed Computing HandBook, McGraw, 1997.

Алфавитный указатель

Б

В

Г

Д

Е

Ж

З

И

К

Н

О

П

Р

С

Т

У

Ф

Х

Ц

Ч

Ш

Э

Я

Сергей Александрович Орлов, Борис Яковлевич Цилькер

Организация ЭВМ и систем: Учебник для вузов

2-е издание

Заведующий редакцией	*А. Кривцов*
Руководитель проекта	*А. Юрченко*
Ведущий редактор	*Ю. Сергиенко*
Художественный редактор	*К. Радзевич*
Корректор	*В. Листова*
Верстка	*Л. Родионова*

Подписано в печать 22.07.10. Формат 70x100/16. Усл. п. л. 55,47. Тираж 2500. Заказ 0000.
ООО «Лидер», 194044, Санкт-Петербург, Б. Сампсониевский пр., 29а.
Налоговая льгота — общероссийский классификатор продукции ОК 005-93, том 2; 95 3005 — литература учебная.
Отпечатано по технологии CtP в ОАО «Печатный двор» им. А. М. Горького.
197110, Санкт-Петербург, Чкаловский пр., 15.

УВАЖАЕМЫЕ ГОСПОДА!
КНИГИ ИЗДАТЕЛЬСКОГО ДОМА «ПИТЕР»
ВЫ МОЖЕТЕ ПРИОБРЕСТИ
ОПТОМ И В РОЗНИЦУ У НАШИХ
РЕГИОНАЛЬНЫХ ПАРТНЕРОВ.

ДАЛЬНИЙ ВОСТОК

Владивосток
«Приморский торговый дом книги»
тел./факс: (4232) 23-82-12
e-mail: bookbase@mail.primorye.ru

Хабаровск, «Деловая книга», ул. Путевая, д. 1а
тел.: (4212) 36-06-65, 33-95-31
e-mail: dkniga@mail.kht.ru

Хабаровск, «Книжный мир»
тел.: (4212) 32-85-51, факс: (4212) 32-82-50
e-mail: postmaster@worldbooks.kht.ru

Хабаровск, «Мирс»
тел.: (4212) 39-49-60
e-mail: zakaz@booksmirs.ru

ЕВРОПЕЙСКИЕ РЕГИОНЫ РОССИИ

Архангельск, «Дом книги», пл. Ленина, д. 3
тел.: (8182) 65-41-34, 65-38-79
e-mail: marketing@avfkniga.ru

Воронеж, «Амиталь», пл. Ленина, д. 4
тел.: (4732) 26-77-77
http://www.amital.ru

Калининград, «Вестер»,
сеть магазинов «Книги и книжечки»
тел./факс: (4012) 21-56-28, 6 5-65-68
e-mail: nshibkova@vester.ru
http://www.vester.ru

Самара, «Чакона», ТЦ «Фрегат»
Московское шоссе, д.15
тел.: (846) 331-22-33
e-mail: chaconne@chaccone.ru

Саратов, «Читающий Саратов»
пр. Революции, д. 58
тел.: (4732) 51-28-93, 47-00-81
e-mail: manager@kmsvrn.ru

СЕВЕРНЫЙ КАВКАЗ

Ессентуки, «Россы», ул. Октябрьская, 424
тел./факс: (87934) 6-93-09
e-mail: rossy@kmw.ru

СИБИРЬ

Иркутск, «ПродаЛитЪ»
тел.: (3952) 20-09-17, 24-17-77
e-mail: prodalit@irk.ru
http://www.prodalit.irk.ru

Иркутск, «Светлана»
тел./факс: (3952) 25-25-90
e-mail: kkcbooks@bk.ru
http://www.kkcbooks.ru

Красноярск, «Книжный мир»
пр. Мира, д. 86
тел./факс: (3912) 27-39-71
e-mail: book-world@public.krasnet.ru

Новосибирск, «Топ-книга»
тел.: (383) 336-10-26
факс: (383) 336-10-27
e-mail: office@top-kniga.ru
http://www.top-kniga.ru

ТАТАРСТАН

Казань, «Таис»,
сеть магазинов «Дом книги»
тел.: (843) 272-34-55
e-mail: tais@bancorp.ru

УРАЛ

Екатеринбург, ООО «Дом книги»
ул. Антона Валека, д. 12
тел./факс: (343) 358-18-98, 358-14-84
e-mail: domknigi@k66.ru

Екатеринбург, ТЦ «Люмна»
ул. Студенческая, д. 1в
тел./факс: (343) 228-10-70
e-mail: igm@lumna.ru
http://www.lumna.ru

Челябинск, ООО «ИнтерСервис ЛТД»
ул. Артиллерийская, д. 124
тел.: (351) 247-74-03, 247-74-09,
247-74-16
e-mail: zakup@intser.ru
http://www.fkniga.ru, www.intser.ru