А. Е. ЖУРАВЛЕВ

ОРГАНИЗАЦИЯ И АРХИТЕКТУРА ЭВМ. ВЫЧИСЛИТЕЛЬНЫЕ СИСТЕМЫ

Учебное пособие
Издание второе, стереотипное

ЛАНЬ

• САНКТ-ПЕТЕРБУРГ • МОСКВА • КРАСНОДАР •
• 2021 •

УДК 004.2
ББК 32.973-018я723

Ж 91 **Журавлев А. Е.** Организация и архитектура ЭВМ. Вычислительные системы : учебное пособие для СПО / А. Е. Журавлев. — 2-е изд., стер. — Санкт-Петербург : Лань, 2021. — 144 с. : ил. — Текст : непосредственный.

ISBN 978-5-8114-8611-3

В учебном пособии излагаются элементы общей теории и эволюции разработки компьютерной архитектуры и рассматриваются вопросы организации компьютерной структуры различных систем на определенных уровнях. Описываются принципы и способы управления ресурсами компонентов и компьютера и варианты их реализации.

Учебное пособие предназначено для учащихся, обучающихся по специальностям среднего профессионального образования «Прикладная математика и информатика», «Информационные системы» и «Организация и технология защиты информации», изучающих курсы профессионального учебного цикла «Архитектура ЭВМ и систем», «Организация ЭВМ и вычислительных систем» и т. п., в качестве основной литературы. Пособие может быть интересно аспирантам и преподавателям, чья деятельность связана с теоретическими и прикладными аспектами моделирования и управления аппаратными ресурсами и электроникой.

УДК 004.2
ББК 32.973-018я723

Обложка
П. И. ПОЛЯКОВА

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Компьютер (ЭВМ) — это устройство, которое решает задачи, выполняя полученные команды. Программа — это последовательность команд, описывающих решение определенной задачи. Электроника каждого компьютера может распознавать и выполнять ограниченный набор простых команд. Все программы перед выполнением должны быть преобразованы в последовательность таких команд, которые обычно не сложнее чем:

– сложить пару чисел;

– проверить число на нулевое значение;

– скопировать часть данных из одной области памяти компьютера в другую.

Эти примитивные команды в совокупности составляют язык, на котором организуется диалог человека с компьютером. Такой язык называется машинным языком. Разработчик при создании нового компьютера должен решить, какие команды включить в машинный язык этого компьютера в зависимости от назначения компьютера, т. е. спектра выполняемых им задач. Машинные команды делают как можно проще, чтобы избежать сложностей при конструировании компьютера и снизить затраты на необходимую электронику. Так как большинство машинных языков очень примитивны, использовать их очень сложно.

Сложность использования машинных языков привела к построению ряда уровней абстракций, каждый из которых надстраивается над абстракцией более низкого уровня. Этот подход называется многоуровневой компьютерной организацией.

1. ОРГАНИЗАЦИЯ КОМПЬЮТЕРНОЙ СТРУКТУРЫ

1.1. Языки, уровни и виртуальные машины

Проблему можно решить двумя способами. Оба способа включают в себя разработку новых команд, которые более удобны для человека, чем встроенные машинные команды. Эти новые команды в совокупности формируют язык, который назовем L_1. Встроенные машинные команды тоже формируют язык, назовем его L_0. Компьютер может выполнять только программы, написанные на его машинном языке L_0. Два способа решения проблемы различаются тем, каким образом компьютер будет выполнять программы, написанные на языке L_1.

Первый способ выполнения программы, написанной на языке L_1, — замена каждой команды на эквивалентный набор команд в языке L_0. В этом случае компьютер выполняет новую программу, написанную на языке L_0, вместо старой программы, написанной на L_1. Эта технология называется трансляцией.

Второй способ — написание программы на языке L_0, которая берет программы, написанные на языке L_1, в качестве входных данных, рассматривает каждую команду по очереди и сразу выполняет эквивалентный набор команд языка L_0. Эта технология не требует составления новой программы на L_0. Она называется интерпретацией, а программа, которая осуществляет интерпретацию, называется интерпретатором.

Трансляция и интерпретация сходны. При применении обоих методов компьютер в итоге выполняет набор команд на языке L_0, эквивалентных командам L_1. Различие лишь в том, что при трансляции вся программа L_1 преобразуется в программу L_0, программа L_1 отбрасывается, а новая программа на L_0 загружается в память компьютера и затем выполняется.

При интерпретации каждая команда программы на L_1 перекодируется в L_0 и сразу же выполняется. В отличие от трансляции, здесь не создается новая программа на L_0, а происходит последовательная перекодировка и выполнение команд. Оба эти метода, а также их комбинация широко используются.

Обычно проще представить существование гипотетического компьютера или виртуальной машины, для которой машинным языком является язык L_1. Назовем такую виртуальную машину M_1, а виртуальную машину с

языком L_0 — M_0. Если бы такую машину M_1 можно было бы сконструировать без больших денежных затрат, то язык L_0 и машина M_0 были бы не нужны. Можно было бы просто писать программы на языке L_1, а компьютер сразу бы их выполнял. Даже если виртуальная машина слишком дорога или ее очень трудно сконструировать, можно писать программы для нее. Эти программы могут транслироваться или интерпретироваться программой, написанной на языке L_0, которая сама могла бы выполняться фактически существующим компьютером, т. е. можно писать программы для виртуальных машин, как будто они действительно существуют.

Чтобы трансляция и интерпретация были целесообразными, языки L_0 и L_1 не должны сильно различаться. Это значит, что язык L_1 хотя и лучше, чем L_0, но все же далек от идеала. Очевидное решение этой проблемы — создание еще одного набора команд, которые в большей степени ориентированы на человека и в меньшей степени на компьютер, чем L_1. Этот третий набор команд также формирует язык, который назовем L_2, а соответствующую виртуальную машину — M_2. Человек может писать программы на языке L_2, как будто виртуальная машина M_2 с машинным языком L_2 действительно существует. Такие программы могут или транслироваться на язык L_1, или выполняться интерпретатором, написанным на языке L_1.

Изобретение целого ряда языков, каждый из которых более удобен для человека, чем предыдущий, может продолжаться до тех пор, пока не будет достигнут желаемый уровень удобства. Каждый такой язык использует своего предшественника как основу, поэтому можно рассматривать компьютер в виде ряда уровней, как показано на рисунке 1.1. Язык, находящийся в самом низу иерархической структуры, — самый примитивный, а находящийся на самом верху — самый сложный.

Между языком и виртуальной машиной существует важная зависимость. У каждой машины есть определенный машинный язык, состоящий из полного множества команд, которые эта машина может выполнять. В сущности, машина определяет язык. Сходным образом язык определяет машину, которая может выполнять все программы, написанные на этом языке. Машину, задающуюся определенным языком, очень сложно и дорого сконструировать из электронных схем, но ее можно представить. Компьютер с машинным языком C++ был бы слишком сложным, но его можно было бы сконструировать, если учитывать высокий уровень современных технологий. Однако существуют веские причины не создавать такой компьютер:

это слишком сложно и дорого по сравнению с существующими альтернативами.

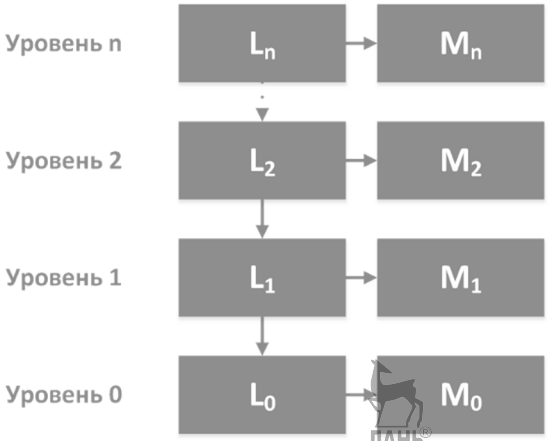


Рис. 1.1. Многоуровневая машина

Компьютер с n уровнями можно рассматривать как n разных виртуальных машин, у каждой из которых есть свой машинный язык. Термины «уровень» и «виртуальная машина» считаем синонимами. Только программы, написанные на L_0, могут выполняться компьютером без применения трансляции и интерпретации. Программы, написанные на L_1, L_2, ..., L_n, должны проходить через интерпретатор более низкого уровня или транслироваться на язык, соответствующий более низкому уровню.

Человеку, который пишет программы для виртуальной машины уровня n, не обязательно знать о трансляторах и интерпретаторах более низких уровней. Машина выполнит эти программы, и не важно, будут ли они выполняться шаг за шагом интерпретатором или их будет выполнять сама машина. В обоих случаях результат один и тот же: программа будет выполнена.

Большинство программистов, использующих машину уровня n, интересуются только самым верхним уровнем, т. е. уровнем, который меньше всего сходен с машинным языком. Те, кто проектирует новые компьютеры или новые уровни (т. е. новые виртуальные машины), также должны быть знакомы со всеми уровнями. Понятия и технические приемы конструирования машин как системы уровней, а также детали уровней составляют основной предмет дисциплины.

1.2. Актуальные многоуровневые комплексы

Большинство современных компьютеров состоит из двух и более уровней. Существуют машины даже с шестью уровнями (рис. 1.2). Уровень 0 — аппаратное обеспечение машины. Его электронные схемы выполняют программы, написанные на языке уровня 1. Ради полноты нужно упомянуть о существовании еще одного уровня, расположенного ниже уровня 0. Этот уровень не показан на рисунке 1.2, так как он попадает в сферу электронной техники. Он называется уровнем физических устройств. На этом уровне находятся транзисторы, которые являются примитивами для разработчиков компьютеров.

Рис. 1.2. Компьютер с шестью уровнями.
Способ поддержки каждого уровня указан под ним
(*в скобках указывается название поддерживающей программы*)

На самом нижнем, цифровом логическом уровне объекты называются вентилями. Хотя вентили состоят из аналоговых компонентов, таких как транзисторы, они могут быть точно смоделированы как цифровые средства. У каждого вентиля есть один или несколько цифровых входов (сигналов, представляющих 0 или 1). Вентиль вычисляет простые функции этих сигналов, такие как «И» или «ИЛИ». Каждый вентиль формируется из нескольких транзисторов. Несколько вентилей формируют 1 бит памяти, который может содержать 0 или 1. Биты памяти, объединенные в группы (например, по 16, 32 или 64), формируют регистры. Каждый регистр может содержать

одно двоичное число до определенного предела. Из вентилей также может состоять сам компьютер.

Следующий уровень (первый) — микроархитектурный уровень**.** На этом уровне можно видеть совокупности 8 или 32 регистров, которые формируют локальную память и схему, называемую АЛУ (арифметико-логическое устройство). АЛУ выполняет простые арифметические операции. Регистры вместе с АЛУ формируют тракт данных**,** по которому поступают данные. Основная операция тракта данных состоит в следующем. Выбирается один или два регистра, АЛУ производит над ними какую-либо операцию, например сложения, а результат помещается в один из этих регистров.

На некоторых машинах работа тракта данных контролируется особой программой, которая называется микропрограммой. На других машинах тракт данных контролируется аппаратными средствами. На машинах, где тракт данных контролируется программным обеспечением, микропрограмма — это интерпретатор для команд на уровне 2. Микропрограмма вызывает команды из памяти и выполняет их последовательно, используя тракт данных. Например, для того чтобы выполнить команду сложения (ADD), эта команда вызывается из памяти, ее операнды помещаются в регистры, АЛУ вычисляет сумму, а затем результат переправляется обратно. На компьютере с аппаратным контролем тракта данных происходит такая же процедура, но при этом нет программы, которая контролирует интерпретацию команд уровня 2.

Второй уровень называется уровнем архитектуры системы команд. Каждый производитель публикует руководство для компьютеров, которые он продает, под названием «Руководство по машинному языку» или «Принципы работы компьютера» и т. п. Такие руководства содержат информацию именно об этом уровне. Когда они описывают набор машинных команд, они в действительности описывают команды, которые выполняются микропрограммой-интерпретатором или аппаратным обеспечением. Если производитель поставляет два интерпретатора для одной машины, он должен издать два руководства по машинному языку, отдельно для каждого интерпретатора.

Следующий уровень обычно гибридный. Большинство команд в его языке есть также и на уровне архитектуры системы команд (команды, имеющиеся на одном из уровней, вполне могут находиться на других уровнях). У этого уровня есть некоторые дополнительные особенности: набор новых

команд, другая организация памяти, способность выполнять две и более программ одновременно и некоторые другие. При построении третьего уровня возможно больше вариантов, чем при построении первого и второго.

Новые средства, появившиеся на третьем уровне, выполняются интерпретатором, который работает на втором уровне. Этот интерпретатор был когда-то назван операционной системой. Команды третьего уровня, идентичные командам второго уровня, выполняются микропрограммой или аппаратным обеспечением, но не операционной системой. Иными словами, одна часть команд третьего уровня интерпретируется операционной системой, а другая часть — микропрограммой. Вот почему этот уровень считается гибридным. Этот уровень называется *уровнем операционной системы.*

Между третьим и четвертым уровнями есть существенная разница. Нижние три уровня конструируются не для работы обычного программиста. Они изначально предназначены для работы интерпретаторов и трансляторов, поддерживающих более высокие уровни. Эти трансляторы и интерпретаторы составляются системными программистами, которые специализируются на разработке и построении новых виртуальных машин. Уровни с четвертого и выше предназначены для прикладных программистов, решающих конкретные задачи.

Еще одно изменение, появившееся на уровне 4, — способ, которым поддерживаются более высокие уровни. Уровни 2 и 3 обычно интерпретируются, а уровни 4, 5 и выше обычно, хотя и не всегда, поддерживаются транслятором.

Другое различие между низкими и высокими уровнями — особенность языка. Машинные языки низких уровней — цифровые. Программы, написанные на этих языках, состоят из длинных рядов цифр, которые удобны для компьютеров, но совершенно неудобны для людей. Начиная с четвертого уровня, языки содержат слова и сокращения, понятные человеку.

Четвертый уровень представляет собой символическую форму одного из языков более низкого уровня. На этом уровне можно писать программы в приемлемой для человека форме. Эти программы сначала транслируются на язык низкого уровня, а затем интерпретируются соответствующей виртуальной или фактически существующей машиной. Программа, которая выполняет трансляцию, называется ассемблером**.**

Пятый уровень обычно состоит из языков, разработанных для прикладных программистов. Такие языки называются языками высокого уровня.

Существуют сотни языков высокого уровня. Наиболее известные среди них — BASIC, C, C++, Java, LISP и Prolog. Программы, написанные на этих языках, обычно транслируются на уровень 3 или 4. Трансляторы, которые обрабатывают эти программы, называются компиляторами. Иногда также используется метод интерпретации. Например, программы на языке Java обычно интерпретируются.

В некоторых случаях пятый уровень состоит из интерпретатора для такой сферы приложения, как символическая математика. Он обеспечивает данные и операции для решения задач в этой сфере в ее терминах.

Вывод: компьютер проектируется как иерархическая структура уровней, каждый из которых надстраивается над предыдущим. Каждый уровень представляет собой определенную абстракцию с различными объектами и операциями. Рассматривая компьютер подобным образом, можно не принимать во внимание ненужные детали и свести сложный предмет к более простому для понимания.

Набор типов данных, операций и особенностей каждого уровня называется архитектурой. Архитектура связана с аспектами, которые видны программисту. Например, сведения о том, сколько памяти можно использовать при написании программы, — часть архитектуры, а аспекты разработки (например, какая технология используется при создании памяти) не являются частью архитектуры. Изучение того, как разрабатываются те части компьютерной системы, которые видны программистам, называется изучением компьютерной архитектуры**.** Термины «компьютерная архитектура» и «компьютерная организация» означают в сущности одно и то же.

1.3. Эволюция многоуровневых систем

Программы, написанные на машинном языке (уровень 1), могут сразу выполняться электронными схемами компьютера (уровень 0), без применения интерпретаторов и трансляторов. Эти электронные схемы вместе с памятью и средствами ввода-вывода формируют аппаратное обеспечение**.** Аппаратное обеспечение состоит из осязаемых объектов — интегральных схем, печатных плат, кабелей, источников электропитания, запоминающих устройств и принтеров. Абстрактные понятия, алгоритмы и команды не относятся к аппаратному обеспечению.

Программное обеспечение, напротив, состоит из алгоритмов (подробных последовательностей команд, которые описывают, как решить задачу)

и их компьютерных представлений, т. е. программ. Программы могут храниться на жестком диске, гибком диске, компакт-диске или других носителях, но, в сущности, программное обеспечение — это набор команд, составляющих программы, а не физические носители, на которых эти программы записаны.

В самых первых компьютерах граница между аппаратным и программным обеспечением была очевидна. Со временем, однако, произошло значительное размывание этой границы, в первую очередь благодаря тому, что в процессе развития компьютеров уровни добавлялись, убирались и сливались. В настоящее время очень сложно отделить их друг от друга. В действительности центральную тему можно выразить так: *аппаратное и программное обеспечение логически эквивалентны.*

Любая операция, выполняемая программным обеспечением, может быть встроена в аппаратное обеспечение (желательно после того, как она осознана). Обратное тоже верно: любая команда, выполняемая аппаратным обеспечением, может быть смоделирована в программном обеспечении. Решение разделить функции аппаратного и программного обеспечения основано на таких факторах, как стоимость, скорость, надежность, а также частота ожидаемых изменений. Существует несколько жестких правил, сводящихся к тому, что должно быть в аппаратном обеспечении, а что должно программироваться. Эти решения изменяются в зависимости от тенденций в развитии компьютерных технологий.

2. ЛОГИЧЕСКАЯ ОРГАНИЗАЦИЯ ПРОЦЕССОРОВ

Цифровой компьютер состоит из связанных между собой процессоров, памяти и устройств ввода-вывода.

На рисунке 2.1 показано устройство обычного компьютера. Центральный процессор — это устройство, выполняющее программы, находящиеся в основной памяти. Он вызывает команды из памяти, определяет их тип, а затем выполняет их одну за другой. Компоненты соединены шиной, представляющей собой набор параллельно связанных проводов, по которым передаются адреса, данные и сигналы управления. Шины могут быть внешними (связывающими процессор с памятью и устройствами ввода-вывода) и внутренними.

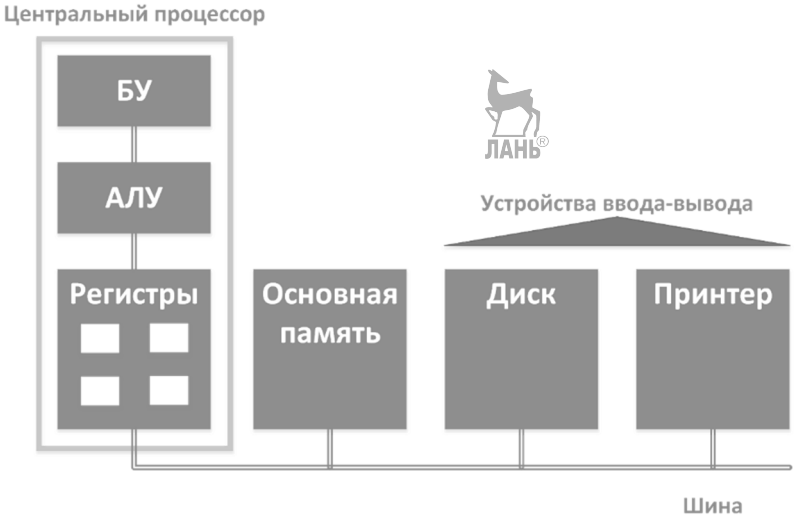


Рис. 2.1. Схема устройства компьютера

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

Внутри центрального процессора находится память для хранения промежуточных результатов и некоторых команд управления. Эта память состоит из нескольких регистров, каждый из которых выполняет определенную функцию. Обычно все регистры одинакового размера. Каждый регистр

содержит одно число, которое ограничивается размером регистра. Регистры считываются и записываются очень быстро, поскольку они находятся внутри центрального процессора.

Самый важный регистр — счетчик команд, который указывает, какую команду нужно выполнять дальше. Также существует регистр команд, в котором находится команда, выполняемая в данный момент. У большинства процессоров имеются и другие регистры, одни из них многофункциональны, другие выполняют только какие-либо специфические функции.

2.1. Устройство центрального процессорного модуля

Внутреннее устройство тракта данных типичного фон-неймановского процессора показано на рисунке 2.2. Тракт данных состоит из регистров (обычно от 1 до 32), АЛУ и нескольких соединяющих шин. Содержимое регистров поступает во входные регистры АЛУ, которые на рисунке 2.2 обозначены буквами *A* и *B*. В них находятся входные данные АЛУ, пока АЛУ производит вычисления. Тракт данных — важная составная часть всех компьютеров.

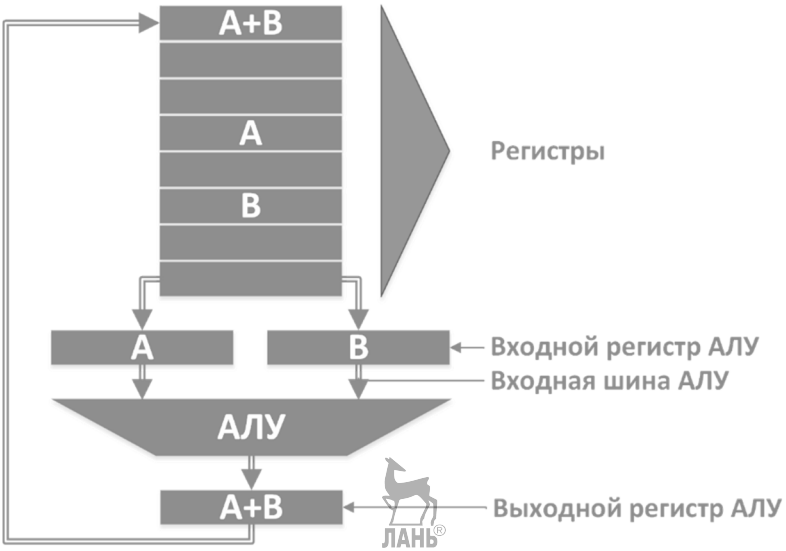


Рис. 2.2. Тракт данных в обычной фон-неймановской машине

АЛУ выполняет сложение, вычитание и другие простые операции над входными данными и помещает результат в выходной регистр. Этот выходной регистр может помещаться обратно в один из регистров. Он может быть сохранен в памяти, если это необходимо. На рисунке 2.2 показана операция сложения. Входные и выходные регистры есть не у всех компьютеров.

Большинство команд можно разделить на две группы.

1. Команды типа «регистр — память». Вызывают слова из памяти, помещают их в регистры, где они используются в качестве входных данных АЛУ. Слово — это элемент данных, который перемещается между памятью и регистрами. Словом может быть целое число. Другие команды этого типа помещают регистры обратно в память. Размер слова обычно соответствует разрядности регистра данных. Например, у 16-битных микропроцессоров 8086 и 8088 слово было 16-битным, а у 32-битных микропроцессоров слово имеет длину 32 бита.

2. Команды типа «регистр — регистр» второго типа вызывают два операнда из регистров, помещают их во входные регистры АЛУ, выполняют над ними необходимую арифметическую или логическую операцию и переносят результат обратно в один из регистров. Этот процесс называется циклом тракта данных. В какой-то степени он определяет функционал машины. Чем быстрее происходит цикл тракта данных, тем выше производительность процессора.

2.1.1. Выполнение команд

Центральный процессор выполняет каждую команду за несколько шагов:

1) вызывает следующую команду из памяти и переносит ее в регистр команд;

2) меняет положение счетчика команд, который теперь должен указывать на следующую команду (это происходит после декодирования текущей команды, а иногда и после ее выполнения);

3) определяет тип вызванной команды;

4) если команда использует слово из памяти, определяет, где находится это слово;

5) переносит слово, если это необходимо, в регистр центрального процессора (бывают команды, которые требуют загрузки из памяти целого множества слов и их обработки в рамках одной-единственной команды);

6) выполняет команду;

7) переходит к шагу 1, чтобы начать выполнение следующей команды.

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

Описание работы центрального процессора можно представить в виде программы. Сама возможность написать программу, имитирующую работу центрального процессора, показывает, что программа не обязательно должна выполняться реальным процессором, относящимся к аппаратному обеспечению. Напротив, вызывать из памяти, определять тип команд и выполнять эти команды может другая программа. Такая программа называется интерпретатором.

Написание программ-интерпретаторов, которые имитируют работу процессора, широко используется при разработке компьютерных систем. После того как разработчики выбрали машинный язык (*L*) для нового компьютера, они должны решить, строить ли им процессор, который будет выполнять программы на языке *L*, или написать специальную программу для интерпретации программ на языке *L*.

Если они решают написать интерпретатор, они должны создать аппаратное обеспечение для выполнения этого интерпретатора. Возможны также гибридные конструкции, когда часть команд выполняется аппаратным обеспечением, а часть интерпретируется.

Интерпретатор разбивает команды на маленькие шаги. Таким образом, машина с интерпретатором может быть гораздо проще по строению и дешевле, чем процессор, выполняющий программы без интерпретации. Такая экономия особенно важна, если компьютер содержит большое количество сложных команд с различными опциями. В сущности, экономия проистекает из самой замены аппаратного обеспечения программным обеспечением (интерпретатором).

Первые компьютеры содержали небольшое количество команд, и эти команды были простыми. Но процесс разработки и создания все более мощных компьютеров привел, кроме всего прочего, к появлению более сложных команд. Вскоре разработчики поняли, что при наличии сложных команд программы выполняются быстрее, хотя выполнение отдельных команд занимает больше времени. В качестве примеров сложных команд можно назвать выполнение операций с плавающей точкой, обеспечение прямого доступа к элементам массива и т. п. Если обнаруживалось, что две определенные команды часто выполнялись последовательно одна за другой, то вводилась новая команда, заменяющая работу этих двух.

Сложные команды были лучше, потому что некоторые операции иногда перекрывались. Какие-то операции могли выполняться параллельно, для этого использовались разные части аппаратного обеспечения. Для дорогих компьютеров с высокой производительностью стоимость этого дополнительного аппаратного обеспечения была вполне оправданна. Таким образом, у дорогих компьютеров было гораздо больше команд, чем у дешевых. Однако развитие программного обеспечения и требования совместимости команд привели к тому, что сложные команды стали использоваться и в дешевых компьютерах, хотя там во главу угла ставилась стоимость, а не скорость работы.

К концу 1950-х гг. компания *IBM*, которая лидировала тогда на компьютерном рынке, решила, что производство семейства компьютеров, каждый из которых выполняет одни и те же команды, имеет много преимуществ и для самой компании, и для покупателей. Чтобы описать этот уровень совместимости, *IBM* ввела термин «архитектура». Новое семейство компьютеров должно было иметь одну общую архитектуру и много разных разработок, различающихся по цене и скорости, которые могли выполнять одну и ту же программу. Но как построить дешевый компьютер, который будет выполнять все сложные команды, предназначенные для высокоэффективных дорогостоящих машин?

Решением этой проблемы стала интерпретация. Эта технология, впервые предложенная Уилксом в 1951 г., позволяла разрабатывать простые дешевые компьютеры, которые, тем не менее, могли выполнять большое количество команд. В результате *IBM* создала архитектуру System/360, семейство совместимых компьютеров, различных по цене и производительности. Аппаратное обеспечение без интерпретации использовалось только в самых дорогих моделях.

Простые компьютеры с интерпретированными командами имели другие преимущества. Наиболее важными среди них были:

– возможность фиксировать неправильно выполненные команды или даже восполнять недостатки аппаратного обеспечения;

– возможность добавлять новые команды при минимальных затратах даже после покупки компьютера;

– структурированная организация, которая позволяла разрабатывать, проверять и документировать сложные команды.

В 1970-х гг. компьютерный рынок быстро разрастался, новые компьютеры могли выполнять все больше и больше функций. Спрос на дешевые компьютеры провоцировал создание компьютеров с использованием интерпретаторов. Возможность разрабатывать аппаратное обеспечение и интерпретатор для определенного набора команд вылилась в создание дешевых процессоров. Полупроводниковые технологии быстро развивались, преимущества низкой стоимости преобладали над возможностями более высокой производительности, и использование интерпретаторов при разработке компьютеров стало широко применимым. Интерпретация использовалась практически во всех компьютерах, выпущенных в 1970-х гг., — от миникомпьютеров до самых больших машин.

К концу 1970-х гг. интерпретаторы стали применяться практически во всех моделях, кроме самых дорогостоящих машин с очень высокой производительностью (например, Cray-1 и компьютеров серии Control Data Cyber). Использование интерпретаторов исключало высокую стоимость сложных команд, и разработчики могли вводить все более и более сложные команды, в особенности различные способы определения используемых операндов.

Эта тенденция достигла пика своего развития в разработке компьютера VAX (производитель *Digital Equipment Corporation*), у которого было несколько сотен команд и более 200 способов определения операндов в каждой команде. К несчастью, архитектура VAX с самого начала разрабатывалась с использованием интерпретатора, а производительности уделялось мало внимания. Это привело к появлению большого количества команд второстепенного значения, которые трудно было выполнять сразу, без интерпретации. Данное упущение стало фатальным как для VAX, так и для его производителя (компании *DEC*). *Compaq* купил *DEC* в 1998 г.

Хотя самые первые 8-битные микропроцессоры были очень простыми и содержали небольшой набор команд, к концу 1970-х гг. даже они стали разрабатываться с использованием интерпретаторов. В этот период основной проблемой для разработчиков стала возрастающая сложность микропроцессоров. Главное преимущество интерпретации заключалось в том, что можно было разработать простой процессор, а вся сложность сводилась к созданию интерпретатора. Таким образом, разработка сложного аппаратного обеспечения замещалась разработкой сложного программного обеспечения.

Успех Motorola 68000 с большим набором интерпретируемых команд и одновременный провал Zilog Z8000, у которого был столь же обширный набор команд, но не было интерпретатора, продемонстрировали все преимущества использования интерпретаторов при разработке новых машин. Успех Motorola 68000 был несколько неожиданным, учитывая, что Z80 (предшественник Zilog Z8000) пользовался большей популярностью, чем Motorola 6800 (предшественник Motorola 68000). Конечно, важную роль здесь играли и другие факторы, например то, что *Motorola* много лет занималась производством микросхем, а *Exxon* (владелец Zilog) долгое время был нефтяной компанией.

Еще один фактор в пользу интерпретации — существование быстрых постоянных запоминающих устройств (так называемых командных ПЗУ) для хранения интерпретаторов. Предположим, что для выполнения обычной интерпретируемой команды Motorola 68000 интерпретатору нужно выполнить десять команд, которые называются микрокомандами, по 100 нс каждая, и произвести два обращения к оперативной памяти по 500 нс каждое. Общее время выполнения команды составит, следовательно, 2000 нс, всего лишь в два раза больше, чем в лучшем случае могло бы занять непосредственное выполнение этой команды без интерпретации. А если бы не было специального быстродействующего постоянного запоминающего устройства, выполнение этой команды заняло бы целых 6000 нс. Таким образом, важность наличия командных ПЗУ очевидна.

2.1.2. RISC и CISC

В конце 1970-х гг. проводилось много экспериментов с очень сложными командами, появление которых стало возможным благодаря интерпретации. Разработчики пытались уменьшить пропасть между тем, что компьютеры способны делать, и тем, что требуют языки высокого уровня. В компании *IBM* группа разработчиков во главе с Джоном Коком противостояла этой тенденции; они попытались воплотить идеи Сеймура Крея, создав экспериментальный высокоэффективный мини-компьютер 801. В 1980 г. группа разработчиков в Университете Беркли во главе с Дэвидом Паттерсоном и Карло Секвином начала разработку процессоров VLSI без использования интерпретации. Для обозначения этого понятия они придумали термин RISC (*Reduced Instruction Set Computer*) и назвали новый процессор RISC I, вслед за которым вскоре был выпущен RISC II. Немного позже, в

1981 г., Джон Хеннеси в Стэнфорде разработал и выпустил другую микросхему, которую он назвал MIPS (*Millions of Instructions Per Second* — миллионы команд в секунду). Эти две микросхемы развились в коммерчески важные продукты SPARC и MIPS соответственно.

Новые процессоры существенно отличались от коммерческих процессоров того времени. Поскольку они не были совместимы с существующей продукцией, разработчики вправе были включать туда новые наборы команд, которые могли бы увеличить общую производительность системы. Так как основное внимание уделялось простым командам, которые могли быстро выполняться, разработчики вскоре осознали, что ключом к высокой производительности компьютера была разработка команд, к выполнению которых можно быстро приступать. Сколько времени занимает выполнение одной команды, было не столь важным, нежели то, сколько команд может быть начато в секунду.

В то время как разрабатывались эти простые процессоры, всеобщее внимание привлекало относительно небольшое количество команд (обычно их было около 50). Для сравнения: число команд в DEC VAX и больших IBM в то время составляло от 200 до 300. RISC — компьютер с сокращенным набором команд. RISC противопоставлялся CISC (*Complex Instruction Set Computer*) — компьютеру с полным набором команд.

В качестве примера CISC можно привести VAX, который доминировал в научных компьютерных центрах. С этого момента началась грандиозная идеологическая война между сторонниками RISC и разработчиками VAX, Intel и больших IBM. По их мнению, наилучший способ разработки компьютеров — включение туда небольшого количества простых команд, каждая из которых выполняется за один цикл тракта данных (см. рис. 2.2), т. е. берет два регистра, производит над ними какую-либо арифметическую или логическую операцию (например, сложения или логическое «И») и помещает результат обратно в регистр. В качестве аргумента они утверждали, что даже если RISC должна выполнять четыре или пять команд вместо одной, которую выполняет CISC, при том что команды RISC выполняются в десять раз быстрее (поскольку они не интерпретируются), он выигрывает в скорости. Следует также отметить, что к этому времени скорость работы основной памяти приблизилась к скорости специальных управляющих постоянных запоминающих устройств, поэтому недостатки интерпретации были налицо, что повышало популярность компьютеров RISC.

Учитывая преимущества производительности RISC, можно было бы предположить, что такие компьютеры, как Alpha компании *DEC*, стали доминировать над компьютерами CISC (Pentium и т. д.) на рынке. Однако ничего подобного не произошло. Возникает вопрос: почему?

Во-первых, компьютеры RISC были несовместимы с другими моделями, а многие компании вложили миллиарды долларов в программное обеспечение для продукции *Intel*. Во-вторых, как ни странно, компания *Intel* сумела воплотить те же идеи в архитектуре CISC. Процессоры Intel, начиная с 486-го, содержат ядро RISC, которое выполняет самые простые (и обычно самые распространенные) команды за один цикл тракта данных, а по обычной технологии CISC интерпретируются более сложные команды. В результате обычные команды выполняются быстро, а более сложные и редкие — медленно. Хотя при таком «гибридном» подходе работа происходит не так быстро, как у RISC, данная архитектура имеет ряд преимуществ, поскольку позволяет использовать старое программное обеспечение без изменений.

2.1.3. Принципы разработки современных компьютеров

Прошло уже почти 40 лет с тех пор, как были сконструированы первые компьютеры RISC, однако некоторые принципы разработки можно перенять, учитывая современное состояние технологий аппаратного обеспечения. Если происходит очень резкое изменение в технологиях (например, новый процесс производства делает время цикла памяти в 10 раз меньше, чем время цикла центрального процессора), меняются все условия. Поэтому разработчики всегда должны учитывать возможные технологические изменения, которые могут повлиять на баланс между компонентами компьютера.

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

Все команды непосредственно выполняются аппаратным обеспечением. Все обычные команды непосредственно выполняются аппаратным обеспечением. Они не интерпретируются микрокомандами. Устранение

уровня интерпретации обеспечивает высокую скорость выполнения большинства команд. В компьютерах типа CISC более сложные команды могут разбиваться на несколько частей, которые затем выполняются как последовательность микрокоманд. Эта дополнительная операция снижает скорость работы машины, но она может быть применима для редко встречающихся команд.

Компьютер должен начинать выполнение большого числа команд. В современных компьютерах используется много различных способов для увеличения производительности, главное из которых — возможность обращаться к как можно большему количеству команд в секунду. Процессор 500-MIPS способен приступать к выполнению 500 млн команд в секунду, и при этом не имеет значения, сколько времени занимает само выполнение этих команд. Этот принцип предполагает, что параллелизм может играть главную роль в улучшении производительности, поскольку приступать к большому количеству команд за короткий промежуток времени можно только в том случае, если одновременно может выполняться несколько команд.

Хотя команды некоторой программы всегда расположены в определенном порядке, компьютер может приступать к их выполнению и в другом порядке (так как необходимые ресурсы памяти могут быть заняты) и, кроме того, может заканчивать их выполнение не в том порядке, в котором они расположены в программе. Конечно, если команда 1 устанавливает регистр, а команда 2 использует этот регистр, нужно действовать с особой осторожностью, чтобы команда 2 не считывала регистр до тех пор, пока он не будет содержать нужное значение. Чтобы не допустить подобных ошибок, необходимо вводить большое количество соответствующих записей в память, но производительность все равно становится выше благодаря возможности выполнять несколько команд одновременно.

Команды должны легко декодироваться. Предел количества вызываемых команд в секунду зависит от процесса декодирования отдельных команд. Декодирование команд осуществляется для того, чтобы определить, какие ресурсы им необходимы и какие действия нужно выполнить. Полезны любые средства, которые способствуют упрощению этого процесса. Например, используются регулярные команды с фиксированной длиной и с небольшим количеством полей. Чем меньше разных форматов команд, тем лучше.

К памяти должны обращаться только команды загрузки и сохранения. Один из самых простых способов разбивания операций на отдельные шаги — потребовать, чтобы операнды для большинства команд брались из регистров и возвращались туда же. Операция перемещения операндов из памяти в регистры может осуществляться в разных командах. Поскольку доступ к памяти занимает много времени, а подобная задержка нежелательна, работу этих команд могут выполнять другие команды, если они не делают ничего, кроме передвижения операндов между регистрами и памятью. Из этого наблюдения следует, что к памяти должны обращаться только команды загрузки и сохранения (LOAD и STORE).

Должно быть большое количество регистров. Поскольку доступ к памяти происходит довольно медленно, в компьютере должно быть много регистров (по крайней мере, 32). Если слово однажды вызвано из памяти, при наличии большого числа регистров оно может содержаться в регистре до тех пор, пока станет ненужным. Возвращение слова из регистра в память и новая загрузка этого же слова в регистр нежелательны. Лучший способ избежать излишних перемещений — наличие достаточного количества регистров.

2.1.4. Параллелизм на уровне команд

Разработчики компьютеров стремятся к тому, чтобы улучшить производительность машин, над которыми они работают. Один из способов заставить процессоры работать быстрее — увеличение их скорости, однако при этом существуют некоторые технологические ограничения, связанные с конкретным историческим периодом. Поэтому большинство разработчиков для достижения лучшей производительности при данной скорости работы процессора используют параллелизм (возможность выполнять две или более операций одновременно).

Существуют две основные формы параллелизма: на уровне команд и на уровне процессоров. В первом случае параллелизм осуществляется в пределах отдельных команд и обеспечивает выполнение большого количества команд в секунду. Во втором случае над одной задачей работают одновременно несколько процессоров. Каждый подход имеет свои преимущества. В этом разделе будет рассмотрен параллелизм на уровне команд, а в следующем — параллелизм на уровне процессоров.

2.1.5. Конвейеры

Уже много лет известно, что главным препятствием высокой скорости выполнения команд является их вызов из памяти. Для разрешения этой проблемы разработчики придумали средство для вызова команд из памяти заранее, чтобы они имелись в наличии в тот момент, когда будут необходимы. Эти команды помещались в набор регистров, который назывался буфером выборки с упреждением. Таким образом, когда была нужна определенная команда, она вызывалась прямо из буфера и не нужно было ждать, пока она считается из памяти. Эта идея использовалась еще при разработке IBM Stretch, который был сконструирован в 1959 г.

В действительности процесс выборки с упреждением подразделяет выполнение команды на два этапа: вызов и собственно выполнение. Идея конвейера еще больше продвинула эту стратегию вперед. Теперь команда подразделялась уже не на два, а на несколько этапов, каждый из которых выполнялся определенной частью аппаратного обеспечения, причем все эти части могли работать параллельно.

На рисунке 2.3*а* изображен конвейер из пяти блоков, которые называются стадиями. Стадия С1 вызывает команду из памяти и помещает ее в буфер, где она хранится до тех пор, пока не станет ненужной. Стадия С2 декодирует эту команду, определяя ее тип и тип операндов, над которыми она будет производить определенные действия. Стадия С3 определяет местонахождение операндов и вызывает их или из регистров, или из памяти. Стадия С4 выполняет команду — обычно путем провода операндов через тракт данных (рис. 2.2). И наконец, стадия С5 записывает результат обратно в нужный регистр.

На рисунке 2.3*б* мы видим, как действует конвейер во времени. Во время цикла 1 стадия С1 работает над командой 1, вызывая ее из памяти. Во время цикла 2 стадия С2 декодирует команду 1, в то время как стадия С1 вызывает из памяти команду 2. Во время цикла 3 стадия С3 вызывает операнды для команды 1, стадия С2 декодирует команду 2, а стадия С1 вызывает третью команду. Во время цикла 4 стадия С4 выполняет команду 1, С3 вызывает операнды для команды 2, С2 декодирует команду 3, а С1 вызывает команду 4. Наконец, во время пятого цикла С5 записывает результат выполнения команды 1 обратно в регистр, тогда как другие стадии работают над следующими командами.

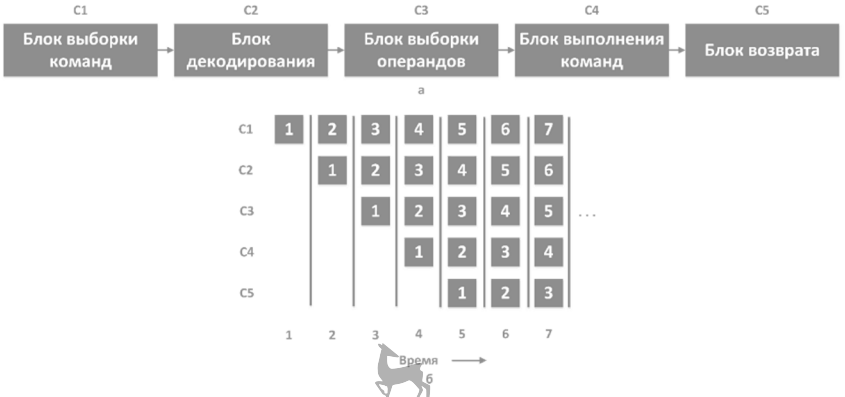


Рис. 2.3. Конвейер из пяти стадий (*а*); состояние каждой стадии в зависимости от количества пройденных циклов (*б*). Показано 7 циклов

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

Конвейеры позволяют найти компромисс между временем ожидания (сколько времени занимает выполнение одной команды) и пропускной способностью процессора (сколько миллионов команд в секунду выполняет процессор). Если время цикла составляет T нс, а конвейер содержит n стадий, то время ожидания составит $n \cdot T$ нс, а пропускная способность — $1000/T$ млн команд в секунду.

2.2. Суперскалярная архитектура

Одна из возможных схем процессора с двойным конвейером показана на рисунке 2.4. В основе разработки лежит конвейер, изображенный на рисунке 2.3. Здесь общий отдел вызова команд берет из памяти сразу по две команды и помещает каждую из них в один из конвейеров. Каждый конвейер содержит АЛУ для параллельных операций. Чтобы выполняться параллельно, две команды не должны конфликтовать при использовании ресурсов (например, регистров) и ни одна из них не должна зависеть от ре-

зультата выполнения другой. Как и в случае с одним конвейером, либо компилятор должен следить, чтобы не возникало неприятных ситуаций (например, когда аппаратное обеспечение выдает некорректные результаты, если команды несовместимы), либо же конфликты выявляются и устраняются прямо во время выполнения команд благодаря использованию дополнительного аппаратного обеспечения.

Рис. 2.4. Двойной конвейер из пяти стадий с общим отделом вызова команд

Сначала конвейеры (как двойные, так и одинарные) использовались только в компьютерах RISC. У 386-го и его предшественников их не было, однако необходимо отметить, что параллельное функционирование отдельных блоков процессора использовалось и в 386-м микропроцессоре — оно стало прообразом 5-стадийного конвейера микропроцессора 486. Конвейеры в процессорах компании *Intel* появились в 486-й модели. 486-й процессор содержал один конвейер, а Pentium — два конвейера из пяти стадий. Похожая схема изображена на рисунке 2.4, но разделение функций между второй и третьей стадиями (они назывались декодирование 1 и декодирование 2) было немного другим. Главный конвейер (*u*-конвейер) мог выполнять произвольные команды. Второй конвейер (*v*-конвейер) мог выполнять только простые команды с целыми числами, а также одну простую команду с плавающей точкой (FXCH).

Имеются сложные правила определения, является ли пара команд совместимой для того, чтобы выполняться параллельно. Если команды, входящие в пару, были сложными или несовместимыми, выполнялась только одна из них (в *u*-конвейере). Оставшаяся вторая команда составляла затем пару со следующей командой. Команды всегда выполнялись по порядку. Таким образом, Pentium содержал особые компиляторы, которые объединяли совместимые команды в пары и могли порождать программы, выполняющиеся быстрее, чем в предыдущих версиях. Измерения показали, что программы, производящие операции с целыми числами, на компьютере

Pentium выполняются почти в два раза быстрее, чем на 486-м, хотя у него такая же тактовая частота. Вне всяких сомнений, преимущество в скорости появилось благодаря второму конвейеру.

Переход к четырем конвейерам возможен, но это потребовало бы создания громоздкого аппаратного обеспечения. Вместо этого используется другой подход. Основная идея — один конвейер с большим количеством функциональных блоков, как показано на рисунке 2.5. Pentium II, к примеру, имеет сходную структуру. В 1987 г. для обозначения этого подхода был введен термин «суперскалярная архитектура». Однако подобная идея нашла воплощение еще более 30 лет назад в компьютере CDC 6600. CDC 6600 вызывал команду из памяти каждые 100 нс и помещал ее в один из десяти функциональных блоков для параллельного выполнения. Пока команды выполнялись, центральный процессор вызывал следующую команду.

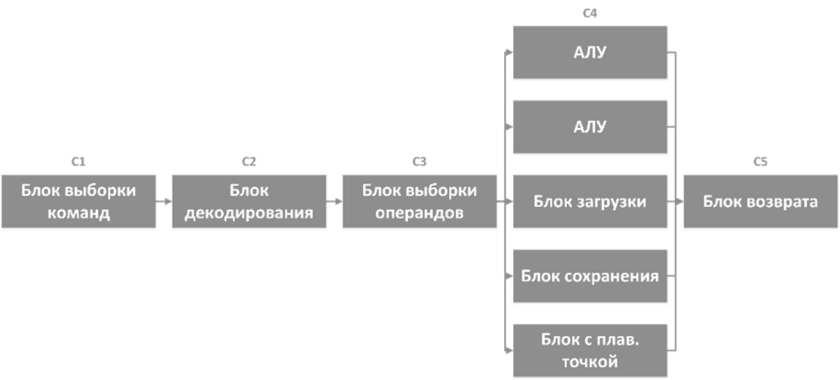


Рис. 2.5. Суперскалярный процессор с пятью функциональными блоками

Стадия 3 выпускает команды значительно быстрее, чем стадия 4 способна их выполнять. Если бы стадия 3 выпускала команду каждые 10 нс, а все функциональные блоки выполняли бы свою работу также за 10 нс, то на четвертой стадии всегда функционировал бы только один блок, что сделало бы саму идею конвейера бессмысленной. В действительности большинству функциональных блоков четвертой стадии для выполнения команды требуется значительно больше времени, чем занимает один цикл (это блоки доступа к памяти и блок выполнения операций с плавающей точкой). Как видно из рисунка 2.5, на четвертой стадии может быть несколько АЛУ.

2.2.1. Основная память: бит и адрес

Память — часть компьютера, где хранятся программы и данные. Можно также употреблять термин «запоминающее устройство». Без памяти, откуда процессоры считывают и куда записывают информацию, не было бы цифровых компьютеров со встроенными программами.

Основной единицей памяти является двоичный разряд, который называется битом. Бит может содержать 0 или 1. Это самая маленькая единица памяти. Цифровая информация может храниться благодаря различию между разными величинами какой-либо физической характеристики (например, напряжения или тока). Чем больше величин, которые нужно различать, тем меньше различий между смежными величинами и тем менее надежна память. Двоичная система требует различения всего двух величин, следовательно, это самый надежный метод кодирования цифровой информации.

Некоторые компьютеры (например, большие IBM) используют и десятичную, и двоичную арифметику. Здесь применяется так называемый двоично-десятичный код. Для хранения одного десятичного разряда используется 4 бита. Эти 4 бита дают 16 (2^4) комбинаций для размещения десяти различных значений (от 0 до 9). При этом шесть оставшихся комбинаций не используются. 16 битов в двоично-десятичном формате могут хранить числа от 0 до 9999, т. е. всего 10 000 различных комбинаций, а 16 битов в двоичном формате — 65 536 (2^{16}) комбинаций. Именно по этой причине говорят, что двоичная система эффективнее.

Если бы было создано очень надежное электронное устройство, которое могло бы хранить разряды от 0 до 9, разделив участок напряжения от 0 до 10 В на десять интервалов, то четыре таких устройства могли бы хранить десятичное число от 0 до 9999, т. е. 10 000 комбинаций. А если бы те же устройства использовались для хранения двоичных чисел, они могли бы содержать всего 16 комбинаций. Естественно, в этом случае десятичная система была бы более эффективной.

Память состоит из ячеек, каждая из которых может хранить некоторую порцию информации. Каждая ячейка имеет номер, который называется адресом. По адресу программы могут ссылаться на определенную ячейку. Если память содержит n ячеек, они будут иметь адреса от 0 до (n – 1). Все

ячейки памяти содержат одинаковое число битов. Если ячейка состоит из k битов, она может содержать любую из 2^k комбинаций.

В компьютерах, где используется двоичная система счисления (включая восьмеричное и шестнадцатеричное представление двоичных чисел), адреса памяти также выражаются в двоичных числах. Если адрес состоит из m битов, максимальное число адресованных ячеек будет составлять 2^m. Число битов в адресе определяет максимальное количество адресованных ячеек памяти и не зависит от числа битов в ячейке. 12-битные адреса нужны и памяти с 212 ячейками по 8 битов каждая, и памяти с 212 ячейками по 64 бита каждая.

Ячейка — минимальная единица, к которой можно обращаться. В последние годы практически все производители выпускают компьютеры с 8-битными ячейками, которые называются байтами. Байты группируются в слова. Компьютер с 32-битными словами имеет 4 байта на каждое слово, а компьютер с 64-битными словами — 8 байтов на каждое слово. Такая единица, как слово, необходима, поскольку большинство команд производят операции над целыми словами (например, складывают два слова). Таким образом, 32-битная машина будет содержать 32-битные регистры и команды для манипуляций с 32-битными словами, тогда как 64-битная машина будет иметь 64-битные регистры и команды для перемещения, сложения, вычитания и других операций над 64-битными словами.

2.2.2. Код с исправлением ошибок

Память компьютера время от времени может делать ошибки из-за всплесков напряжения на линии электропередачи и по другим причинам. Чтобы бороться с такими ошибками, используются коды с обнаружением и исправлением ошибок. При этом к каждому слову в памяти особым образом добавляются дополнительные биты.

Когда слово считывается из памяти, эти биты проверяются на наличие ошибок. Чтобы понять, как обращаться с ошибками, необходимо внимательно изучить, что представляют собой эти ошибки. Предположим, что слово состоит из m битов данных, к которым мы прибавляем r дополнительных битов (контрольных разрядов). Пусть общая длина слова будет n, т. е. $n = m + r$. Таким образом, n-битную единицу, содержащую m битов данных и r контрольных разрядов, часто называют кодированным словом.

Для любых двух кодированных слов, например 10001001 и 10110001, можно определить, сколько соответствующих битов в них различается. В данном примере таких бита три. Чтобы определить количество различающихся битов, нужно над двумя кодированными словами произвести логическую операцию «ИСКЛЮЧАЮЩЕЕ ИЛИ» и сосчитать число битов со значением 1 в полученном результате. Число битовых позиций, по которым различаются два слова, называется интервалом Хэмминга. Если интервал Хэмминга для двух слов равен d, это значит, что достаточно d битовых ошибок, чтобы превратить одно слово в другое. Например, интервал Хэмминга кодированных слов 11110001 и 00110000 равен 3, поскольку для превращения первого слова во второе достаточно трех ошибок в битах.

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

Свойства проверки и исправления ошибок определенного кода зависят от его интервала Хэмминга. Чтобы обнаружить d ошибок в битах, необходим код с интервалом $(d + 1)$, поскольку d ошибок не могут изменить одно допустимое кодированное слово на другое допустимое кодированное слово. Соответственно, чтобы исправить d ошибок в битах, необходим код с интервалом $(2d + 1)$, поскольку в этом случае допустимые кодированные слова так сильно отличаются друг от друга, что даже если произойдет d изменений, начальное кодированное слово будет ближе к ошибочному, чем любое другое кодированное слово, поэтому его без труда можно будет определить.

В качестве простого примера кода с обнаружением ошибок рассмотрим код, в котором к данным присоединяется один бит четности. Бит четности выбирается таким образом, что число битов со значением 1 в кодированном слове четное (или нечетное). Интервал этого кода равен 2, поскольку любая

ошибка в битах приводит к кодированному слову с неправильной четностью. Другими словами, достаточно двух ошибок в битах для перехода от одного допустимого кодированного слова к другому допустимому слову. Такой код может использоваться для обнаружения одиночных ошибок. Если из памяти считывается слово, содержащее неверную четность, поступает сигнал об ошибке. Программа не сможет продолжаться, но зато не будет неверных результатов.

Представим, что мы хотим разработать код с m битами данных и r контрольных и разрядов, который позволил бы исправлять все ошибки в битах. Каждое из 2^r допустимых слов имеет n недопустимых кодированных слов, которые отличаются от допустимого одним битом. Они образуются инвертированием каждого из n битов в n-битном кодированном слове. Следовательно, каждое из 2^r допустимых слов требует $(n + 1)$ возможных сочетаний битов, приписываемых этому слову (n возможных ошибочных вариантов и один правильный). Поскольку общее число различных сочетаний битов равно 2^n, $(n + 1) \cdot 2^m < 2^n$. Так как $n = (m + r)$, $(m + r + 1) < 2^r$. Эта формула дает нижний предел числа контрольных разрядов, необходимых для исправления одиночных ошибок.

2.2.3. Кэш-память

Процессоры всегда работали быстрее, чем память. Процессоры и память совершенствовались параллельно, поэтому это несоответствие сохранялось. Поскольку на микросхему можно помещать все больше и больше транзисторов, разработчики процессоров использовали эти преимущества для создания конвейеров и суперскалярной архитектуры, что еще больше повышало скорость работы процессоров.

Разработчики памяти обычно использовали новые технологии для увеличения емкости, а не скорости, что еще больше усугубляло проблему. На практике такое несоответствие в скорости работы приводит к следующему: после того как процессор дает запрос памяти, должно пройти много циклов, прежде чем он получит слово, которое ему нужно. Чем медленнее работает память, тем дольше процессору приходится ждать, тем больше циклов должно пройти.

Есть два пути решения этой проблемы. Самый простой из них — начать считывать информацию из памяти, когда это необходимо, и при этом про-

должать выполнение команд, но если какая-либо команда попытается использовать слово до того, как оно считалось из памяти, процессор должен приостанавливать работу. Чем медленнее работает память, тем чаще будет возникать такая проблема и тем больше будет проигрыш в работе. Например, если отсрочка составляет десять циклов, весьма вероятно, что одна из десяти следующих команд попытается использовать слово, которое еще не считалось из памяти.

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

На самом деле эта проблема не технологическая, а экономическая. Инженеры знают, как построить память, которая будет работать так же быстро, как и процессор, но при этом ее приходится помещать прямо на микросхему процессора (поскольку информация через шину поступает очень медленно). Установка большой памяти на микросхему процессора делает его больше и, следовательно, дороже, и даже если бы стоимость не имела значения, все равно существуют ограничения в размерах процессора, который можно сконструировать. Таким образом, приходится выбирать между быстрой памятью небольшого размера и медленной памятью большого размера. Мы бы предпочли память большого размера с высокой скоростью работы по низкой цене.

Существуют технологии сочетания маленькой и быстрой памяти с большой и медленной, что позволяет получить и высокую скорость работы, и большую емкость по разумной цене. Маленькая память с высокой скоростью работы называется кэш-памятью.

Основная идея кэш-памяти проста: в ней находятся слова, которые чаще всего используются. Если процессору нужно какое-нибудь слово, сначала он обращается к кэш-памяти. Только в том случае, если слова там нет, он обращается к основной памяти. Если значительная часть слов находится в кэш-памяти, среднее время доступа значительно сокращается.

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

То, что при последовательных отсылках к памяти в течение некоторого промежутка времени используется только небольшой ее участок, называется принципом локальности. Этот принцип составляет основу всех систем кэш-памяти. Идея состоит в следующем: когда определенное слово вызывается из памяти, оно вместе с соседними словами переносится в кэш-память, что позволяет при очередном запросе быстро обращаться к следующим словам. Общее устройство процессора, кэш-памяти и основной памяти показано на рисунке 2.6. Если слово считывается или записывается *k* раз, компьютеру понадобится сделать одно обращение к медленной основной памяти и $(k - 1)$ обращений к быстрой кэш-памяти. Чем больше *k*, тем выше общая производительность.

Рис. 2.6. Расположение кэш-памяти

В некоторых системах обращение к основной памяти может начинаться параллельно с исследованием кэш-памяти, чтобы в случае неудачного поиска цикл обращения к основной памяти уже начался. Однако эта стратегия требует способности останавливать процесс обращения к основной памяти в случае результативного обращения к кэш-памяти, что делает разработку такого компьютера более сложной.

Основная память и кэш-память делятся на блоки фиксированного размера с учетом принципа локальности. Блоки внутри кэш-памяти обычно называют строками кэш-памяти (*cache lines*). Если обращение к кэш-памяти нерезультативно, из основной памяти в кэш-память загружается вся строка, а не только необходимое слово. Например, если строка состоит из 64 байтов, обращение к адресу 260 повлечет за собой загрузку в кэш-память всей строки, т. е. с 256-го по 319-й байт.

Возможно, через некоторое время понадобятся другие слова из этой строки. Такой путь обращения к памяти более эффективен, чем вызов каждого слова по отдельности, потому что вызвать *k* слов один раз можно гораздо быстрее, чем одно слово *k* раз. Если входные сообщения кэш-памяти содержат более одного слова, это значит, что будет меньше таких входных сообщений и, следовательно, меньше непроизводительных затрат.

Разработка кэш-памяти очень важна для процессоров с высокой производительностью.

Первый вопрос — размер кэш-памяти. Чем больше размер, тем лучше работает память, но тем дороже она стоит.

Второй вопрос — размер строки кэш-памяти. Кэш-память объемом 16 Кбайт можно разделить на 1К строк по 16 байтов, 2К строк по 8 байтов и т. д.

Третий вопрос — как устроена кэш-память, т. е. как она определяет, какие именно слова содержатся в ней в данный момент.

Четвертый вопрос — должны ли команды и данные находиться вместе в общей кэш-памяти. Проще разработать смежную кэш-память**,** в которой хранятся и данные, и команды. При этом вызов команд и данных автоматически уравновешивается. Тем не менее в настоящее время существует тенденция к использованию разделенной кэш-памяти**,** когда команды хранятся в одной кэш-памяти, а данные — в другой. Такая структура также называется Гарвардской (*Harvard Architecture*), поскольку идея использования отдельной памяти для команд и отдельной памяти для данных впервые воплотилась в компьютере Mark III, который был создан Говардом Айкеном в Гарварде. Современные разработчики пошли по этому пути, поскольку сейчас широко используются процессоры с конвейерами, а при такой организации должна быть возможность одновременного доступа и к командам, и к данным (операндам). Разделенная кэш-память поз-

воляет осуществлять параллельный доступ, а общая — нет. К тому же, поскольку команды обычно не меняются во время выполнения, содержание командной кэш-памяти никогда не приходится записывать обратно в основную память.

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

3. ЦИФРОВАЯ ЛОГИКА

В самом низу иерархической схемы устройства компьютера находится цифровой логический уровень, или аппаратное обеспечение компьютера. Различные аспекты цифровой логики находятся на границе информатики и электротехники:

– основные элементы, из которых конструируются цифровые компьютеры;

– специальная двузначная алгебра (булева алгебра), которая используется при конструировании этих элементов;

– основные схемы, которые можно построить из вентилей в различных комбинациях, в том числе схемы для выполнения арифметических действий;

– наличие возможностей по комбинированию вентилей для хранения информации, т. е. организация устройства памяти;

– процессоры и то, как они на одной микросхеме обмениваются информацией с памятью и периферийными устройствами.

3.1. Вентили, булева алгебра и их реализация

Цифровые схемы могут конструироваться из небольшого числа простых элементов путем сочетания этих элементов в различных комбинациях. В следующих разделах будет дано описание этих основных элементов, показано, как их можно сочетать, а также введен математический метод, который можно использовать при анализе их работы.

3.1.1. Вентили

Цифровая схема — это схема, в которой есть только два логических значения. Обычно сигнал от 0 до 1 В представляет одно значение (например, 0), а сигнал от 2 до 5 В — другое значение (например, 1). Напряжение за пределами указанных величин недопустимо. Крошечные электронные устройства, которые называются вентилями**,** могут вычислять различные функции от этих двузначных сигналов. Эти вентили формируют основу аппаратного обеспечения, на которой строятся все цифровые компьютеры.

Описание принципов работы вентилей относится к уровню физических устройств, который находится ниже уровня 0. Кратко рассмотрим основной их принцип, который не так уж и сложен. Вся современная цифровая логика

основывается на том, что транзистор может работать как очень быстрый бинарный переключатель. На рисунке 3.1*а* изображен биполярный транзистор, встроенный в простую схему. Транзистор имеет три соединения с внешним миром: коллектор, базу и эмиттер. Если входное напряжение V_{in} ниже определенного критического значения, транзистор выключается и действует как очень большое сопротивление. Это приводит к выходному сигналу V_{out}, близкому к V_{cc} (напряжению, подаваемому извне), обычно +5 В для данного типа транзистора. Если V_{in} превышает критическое значение, транзистор включается и действует как провод, вызывая заземление сигнала V_{out} (по соглашению 0 В).

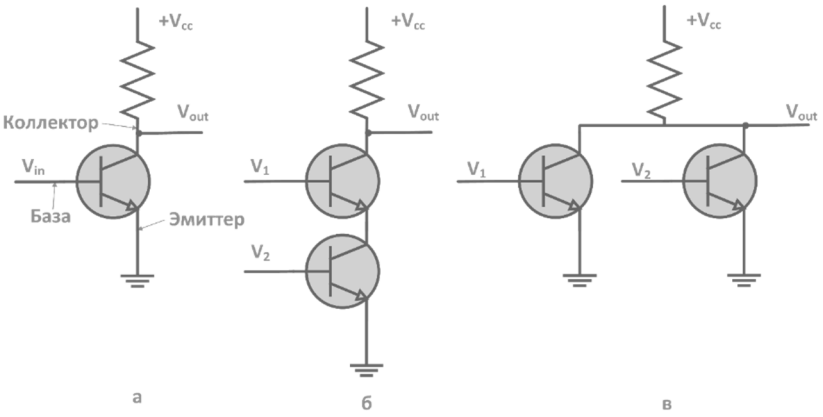


Рис. 3.1. Транзисторный инвертор (*а*); вентиль «НЕ-И» (*б*); вентиль «НЕ-ИЛИ» (*в*)

Важно отметить, что если напряжение V_{in} низкое, то V_{out} высокое, и наоборот. Эта схема, таким образом, является инвертором, превращающим логический 0 в логическую 1 и логическую 1 в логический 0. Резистор (ломаная линия) нужен для ограничения количества тока, проходящего через транзистор, чтобы транзистор не сгорел. На переключение с одного состояния на другое обычно требуется несколько наносекунд.

На рисунке 3.1*б* два транзистора соединены последовательно. Если и напряжение V_1, и напряжение V_2 высокое, то оба транзистора будут служить проводниками и снижать V_{out}. Если одно из входных напряжений низкое, то соответствующий транзистор будет выключаться и напряжение на выходе

будет высоким. Другими словами, V_{out} будет низким тогда и только тогда, когда и напряжение V_1, и напряжение V_2 высокие.

На рисунке 3.1*в* два транзистора соединены параллельно. Если один из входных сигналов высокий, будет включаться соответствующий транзистор и снижать выходной сигнал. Если оба напряжения на входе низкие, то выходное напряжение будет высоким.

Эти три схемы образуют три простейших вентиля. Они называются вентилями «НЕ», «НЕ-И» и «НЕ-ИЛИ». Вентили «НЕ» часто называют инверторами. Мы будем использовать оба термина. Если мы примем соглашение, что высокое напряжение (V_{cc}) — это логическая 1, а низкое напряжение («земля») — логический 0, то сможем выражать значение на выходе как функцию от входных значений. Значки, которые используются для изображения этих трех типов вентилей, показаны на рисунке 3.2*а–в.* Там же приводится поведение функции для каждой схемы.

На этих рисунках A и B — это входные сигналы, а X — выходной сигнал. Каждая строка таблицы определяет выходной сигнал для различных комбинаций входных сигналов.

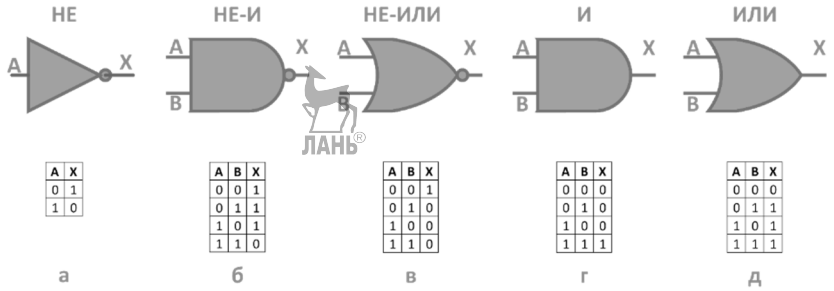


Рис. 3.2. Значки для изображения пяти основных вентилей. Поведение функции для каждого вентиля

Если выходной сигнал (рис. 3.2*б*) подать в инвертор, мы получим другую схему, противоположную вентилю «НЕ-И», т. е. такую схему, у которой выходной сигнал равен 1 тогда и только тогда, когда оба входных сигнала равны 1. Такая схема называется вентилем «И»; ее схематическое изображение и описание соответствующей функции даны на рисунке 3.2*г.* Точно так же вентиль «НЕ-ИЛИ» может быть связан с инвертором. Тогда получится схема, у которой выходной сигнал равен 1 в том случае, если хотя бы

один из входных сигналов — 1, и равен 0, если оба входных сигнала равны 0. Изображение этой схемы, которая называется вентилем «ИЛИ», а также описание соответствующей функции даны на рисунке 3.2*д*. Маленькие кружочки в схемах инвертора, вентиля «НЕ-И» и вентиля «НЕ-ИЛИ» называются инвертирующими выходами. Они также могут использоваться и в другом контексте для указания на инвертированный сигнал.

Пять вентилей, изображенных на рисунке 3.2, составляют основу цифрового логического уровня. Вентили «НЕ-И» и «НЕ-ИЛИ» требуют два транзистора каждый, а вентили «И» и «ИЛИ» — три транзистора каждый. По этой причине во многих компьютерах используются вентили «НЕ-И» и «НЕ-ИЛИ», а не «И» и «ИЛИ» (на практике все вентили выполняются несколько по-другому, но «НЕ-И» и «НЕ-ИЛИ» все равно проще, чем «И» и «ИЛИ»). Вентили могут иметь более двух входов. В принципе вентиль «НЕ-И», например, может иметь произвольное количество входов, но на практике больше восьми обычно не бывает.

Существуют две основные производственные технологии вентилей — биполярная и МОП (металл — оксид — полупроводник). Среди биполярных технологий можно назвать ТТЛ (транзисторно-транзисторную логику), которая служила основой цифровой электроники на протяжении многих лет, и ЭСЛ (эмиттерно-связанную логику), которая используется в тех случаях, когда требуется высокая скорость выполнения операций.

Вентили МОП работают медленнее, чем ТТЛ и ЭСЛ, но потребляют гораздо меньше энергии и занимают гораздо меньше места, поэтому можно компактно расположить большое количество таких вентилей. Вентили МОП имеют несколько разновидностей: *p*-канальный МОП-прибор, *n*-канальный МОП-прибор и комплементарный МОП. Хотя МОП-транзисторы конструируются не так, как биполярные транзисторы, они обладают такой же способностью функционировать, как и электронные переключатели. Современные процессоры и память чаще всего производятся с использованием технологии комплементарных МОП, которая работает при напряжении +3,3 В.

3.1.2. Булева алгебра

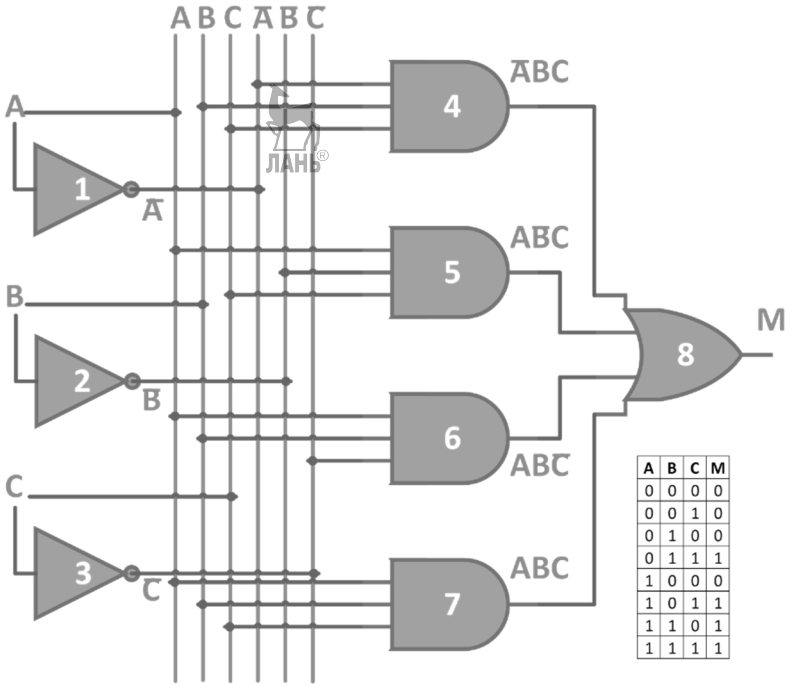
Чтобы описать схемы, которые строятся путем сочетания различных вентилей, нужен особый тип алгебры, в которой все переменные и функции могут принимать только два значения: 0 и 1. Такая алгебра называется булевой алгеброй. Она названа в честь английского математика Джорджа Буля

(1815–1864). На самом деле в данном случае мы говорим об особом типе булевой алгебры, а именно об *алгебре релейных схем*, но термин «булева алгебра» очень часто используется в значении «алгебра релейных схем», поэтому мы не будем их различать.

Как и в обычной алгебре (т. е. в той, которую изучают в школе), в булевой алгебре есть свои функции. Булева функция имеет одну или несколько переменных и выдает результат, который зависит только от значений этих переменных. Можно определить простую функцию f, сказав, что $f(A) = 1$, если $A = 0$, и $f(A) = 0$, если $A = 1$. Такая функция будет функцией «НЕ».

Так как булева функция от n переменных имеет только 2^n возможных комбинаций значений переменных, то такую функцию можно полностью описать в таблице с 2^n строками. В каждой строке будет даваться значение функции для разных комбинаций значений переменных. Такая таблица называется таблицей истинности. Все таблицы на рисунке 3.2 представляют собой таблицы истинности. Если мы договоримся всегда располагать строки таблицы истинности по порядку номеров, т. е. для двух переменных в порядке 00, 01, 10, 11, то функцию можно полностью описать 2^n-битным двоичным числом, которое получается, если считывать по вертикали колонку результатов в таблице истинности. Таким образом, «НЕ-И» — это 1110, «НЕ-ИЛИ» — 1000, «И» — 0001 и «ИЛИ» — 0111. Очевидно, что существует только 16 булевых функций от двух переменных, которым соответствуют 16 возможных 4-битных цепочек. В обычной алгебре, напротив, есть бесконечное число функций от двух переменных, и ни одну из них нельзя описать, дав таблицу значений этой функции для всех возможных значений переменных, поскольку каждая переменная может принимать бесконечное число значений.

На рисунке 3.3 показана таблица истинности для булевой функции от трех переменных $M = f(A, B, C)$. Это функция большинства, которая принимает значение 0, если большинство переменных равно 0, и 1, если большинство переменных равно 1. Хотя любая булева функция может быть определена с помощью таблицы истинности, с возрастанием количества переменных такой тип записи становится громоздким. Поэтому вместо таблиц истинности часто используется другой тип записи.



A	B	C	M
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Рис. 3.3. Схема для функции большинства от трех переменных и таблица истинности для этой функции

Чтобы увидеть, каким образом осуществляется этот другой тип записи, отметим, что любую булеву функцию можно определить, указав, какие комбинации значений переменных дают значение функции 1. Для функции, приведенной на рисунке 3.3, существует четыре комбинации переменных, которые дают значение функции 1. Мы будем рисовать черту над переменной, чтобы показать, что ее значение инвертируется. Отсутствие черты означает, что значение переменной не инвертируется. Кроме того, мы будем использовать знак умножения (точку) для обозначения булевой функции «И» (знак умножения может опускаться) и «+» для обозначения булевой функции «ИЛИ». Например, $A\bar{B}C$ принимает значение 1, только если $A = 1$, $B = 0$ и $C = 1$; $(AB + BC)$ принимает значение 1, только если $A = 1$ и $B = 0$ или $B = 1$ и $C = 0$. В таблице на рисунке 3.3 функция принимает значение 1 в четырех строках: $\bar{A}BC$, $A\bar{B}C$, $AB\bar{C}$ и ABC. Функция M принимает значение

истины (т. е. 1), если одно из этих четырех условий истинно. Следовательно, мы можем написать $M = \bar{A}BC + A\bar{B}C + AB\bar{C} + ABC$.

Это компактная запись таблицы истинности. Таким образом, функцию от n переменных можно описать суммой максимум 2^n произведений, при этом в каждом произведении будет по n множителей. Как мы скоро увидим, такая формулировка особенно важна, поскольку она ведет прямо к реализации данной функции с использованием стандартных вентилей.

Важно понимать различие между абстрактной булевой функцией и ее реализацией с помощью электронной схемы. Булева функция состоит из переменных, например A, B и C, и операторов «И», «ИЛИ» и «НЕ». Булева функция описывается с помощью таблицы истинности или специальной записи, например $F = A\bar{B}C + ABC$.

Булева функция может реализовываться с помощью электронной схемы (часто различными способами) с использованием сигналов, которые представляют входные и выходные переменные, и вентилей, например, «И», «ИЛИ» и «НЕ».

3.1.3. Реализация булевых функций

Представление булевой функции в виде суммы максимум 2^n произведений делает возможной реализацию этой функции. На рисунке 3.3 можно увидеть, как это осуществляется. На рисунке 3.3 входные сигналы A, B и C показаны с левой стороны, а функция M, полученная на выходе, показана с правой стороны. Поскольку необходимы дополнительные величины (инверсии) входных переменных, они образуются путем провода сигнала через инверторы 1, 2 и 3. Чтобы сделать рисунок понятней, мы нарисовали шесть вертикальных линий, три из которых связаны с входными переменными, а три другие — с их инверсиями. Эти линии обеспечивают передачу входного сигнала к вентилям. Например, вентили 5, 6 и 7 в качестве входа используют A. В реальной схеме эти вентили, вероятно, будут непосредственно соединены проводом с A без каких-либо промежуточных вертикальных проводов.

Схема содержит четыре вентиля «И», по одному для каждого члена в уравнении для M (т. е. по одному для каждой строки в таблице истинности с результатом 1). Каждый вентиль «И» вычисляет одну из указанных строк

таблицы истинности. В конце концов все данные произведения суммируются (имеется в виду операция «ИЛИ») для получения конечного результата.

Если две линии на рисунке пересекаются, связь подразумевается только в том случае, если на пересечении указана жирная точка. Например, выход вентиля 3 пересекает все шесть вертикальных линий, но связан он только с *C*.

Из рисунка 3.3 должно быть ясно, как реализовать схему для любой булевой функции:

1) составить таблицу истинности для данной функции;

2) обеспечить инверторы, чтобы порождать инверсии для каждого входного сигнала;

3) нарисовать вентиль «И» для каждой строки таблицы истинности с результатом 1;

4) соединить вентили «И» с соответствующими входными сигналами;

5) вывести выходы всех вентилей «И» в вентиль «ИЛИ».

Мы показали, как реализовать любую булеву функцию с использованием вентилей «НЕ», «И» и «ИЛИ». Однако гораздо удобнее строить схемы с использованием одного типа вентилей. К счастью, можно легко преобразовать схемы, построенные по предыдущему алгоритму, в форму «НЕ-И» или «НЕ-ИЛИ». Чтобы осуществить такое преобразование, все, что нам нужно, — это способ воплощения «НЕ», «И» и «ИЛИ» с помощью одного типа вентилей. На рисунке 3.4 показано, как это можно сделать, используя только вентили «НЕ-И» или только вентили «НЕ-ИЛИ». Отметим, что существуют также другие способы подобного преобразования.

Для того чтобы реализовать булеву функцию с использованием только вентилей «НЕ-И» или только вентилей «НЕ-ИЛИ», можно сначала следовать алгоритму, описанному ранее, и сконструировать схему с вентилями «НЕ», «И» и «ИЛИ». Затем нужно заменить многовходовые вентили эквивалентными схемами с использованием двухвходовых вентилей. Например, $A + B + C + D$ можно поменять на $(A + B) + (C + D)$, используя три двухвходовых вентиля. Затем вентили «НЕ», «И» и «ИЛИ» заменяются схемами, изображенными на рисунке 3.4.

Такая процедура не приводит к оптимальным схемам с точки зрения минимального числа вентилей, однако она демонстрирует, что подобное преобразование осуществимо. Вентили «НЕ-И» и «НЕ-ИЛИ» считаются

полными, потому что можно вычислить любую булеву функцию, используя только вентили «НЕ-И» или только вентили «НЕ-ИЛИ». Ни один другой вентиль не обладает таким свойством, вот почему именно эти два типа вентилей предпочтительны при построении схем.

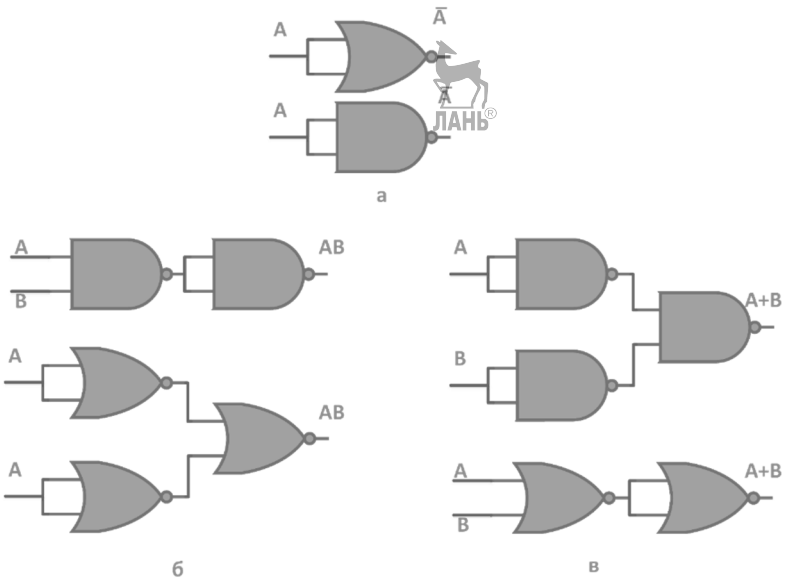


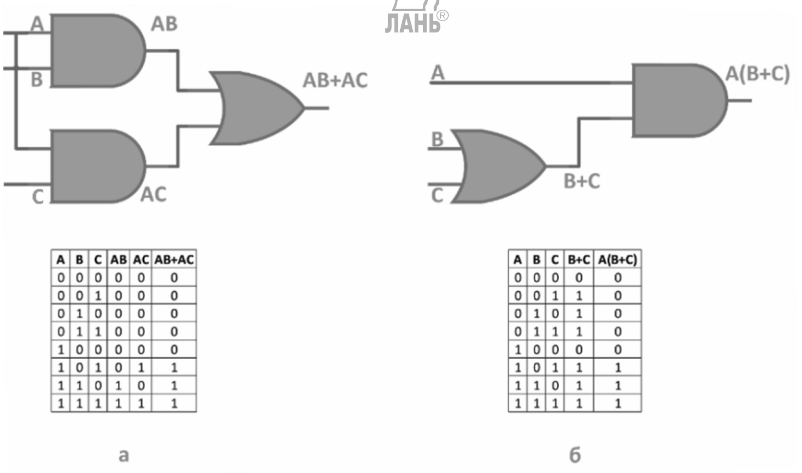
Рис. 3.4. Конструирование вентилей «НЕ» (*а*), «И» (*б*) и «ИЛИ» (*в*) с использованием только вентилей «НЕ-И» или только вентилей «НЕ-ИЛИ»

3.1.4. Эквивалентность схем

Разработчики схем часто стараются сократить число вентилей, чтобы снизить цену, уменьшить занимаемое схемой место, сократить потребление энергии и т. д. Для упрощения схемы ее следует скомпоновать иначе, причем так, чтобы новая схема смогла вычислять ту же функцию, но при этом требовала меньшего количества вентилей (или работала с более простыми вентилями, например — двухвходовыми вместо четырехвходовых). Булева алгебра является ценным инструментом в поиске эквивалентных схем.

В качестве примера использования булевой алгебры рассмотрим схему и таблицу истинности для ($AB + AC$) (рис. 3.5*а*). Многие правила обычной алгебры справедливы и для булевой алгебры. Например, выражение

($AB + AC$) может быть преобразовано в $A(B + C)$ с помощью дистрибутивного закона. На рисунке 3.5*б* показана схема и таблица истинности для $A(B + C)$. Две функции являются эквивалентными тогда и только тогда, когда обе функции принимают одно и то же значение для всех возможных переменных. Из таблиц истинности на рисунке 3.5 ясно видно, что $A(B + C)$ эквивалентно ($AB + AC$). Несмотря на эту эквивалентность, схема на рисунке 3.5*б* лучше, чем схема на рисунке 3.5*а*, поскольку она содержит меньше вентилей.



A	B	C	AB	AC	AB+AC
0	0	0	0	0	0
0	0	1	0	0	0
0	1	0	0	0	0
0	1	1	0	0	0
1	0	0	0	0	0
1	0	1	0	1	1
1	1	0	1	0	1
1	1	1	1	1	1

а

A	B	C	B+C	A(B+C)
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	1	0
1	0	0	0	0
1	0	1	1	1
1	1	0	1	1
1	1	1	1	1

б

Рис. 3.5. Две эквивалентные функции:
а — $AB + AC$; *б* — $A(B + C)$.

Обычно разработчик исходит из определенной булевой функции, а затем применяет к ней законы булевой алгебры, чтобы найти более простую функцию, эквивалентную исходной. На основе полученной функции можно конструировать схему.

Чтобы использовать данный подход, нам нужны некоторые равенства из булевой алгебры (табл. 3.1). Интересно отметить, что каждый закон имеет две формы. Одну форму из другой можно получить, меняя «И» на «ИЛИ» и 0 на 1. Все законы можно легко доказать, составив их таблицы истинности. Почти во всех случаях результаты очевидны, за исключением законов де Моргана, законов поглощения и дистрибутивного закона $A + BC = (A + B)(A + C)$.

Законы де Моргана распространяются на выражения с более чем двумя переменными, например $ABC = A + B + C$.

Таблица 3.1. Некоторые законы булевой алгебры

Название	«И»	«ИЛИ»
Законы тождества	$1A = A$	$0 + A = A$
Законы нуля	$0A = 0$	$1 + A = 1$
Законы идемпотентности	$AA = A$	$A + A = A$
Законы инверсии	$A\overline{A} = 0$	$A + \overline{A} = 1$
Коммуникативные законы	$AB = BA$	$A + B = B + A$
Ассоциативные законы	$(AB)C = A(BC)$	$(A + B) + C = A + (B + C)$
Дистрибутивные законы	$A + BC = (A + B)(A + C)$	$A(B + C) = AB + AC$
Законы поглощения	$A(A + B) = A$	$A + AB = A$
Законы де Моргана	$\overline{AB} = \overline{A} + \overline{B}$	$\overline{A} \pm \overline{B} = \overline{AB}$

Законы де Моргана предполагают альтернативную запись. На рисунке 3.6*а* форма «И» дается с отрицанием, которое показывается с помощью инвертирующих входов и выходов. Таким образом, вентиль «ИЛИ» с инвертированными входными сигналами эквивалентен вентилю «НЕ-И». Из рисунка 3.6*б*, на котором изображена вторая форма закона де Моргана, ясно, что вместо вентиля «НЕ-ИЛИ» можно нарисовать вентиль «И» с инвертированными входами. С помощью отрицания обеих форм закона де Моргана мы приходим к эквивалентным репрезентациям вентилей «И» и «ИЛИ» (см. рис. 3.6*в* и *г*). Аналогичные символические изображения существуют для различных форм закона де Моргана (например, *n*-входовый вентиль «НЕ-И» становится вентилем «ИЛИ» с инвертированными входами).

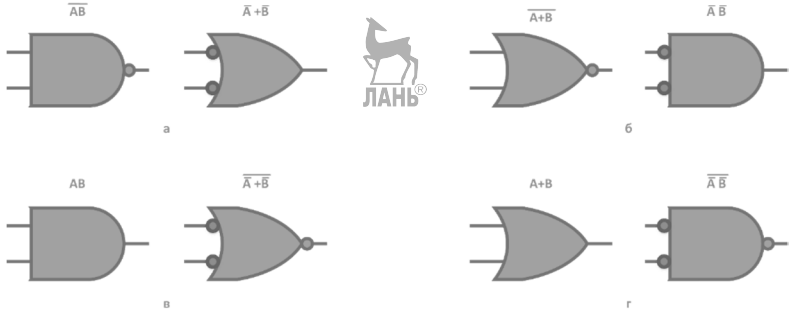
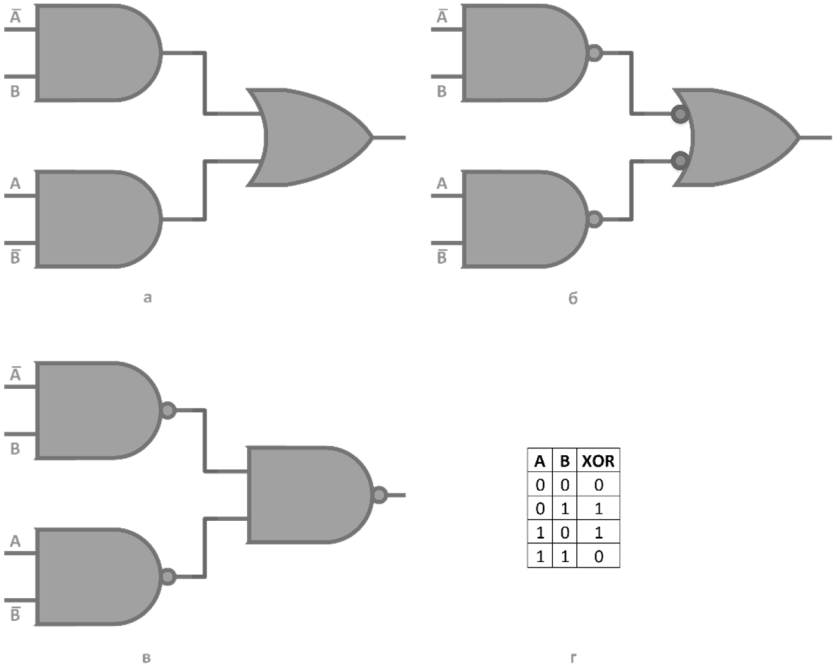


Рис. 3.6. Альтернативные обозначения вентилей «НЕ-И» (*а*); «НЕ-ИЛИ» (*б*); «И» (*в*); «ИЛИ» (*г*)

Используя уравнения, указанные на рисунке 3.6, и аналогичные уравнения для многовходовых вентилей, можно легко преобразовать сумму про-

изведений в чистую форму «НЕ-И» или чистую форму «НЕ-ИЛИ». В качестве примера рассмотрим функцию «ИСКЛЮЧАЮЩЕЕ ИЛИ» (рис. 3.7*а*). Стандартная схема, выражающая сумму произведений, показана на рисунке 3.7*б*. Чтобы перейти к форме «НЕ-И», нужно линии, соединяющие выходы вентилей «И» с входом вентиля «ИЛИ», нарисовать с инвертирующими входами и выходами, как показано на рисунке 3.7*в*. Затем, применяя рисунок 3.7*а*, мы приходим к рисунку 3.7*г*. Новые переменные *A* и *B* можно получить из существующих *A* и *B*, используя вентили «НЕ-И» или «НЕ-ИЛИ» с объединенными входами. Отметим, что инвертирующие входы (выходы) могут перемещаться вдоль линии по желанию, например от выходов входных вентилей к входам выходного вентиля.



A	B	XOR
0	0	0
0	1	1
1	0	1
1	1	0

Рис. 3.7. Схемы вычисления функции «ИСКЛЮЧАЮЩЕЕ ИЛИ» (*а–в*) и таблица истинности для этой функции (*г*)

Очень важно отметить, что один и тот же вентиль может вычислять разные функции в зависимости от используемых соглашений. На рисунке 3.8*а* мы показали выход определенного вентиля *F* для различных комбинаций входных сигналов.

A	B	F
0^V	0^V	0^V
0^V	5^V	0^V
5^V	0^V	0^V
5^V	5^V	5^V

а

A	B	F
0	0	1
0	1	0
1	0	0
1	1	0

б

A	B	F
1	1	1
1	0	1
0	1	1
0	0	0

в

Рис. 3.8. Электрические характеристики устройства (*а*); позитивная логика (*б*); негативная логика (*в*)

И входные, и выходные сигналы показаны в вольтах. Если мы примем соглашение, что 0 В — это логический ноль, а 3,3 В или 5 В — логическая единица, мы получим таблицу истинности, показанную на рисунке 3.8*б*, т. е. функцию «И». Такое соглашение называется позитивной логикой. Однако если мы примем негативную логику, т. е. условимся, что 0 В — это логическая единица, а 3,3 В или 5 В — логический ноль, то мы получим таблицу истинности, показанную на рисунке 3.8*в*, т. е. функцию «ИЛИ».

Таким образом, все зависит от того, какое соглашение выбрано для отображения вольт в логических величинах. В этой книге мы будем использовать позитивную логику. Случаи использования негативной логики будут оговариваться отдельно.

3.2. Тактовые генераторы

Во многих цифровых схемах все зависит от порядка, в котором выполняются действия. Иногда одно действие должно предшествовать другому, иногда два действия должны происходить одновременно. Для контроля временных отношений в цифровые схемы встраиваются тактовые генераторы, чтобы обеспечить синхронизацию. Тактовый генератор — это схема, которая вызывает серию импульсов. Все импульсы одинаковы по длительности. Интервалы между последовательными импульсами также одинаковы. Временной интервал между началом одного импульса и началом следующего называется временем такта. Частота импульсов обычно от 1 до 500 МГц, что соответствует времени такта от 1000 до 2 нс. Частота тактового генератора обычно контролируется кварцевым генератором, чтобы достичь высокой точности.

В компьютере за время одного такта может произойти много событий. Если они должны осуществляться в определенном порядке, то такт следует разделить на подтакты. Чтобы достичь лучшего разрешения, чем у основного тактового

генератора, нужно сделать ответвление от задающей линии тактового генератора и вставить схему с определенным временем задержки. Таким образом порождается вторичный сигнал тактового генератора, который сдвинут по фазе относительно первичного (рис. 3.9*а*). Временная диаграмма (рис. 3.9*б*) обеспечивает четыре начала отсчета времени для дискретных событий:

1) нарастающий фронт *C*1;

2) задний фронт *C*1;

3) нарастающий фронт *C*2;

4) задний фронт *C*2.

Связав различные события с различными фронтами, можно достичь требуемой последовательности выполнения действий. Если в пределах одного такта требуется более четырех начал отсчета, можно сделать еще несколько ответвлений от задающей линии с различным временем задержки.

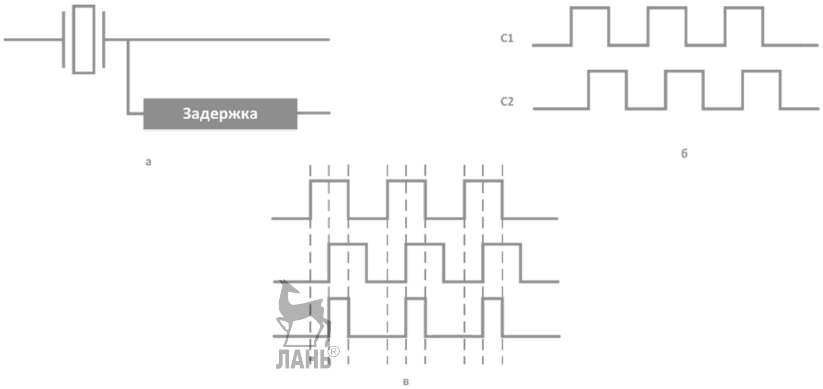


Рис. 3.9. Тактовый генератор (*а*); его временная диаграмма (*б*); порождение асинхронных тактовых импульсов (*в*)

В некоторых схемах важны временные интервалы, а не дискретные моменты времени. Например, некоторое событие может происходить в любое время, когда уровень импульса *C*1 высокий, а не на нарастающем фронте. Другое событие может происходить только в том случае, когда уровень импульса *C*2 высокий. Если необходимо более двух интервалов, нужно обеспечить больше линий передачи синхронизирующих импульсов или сделать так, чтобы состояния с высоким уровнем импульса у двух тактовых генераторов частично пересекались во времени. В последнем случае можно выделить четыре отдельных интервала.

Тактовые генераторы могут быть синхронными. В этом случае время состояния с высоким уровнем импульса равно времени состояния с низким уровнем импульса (рис. 3.9*б*). Чтобы получить асинхронную серию импульсов, нужно сдвинуть сигнал задающего генератора, используя цепь задержки. Затем нужно соединить полученный сигнал с изначальным сигналом с помощью логической функции «И» (см. рис. 3.9*в*).

3.3. Архитектура памяти

Память является необходимым компонентом любого компьютера. Без памяти не было бы компьютеров, по крайней мере таких, какие есть сейчас. Память используется как для хранения команд, которые нужно выполнить, так и для хранения данных. В следующих разделах мы рассмотрим основные компоненты памяти, начиная с уровня вентилей. Мы увидим, как они работают и как из них можно получить память большой емкости.

3.3.1. Триггеры (flip-flops)

Многие схемы выбирают значение на определенной линии в определенный момент времени и запоминают его. В такой схеме, которая называется триггером, переход состояния происходит не тогда, когда синхронизирующий сигнал равен 1, а во время перехода синхронизирующего сигнала с 0 на 1 (нарастающий фронт) или с 1 на 0 (задний фронт). Следовательно, длина синхронизирующего импульса не имеет значения, поскольку переходы происходят быстро.

Подчеркнем еще раз различие между триггером и защелкой. Триггер запускается фронтом сигнала, а защелка запускается уровнем сигнала. Обратите внимание, что в литературе эти термины часто путают. Многие авторы используют термин «триггер», когда речь идет о защелке, и наоборот.

Существует несколько подходов к разработке триггеров. Например, если бы существовал способ генерирования очень короткого импульса на нарастающем фронте синхронизирующего сигнала, этот импульс можно было бы подавать в *D*-защелку. В действительности такой способ существует (рис. 3.10).

Стандартные изображения защелок и триггеров показаны на рисунке 3.11. На рисунке 3.11*а* изображена защелка, состояние которой загружается тогда, когда синхронизирующий сигнал *СК* (от слова *clock*) равен 1, в противоположность защелке, изображенной на рисунке 3.11*б*, у которой

синхронизирующий сигнал обычно равен 1, но переходит на 0, чтобы загрузить состояние из *D*. На рисунке 3.11*в* и *г* изображены триггеры.

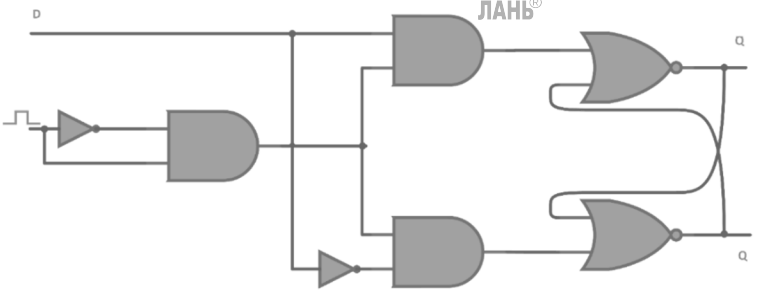


Рис. 3.10. *D*-триггер

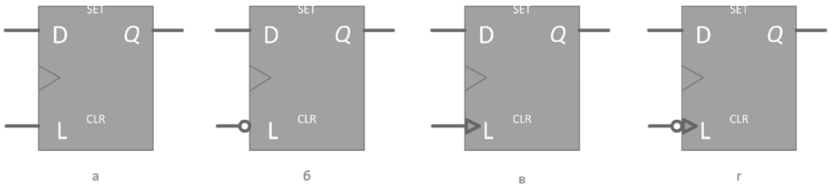


Рис. 3.11. *D*-защелки и *D*-триггеры

То, что это триггеры, а не защелки, показано с помощью уголка при синхронизирующем входе. Триггер на рисунке 3.11*в* изменяет состояние на возрастающем фронте синхронизирующего импульса (переход от 0 к 1), тогда как триггер на рисунке 3.11*г* изменяет состояние на заднем фронте (переход от 0 к 1). Многие (хотя не все) защелки и триггеры также имеют два дополнительных входа — Set (установка), или Preset (предварительная установка), и Reset (сброс), или Clear (очистка). Первый вход (Set или Preset) устанавливает $Q = 1$, а второй (Reset или Clear) — $Q = 0$.

3.3.2. Регистры

Существуют различные конфигурации триггеров. На рисунке 3.12*а* изображена схема, содержащая два независимых *D*-триггера с сигналами предварительной установки и очистки. Хотя эти два триггера находятся на одной микросхеме с 14 выводами, они не связаны между собой. Совершенно по-другому устроен восьмиразрядный триггер, изображенный на рисунке 3.12*б*. Здесь, в отличие от предыдущей схемы, у восьми триггеров нет выхода

J и линий предварительной установки и все синхронизирующие линии связаны вместе и управляются выводом 11. Сами триггеры — того же типа, что и на рисунке 3.12*г*, но инвертирующие входы и аннулируются инвертором, связанным с выводом 11, поэтому триггеры запускаются при переходе от 0 к 1. Все восемь сигналов очистки также объединены, поэтому, когда вывод 1 переходит в состояние 0, все триггеры также переходят в состояние 0. Если вам не понятно, почему вывод 11 инвертируется на входе, а затем инвертируется снова при каждом сигнале СК, то ответ прост: входной сигнал не имеет достаточной мощности, чтобы запустить все восемь триггеров; входной инвертор на самом деле используется в качестве усилителя.

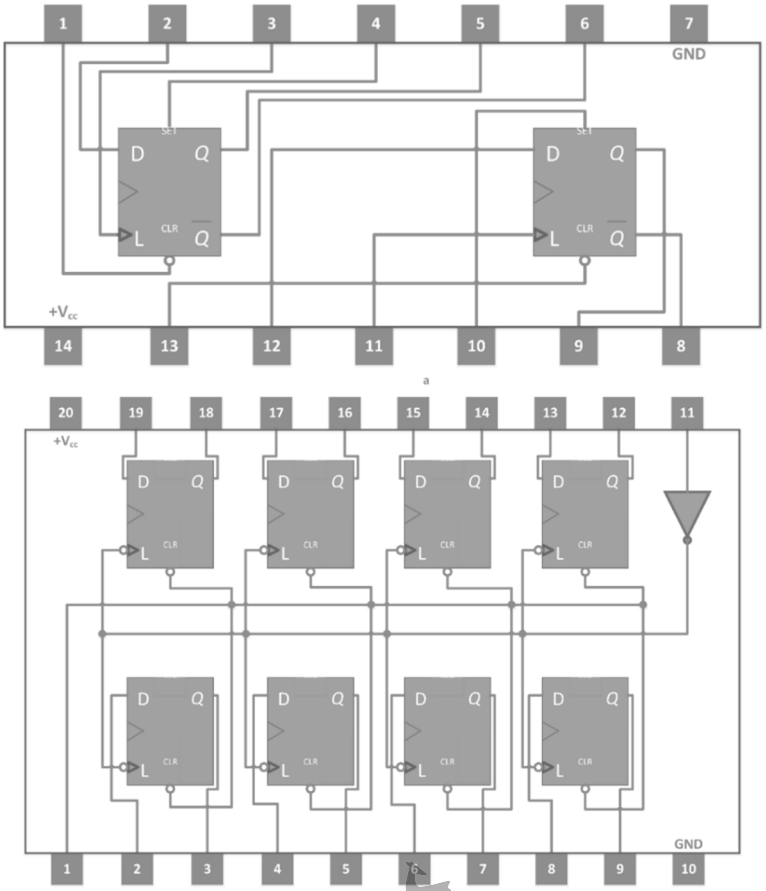


Рис. 3.12. Два *D*-триггера (*а*); восьмиразрядный триггер (*б*)

Одна из причин объединения линий синхронизации и линий очистки в микросхеме на рисунке 3.12*б* — экономия выводов. С другой стороны, микросхема данной конфигурации несколько отличается от восьми несвязанных триггеров. Эта микросхема используется в качестве одного 8-разрядного регистра. Две такие микросхемы могут работать параллельно, образуя 16-разрядный регистр. Для этого нужно связать соответствующие выводы 1 и 11.

3.3.3. Организация памяти

Хотя мы и совершили переход от простой памяти в 1 бит к 8-разрядной памяти (см. рис. 3.12*б*), чтобы построить память большого объема, требуется другой способ организации, при котором можно обращаться к отдельным словам.

Пример организации памяти, которая удовлетворяет этому критерию, показан на рисунке 3.13. Эта память содержит четыре 3-битных слова. Каждая операция считывает или записывает целое 3-битное слово. Хотя общий объем памяти (12 битов) ненамного больше, чем у нашего 8-разрядного триггера, такая память требует меньшего количества выводов, и, что особенно важно, подобная организация применима при построении памяти большого объема.

Структура памяти, изображенная на рисунке 3.13 содержит восемь входных линий (три входа для данных — L_1, L_2 и L_3; два входа для адресов — A_1 и A_0; три входа для управления — CS (*Chip Select* — выбор элемента памяти), RD (для различия между считыванием и записью) и OE (*Output Enable* — разрешение выдачи выходных сигналов)) и три выходные линии для данных — O_1, O_2 и O_3. Такую память в принципе можно поместить в корпус с 14 выводами (включая питание и «землю»), а 8-разрядный триггер требует наличия 20 выводов.

Чтобы выбрать микросхему памяти, внешняя логика должна установить CS на 1, а RD на 1 для чтения и на 0 для записи. Две адресные линии должны указывать, какое из четырех 3-битных слов нужно считывать или записывать. При операции считывания входные линии для данных не используются.

Выбирается слово и помещается на выходные линии для данных. При операции записи биты, находящиеся на входных линиях для данных, загружаются в выбранное слово памяти; выходные линии при этом не используются.

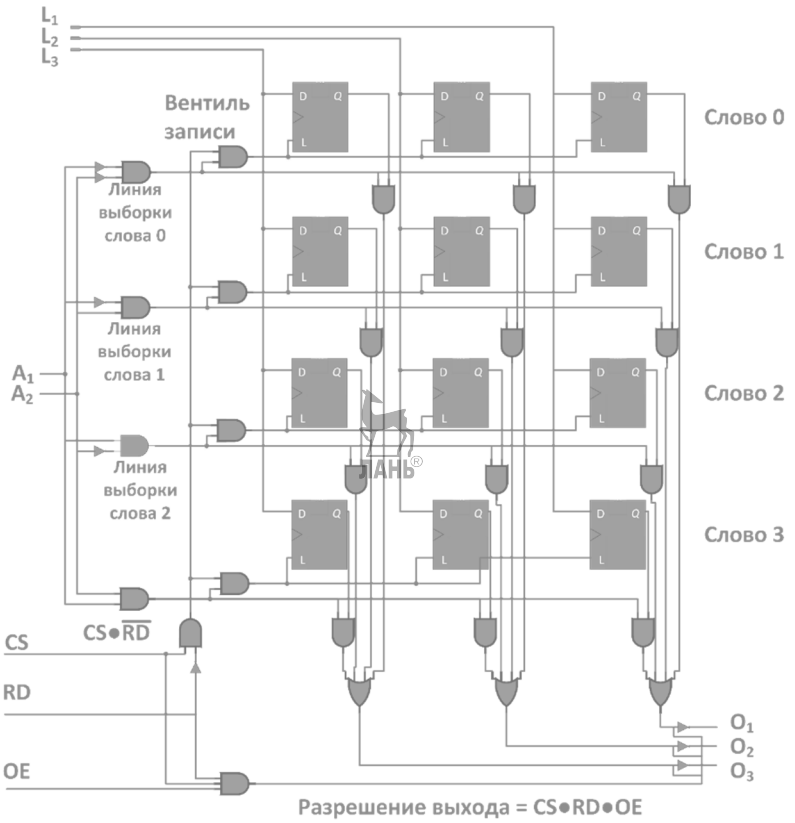


Рис. 3.13. Логическая блок-схема памяти

Четыре вентиля «И» для выбора слов в левой части схемы формируют декодер. Входные инверторы расположены так, что каждый вентиль запускается определенным адресом. Каждый вентиль приводит в действие линию выбора слов (для слов 0, 1, 2 и 3). Когда микросхема должна производить запись, вертикальная линия CS•RD получает значение 1, запуская один из четырех вентилей записи. Выбор вентиля зависит от того, какая именно линия выбора слов равна 1. Выходной сигнал вентиля записи приводит в действие все сигналы СК для выбранного слова, загружая входные данные в триггеры для этого слова. Запись производится только в том случае, если CS равно 1, а RD равно 0, при этом записывается только слово, выбранное адресами A_1 и A_0, а остальные слова не меняются.

Процесс считывания сходен с процессом записи. Декодирование адреса происходит точно так же, как и при записи. Но в данном случае линия CS • RD принимает значение 0, поэтому все вентили записи блокируются и ни один из триггеров не меняется. Вместо этого линия выбора слов запускает вентили «И», связанные с битами *Q* выбранного слова. Таким образом выбранное слово передает свои данные в 4-входовые вентили «ИЛИ», расположенные в нижней части схемы, а остальные три слова выдают 0. Следовательно, выход вентилей «ИЛИ» идентичен значению, сохраненному в данном слове. Остальные три слова никак не влияют на выходные данные.

Мы могли бы разработать схему, в которой три вентиля «ИЛИ» соединялись бы с тремя линиями вывода данных, но это вызвало бы некоторые проблемы. Мы рассматривали линии ввода данных и линии вывода данных как разные линии. На практике же используются одни и те же линии. Если бы мы связали вентили «ИЛИ» с линиями вывода данных, микросхема пыталась бы выводить данные (т. е. задавать каждой линии определенную величину) даже в процессе записи, мешая нормальному вводу данных. По этой причине желательно каким-то образом соединять вентили «ИЛИ» с линиями вывода данных при считывании и полностью разъединять их при записи. Все, что нам нужно, — электронный переключатель, который может устанавливать и разрушать связь за несколько наносекунд.

К счастью, такие переключатели существуют. На рисунке 3.14*а* показано символическое изображение так называемого буферного элемента без инверсии. Он содержит вход для данных, выход для данных и вход управления. Когда вход управления равен 1, буферный элемент работает как провод (рис. 3.14*б*). Когда вход управления равен 0, буферный элемент работает как разомкнутая цепь (рис. 3.14*б*), как будто кто-то отрезал выход для данных от остальной части схемы кусачками. Соединение может быть восстановлено за несколько наносекунд, если сделать сигнал управления равным 1.

На рисунке 3.14*г* показан буферный элемент с инверсией, который действует как обычный инвертор, когда сигнал управления равен 1, и отделяет выход от остальной части схемы, когда сигнал управления равен 0. Оба буферных элемента представляют собой устройства с тремя состояниями, поскольку они могут выдавать 0, 1 или вообще не выдавать сигнала (в случае с разомкнутой цепью). Буферные элементы, кроме того, усиливают сиг-

налы, поэтому они могут справляться с большим количеством сигналов одновременно. Иногда они используются в схемах именно по этой причине, даже если их свойства переключателя не нужны.

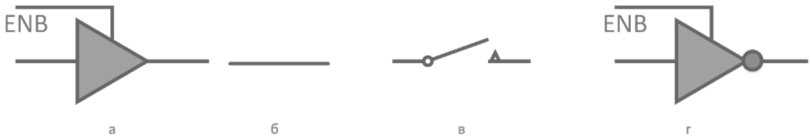


Рис. 3.14. Буферный элемент без инверсии (*а*); его действие, когда сигнал равен 1 (*б*) и 0 (*в*); буферный элемент с инверсией (*г*)

Когда CS, RD и OE все равны 1, то сигнал разрешения выдачи выходных данных также равен 1, в результате чего запускаются буферные элементы и слово помещается на выходные линии. Когда один из сигналов CS, RD и OE равен 0, выходы отсоединяются от остальной части схемы.

4. ЭЛЕКТРОНИКА ПРОЦЕССОРОВ И ШИН

Рассмотрим процессоры на цифровом логическом уровне, включая цоколевку (т. е. значение сигналов на различных выводах). Центральные процессоры тесно связаны с шинами, которые они используют.

4.1. Электронные микросхемы процессоров

Все современные процессоры помещаются на одной микросхеме. Это делает вполне определенным их взаимодействие с остальными частями системы. Каждая микросхема процессора содержит набор выводов, через которые происходит обмен информацией с внешним миром. Одни выводы передают сигналы от центрального процессора, другие принимают сигналы от других компонентов, третьи делают и то и другое. Изучив функции всех выводов, мы сможем узнать, как процессор взаимодействует с памятью и устройствами ввода-вывода на цифровом логическом уровне.

Выводы микросхемы центрального процессора можно разделить на три типа: адресные, информационные и управляющие. Эти выводы связаны с соответствующими выводами на микросхемах памяти и микросхемах устройств ввода-вывода через набор параллельных проводов (так называемую шину). Чтобы вызвать команду, центральный процессор сначала посылает в память адрес этой команды по адресным выводам. Затем он запускает одну или несколько линий управления, чтобы сообщить памяти, что ему нужно, например, прочитать слово. Память выдает ответ, помещая требуемое слово на информационные выводы процессора и посылая сигнал о том, что это сделано. Когда центральный процессор получает данный сигнал, он принимает слово и выполняет вызванную команду.

Команда может требовать чтения или записи слов, содержащих данные. В этом случае весь процесс повторяется для каждого дополнительного слова. Как происходит процесс чтения и записи, мы подробно рассмотрим далее. Важно понимать, что центральный процессор обменивается информацией с памятью и устройствами ввода-вывода, подавая сигналы на выводы и принимая сигналы на входы. Другого способа обмена информацией не существует.

Число адресных выводов и число информационных выводов — два ключевых параметра, которые определяют производительность процессора. Микросхема, содержащая m адресных выводов, может обращаться к $2m$

ячейкам памяти. Обычно *m* равно 16, 20, 32 или 64. Микросхема, содержащая *n* информационных выводов, может считывать или записывать *n*-битное слово за одну операцию. Обычно *n* равно 8, 16, 32, 36 или 64. Центральному процессору с восемью информационными выводами понадобится четыре операции, чтобы считать 32-битное слово, тогда как процессор, имеющий 32 информационных вывода, может сделать ту же работу в одну операцию. Следовательно, микросхема с 32 информационными выводами работает гораздо быстрее, но и стоит гораздо дороже.

Кроме адресных и информационных выводов каждый процессор содержит выводы управления. Выводы управления регулируют и синхронизируют поток данных к процессору и от него, а также выполняют другие разнообразные функции.

Все процессоры содержат выводы для питания (обычно +3,3 В или +5 В), «земли» и синхронизирующего сигнала (меандра). Остальные выводы разнятся от процессора к процессору. Тем не менее выводы управления можно разделить на несколько основных категорий:

1) управление шиной;

2) прерывание;

3) арбитраж шины;

4) состояние;

5) разное.

Схема типичного центрального процессора, в котором используются эти типы сигналов, изображена на рисунке 4.1.

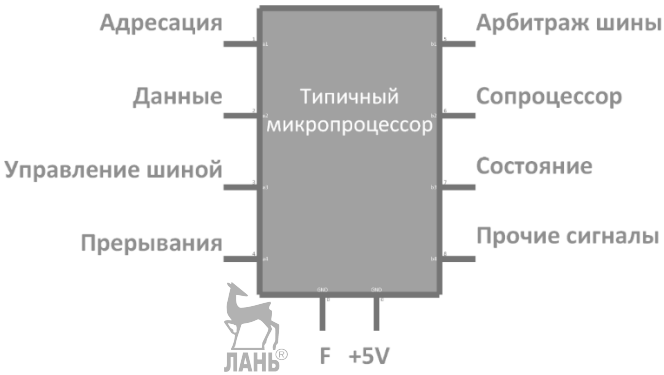


Рис. 4.1. Цоколевка типичного центрального процессора

Выводы управления шиной по большей части представляют собой выходы из центрального процессора в шину (и, следовательно, входы в микросхемы памяти и микросхемы устройств ввода-вывода). Они сообщают, что процессор хочет считать информацию из памяти или записать информацию в память, или сделать что-нибудь еще.

Выводы прерывания — это входы из устройств ввода-вывода в процессор. В большинстве систем процессор может дать сигнал устройству ввода-вывода начать операцию, а затем приступить к какому-нибудь другому действию, пока устройство ввода-вывода выполняет свою работу. Когда устройство ввода-вывода заканчивает свою работу, контроллер ввода-вывода посылает сигнал на один из выводов прерывания, чтобы прервать работу процессора и заставить его обслуживать устройство ввода-вывода (например, проверять ошибки ввода-вывода). Некоторые процессоры содержат выходной вывод, чтобы подтверждать получение сигнала прерывания.

Выводы разрешения конфликтов в шине нужны для того, чтобы регулировать поток информации в шине, т. е. не допускать таких ситуаций, когда два устройства пытаются воспользоваться шиной одновременно. В целях разрешения конфликтов центральный процессор считается устройством.

Некоторые центральные процессоры могут работать с различными сопроцессорами (например, с графическими процессорами, процессорами с плавающей точкой и т. п.). Чтобы обеспечить обмен информации между процессором и сопроцессором, нужны специальные выводы для передачи сигналов.

Кроме этих выводов у некоторых процессоров есть различные дополнительные выводы. Одни из них выдают или принимают информацию о состоянии, другие нужны для перезагрузки компьютера, а третьи — для обеспечения совместимости со старыми микросхемами устройств ввода-вывода.

4.2. Соединитель — шина

Шина — это группа проводников, соединяющих различные устройства. Шины можно разделить на группы в соответствии с выполняемыми функциями. Они могут быть внутренними по отношению к процессору и служить для передачи данных в АЛУ и из АЛУ, а могут быть внешними по отношению к процессору и связывать процессор с памятью или устрой-

ствами ввода-вывода. Каждый тип шины обладает определенными свойствами, и к каждому из них предъявляются определенные требования. В этом и следующих разделах мы сосредоточимся на шинах, которые связывают центральный процессор с памятью и устройствами ввода-вывода.

Первые персональные компьютеры имели одну внешнюю шину, которая называлась системной шиной. Она состояла из нескольких медных проводов (от 50 до 100), которые встраивались в материнскую плату. На материнской плате находились разъемы на одинаковых расстояниях друг от друга для микросхем памяти и устройств ввода-вывода. Современные персональные компьютеры обычно содержат специальную шину между центральным процессором и памятью и по крайней мере еще одну шину для устройств ввода-вывода. На рисунке 4.2 изображена система с одной шиной памяти и одной шиной ввода-вывода. В литературе шины обычно изображаются в виде жирных стрелок, как показано на этом рисунке. Разница между жирной и нежирной стрелками небольшая. Когда все биты одного типа, например адресные или информационные, рисуется обычная стрелка. Когда включаются адресные линии, линии данных и управления, используется жирная стрелка.

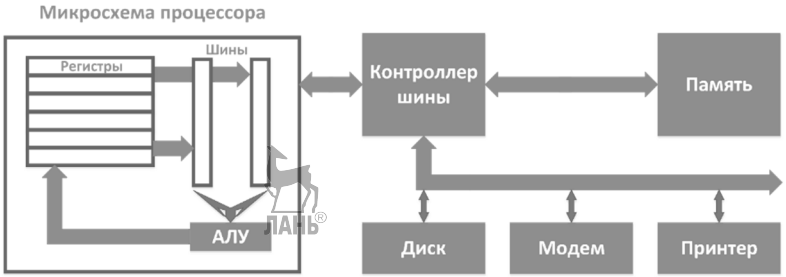


Рис. 4.2. Компьютерная система с несколькими шинами

Хотя разработчики процессоров могут использовать любой тип шины для микросхемы, должны быть введены четкие правила о том, как работает шина, и все устройства, связанные с шиной, должны подчиняться этим правилам, чтобы платы, которые выпускаются третьими лицами, подходили к системной шине. Эти правила называются протоколом шины. Кроме того, должны существовать определенные технические требования, чтобы платы от третьих производителей подходили к каркасу для печатных плат и имели разъемы, соответствующие материнской плате механически и с точки зрения мощностей, синхронизации и т. д.

Существует ряд широко используемых в компьютерном мире шин:

– Omnibus (PDP-8);

– Unibus (PDP-11);

– IBM PC (PC/XT);

– ISA (PC/AT);

– EISA (80386);

– MicroChannel (PC/2);

– PCI (различные персональные компьютеры);

–SCSI (различные персональные компьютеры и рабочие станции);

– Nubus (Macintosh);

– Universal Serial Bus (современные персональные компьютеры);

– FireWire (бытовая электроника);

– VME (оборудование в кабинетах физики);

– CAMAC (физика высоких энергий, измерительные устройства).

Стандартизация в этой области кажется маловероятной, и уже вложено слишком много средств во все эти несовместимые системы.

Некоторые устройства, связанные с шиной, являются активными и могут инициировать передачу информации по шине, тогда как другие являются пассивными и ждут запросов. Активное устройство называется задающим устройством, пассивное — подчиненным устройством.

Когда центральный процессор требует от контроллера диска считать или записать блок информации, центральный процессор действует как задающее устройство, а контроллер диска — как подчиненное устройство. Контроллер диска может действовать как задающее устройство, когда он командует памяти принять слова, которые считал с диска (табл. 4.1). Память ни при каких обстоятельствах не может быть задающим устройством.

Двоичные сигналы, которые выдают устройства компьютера, часто недостаточно интенсивны, чтобы активизировать шину, особенно если она достаточно длинная и если к ней подсоединено много устройств. По этой причине большинство задающих устройств шины обычно связаны с ней через микросхему, которая называется драйвером шины и является по существу двоичным усилителем.

Сходным образом большинство подчиненных устройств связаны с шиной приемником шины. Для устройств, которые могут быть и задающим, и подчиненным устройством, используется приемопередатчик шины. Эти микросхемы взаимодействия с шиной часто являются устройствами с тремя

состояниями, что дает им возможность отсоединяться, когда они не нужны. Иногда они подключаются через открытый коллектор, что дает сходный эффект. Когда одно или несколько устройств на открытом коллекторе требуют доступа к шине в одно и то же время, результатом является булева операция «ИЛИ» над всеми этими сигналами. Такое соглашение называется монтажным «ИЛИ». В большинстве шин одни линии являются устройствами с тремя состояниями, а другие, которым требуется свойство монтажного «ИЛИ», — открытым коллектором.

Таблица 4.1. Примеры задающих и подчиненных устройств

Задающее устройство	Подчиненное устройство	Пример
Центральный процессор	Память	Вызов команд и данных
Центральный процессор	Устройство ввода-вывода	Инициализация передачи данных
Центральный процессор	Сопроцессор	Передача команды от процессора
Устройство ввода-вывода	Память	Прямой доступ к памяти
Сопроцессор	Центральный процессор	Вызов сопроцессором операндов из центрального процессора

Как и процессор, шина имеет адресные линии, информационные линии и линии управления. Тем не менее между выводами процессора и сигналами шины может и не быть взаимно однозначного соответствия. Например, некоторые процессоры содержат три вывода, которые выдают сигнал чтения из памяти или записи в память, или чтения устройства ввода-вывода, или записи на устройство ввода-вывода, или какой-либо другой операции. Обычная шина может содержать одну линию для чтения из памяти, вторую линию для записи в память, третью — для чтения устройства ввода-вывода, четвертую — для записи на устройство ввода-вывода и т. д. Микросхема-декодер должна тогда связывать данный процессор с такой шиной, чтобы преобразовывать 3-битный кодированный сигнал в отдельные сигналы, которые могут управлять линиями шины.

Разработка шин и принципы действия шин — это достаточно сложные вопросы. Принципиальными вопросами в разработке являются ширина шины, синхронизация шины, арбитраж шины и функционирование шины.

Все эти параметры существенно влияют на скорость и пропускную способность шины.

4.2.1. Ширина шины

Ширина шины — самый очевидный параметр при разработке. Чем больше адресных линий содержит шина, тем к большему объему памяти может обращаться процессор. Если шина содержит *n* адресных линий, процессор может использовать ее для обращения к 2^n различным ячейкам памяти. Для памяти большой емкости необходимо много адресных линий. Это звучит достаточно просто.

Проблема заключается в том, что для широких шин требуется больше проводов, чем для узких. Они занимают больше физического пространства (например, на материнской плате), и для них нужны разъемы большего размера. Все эти факторы делают шину дорогостоящей. Следовательно, необходим компромисс между максимальным размером памяти и стоимостью системы. Система с шиной, содержащей 64 адресные линии, и памятью в 232 байт будет стоить дороже, чем система с шиной, содержащей 32 адресные линии, и такой же памятью в 232 байт. Дальнейшее расширение не бесплатное.

Многие разработчики систем недальновидны, что приводит к неприятным последствиям. Первая модель IBM PC содержала процессор 8088 и 20-битную адресную шину (рис. 4.3*а*). Шина позволяла обращаться к 1 Мбайт памяти.

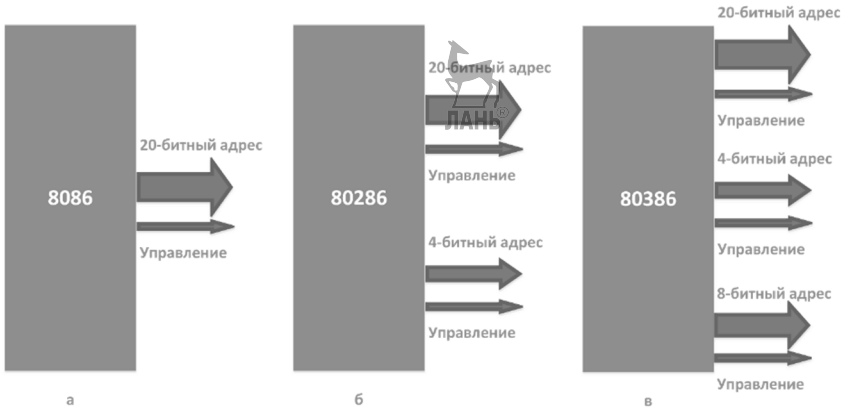


Рис. 4.3. Расширение адресной шины с течением времени; микросхемы Intel типа 8086 (*а*); 80286 (*б*); 80386 (*в*)

4.2.2. Арбитраж шины

До этого момента неявно предполагалось, что существует только одно задающее устройство шины — центральный процессор. В действительности, микросхемы ввода-вывода могут становиться задающим устройством при считывании информации из памяти и записи информации в память. Кроме того, они могут вызывать прерывания. Сопроцессоры также могут становиться задающим устройством шины.

Чтобы предотвратить хаос, который может возникнуть, когда задающим устройством шины могут стать два или несколько устройств одновременно, существует специальный механизм — арбитраж шины.

Механизмы арбитража могут быть централизованными или децентрализованными. Рассмотрим централизованный арбитраж. Простой пример централизованного арбитража показан на рисунке 4.4*а*. В данном примере один арбитр шины определяет, чья очередь следующая. Часто бывает, что арбитр встроен в микросхему процессора, но иногда требуется отдельная микросхема. Шина содержит одну линию запроса (монтажное «ИЛИ»), которая может запускаться одним или несколькими устройствами в любое время. Арбитр не может определить, сколько устройств запрашивают шину. Он может определять только наличие или отсутствие запросов.

Когда арбитр видит запрос шины, он запускает линию предоставления шины. Эта линия последовательно связывает все устройства ввода-вывода (как в елочной гирлянде). Когда физически ближайшее к арбитру устройство воспринимает сигнал предоставления шины, оно проверяет, нет ли запроса шины. Если запрос есть, устройство пользуется шиной, но не распространяет сигнал предоставления дальше по линии. Если запроса нет, устройство передает сигнал предоставления шины следующему устройству. Это устройство тоже проверяет, есть ли запрос, и действует соответствующим образом в зависимости от наличия или отсутствия запроса. Передача сигнала предоставления шины продолжается до тех пор, пока какое-нибудь устройство не воспользуется предоставленной шиной. Такая система называется системой последовательного опроса. При этом приоритеты устройств зависят от того, насколько близко они находятся к арбитру. Ближайшее к арбитру устройство обладает главным приоритетом.

Чтобы обойти такую систему, в которой приоритеты зависят от расстояния от арбитра, в некоторых шинах устраивается несколько уровней приоритета. На каждом уровне приоритета есть линия запроса шины и линия предоставления шины.

На рисунке 4.4*б* изображены два уровня (хотя в действительности шины обычно содержат четыре, восемь или 16 уровней). Каждое устройство связано с одним из уровней запроса шины, причем чем выше уровень приоритета, тем больше устройств привязано к этому уровню. На рисунке 4.4*б* можно видеть, что устройства 1, 2 и 4 используют приоритет 1, а устройства 3 и 5 — приоритет 2.

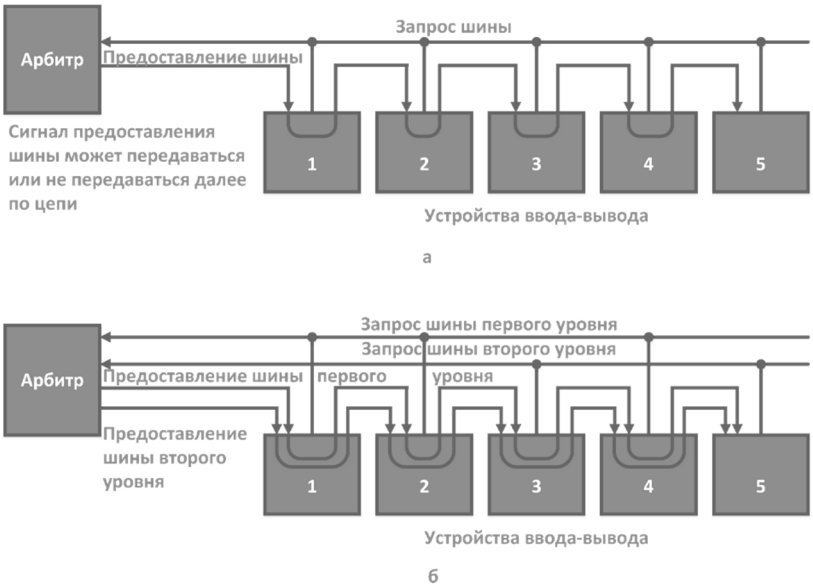


Рис. 4.4. Одноуровневый централизованный арбитраж шины с использованием системы последовательного опроса (*а*); двухуровневый центральный арбитраж (*б*)

Если одновременно запрашивается несколько уровней приоритета, арбитр предоставляет шину самому высокому уровню. Среди устройств одинакового приоритета используется система последовательного опроса. На рисунке 4.4 в случае конфликта устройство 2 «побеждает» устройство 4, а устройство 4 «побеждает» устройство 3. Устройство 5 имеет низший приоритет, поскольку оно находится в самом конце самого нижнего уровня.

Линия предоставления шины второго уровня необязательно должна последовательно связывать устройства 1 и 2, поскольку они не могут посылать на нее запросы. Однако гораздо проще провести все линии предоставления шины через все устройства, чем соединять устройства особым образом в зависимости от их приоритетов.

Некоторые арбитры содержат третью линию, которая запускается, как только устройство принимает сигнал предоставления шины, и берет шину в свое распоряжение. Как только запускается эта линия подтверждения приема, линии запроса и предоставления шины могут быть отключены. В результате другие устройства могут запрашивать шину, пока первое устройство использует ее. К тому моменту, когда закончится текущая передача, следующее задающее устройство уже будет выбрано. Это устройство может начать работу, как только отключается линия подтверждения приема. С этого момента начинается следующий арбитраж. Такая структура требует наличия дополнительной линии и большего количества логических схем в каждом устройстве, но зато при этом циклы шины используются рациональнее.

В системах, где память связана с главной шиной, центральный процессор должен завершать работу со всеми устройствами ввода-вывода практически на каждом цикле шины. Чтобы решить эту проблему, можно предоставить центральному процессору самый низкий приоритет. При этом шина будет предоставляться процессору только в том случае, если она не нужна ни одному другому устройству. Центральный процессор всегда может подождать, а устройства ввода-вывода должны получить доступ к шине как можно быстрее, чтобы не потерять данные. Диски, вращающиеся с высокой скоростью, тоже не могут ждать. Во многих современных компьютерах память помещается на одну шину, а устройства ввода-вывода — на другую, поэтому им не приходится завершать работу, чтобы предоставить доступ к шине.

Возможен также децентрализованный арбитраж шины. Например, компьютер может содержать 16 приоритетных линий запроса шины. Когда устройству нужна шина, оно запускает свою линию запроса. Все устройства контролируют все линии запроса, поэтому в конце каждого цикла шины каждое устройство может определить, обладает ли оно в данный момент высшим приоритетом и, следовательно, разрешено ли линии пользоваться

шиной в следующем цикле. Такой метод требует наличия большего количества линий, но зато не требует затрат на арбитра. Он также ограничивает число устройств числом линий запроса.

При другом типе децентрализованного арбитража используется только три линии независимо от того, сколько устройств имеется в наличии (рис. 4.5). Первая линия — монтажное «ИЛИ». Она используется для запроса шины. Вторая линия называется BUSY. Она запускается текущим задающим устройством шины. Третья линия используется для арбитража шины. Она последовательно соединяет все устройства. Начало цепи связано с источником питания с напряжением 5 В.

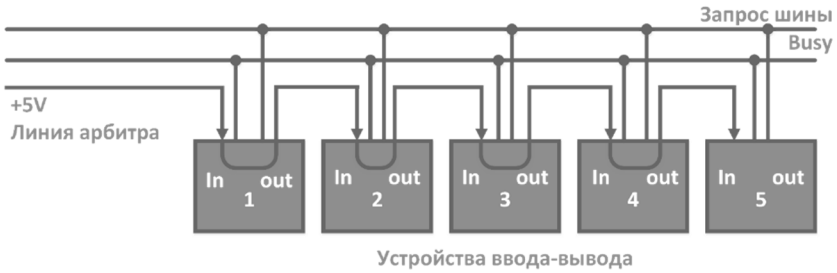


Рис. 4.5. Децентрализованный арбитраж шины

Когда шина не требуется ни одному из устройств, линия арбитра передает сигнал всем устройствам. Чтобы получить доступ к шине, устройство сначала проверяет, свободна ли шина и установлен ли сигнал арбитра IN. Если сигнал IN не установлен, устройство не может стать задающим устройством шины. В этом случае оно сбрасывает сигнал OUT. Если сигнал IN установлен, устройство также сбрасывает сигнал OUT, в результате чего следующее устройство не получает сигнал IN и, в свою очередь, сбрасывает сигнал OUT. Следовательно, все следующие по цепи устройства не получают сигнал IN и сбрасывают сигнал OUT. В результате остается только одно устройство, у которого сигнал IN установлен, а сигнал OUT сброшен. Оно становится задающим устройством шины, запускает линию BUSY и сигнал OUT и начинает передачу данных.

Таким образом, из всех устройств, которым нужна шина, доступ к шине получает самое левое. Такая система сходна с системой последовательного опроса, только в данном случае нет арбитра, поэтому она стоит дешевле и работает быстрее. К тому же не возникает проблем со сбоями арбитра.

4.2.3. Примеры шин

Шины соединяют компьютерную систему в одно целое. Рассмотрим несколько примеров шин: шину ISA, шину PCI и Universal Serial Bus (универсальную последовательную шину). Шина ISA представляет собой небольшое расширение первоначальной шины IBM PC. По соображениям совместимости она все еще используется во всех персональных компьютерах Intel. Однако такие компьютеры всегда содержат еще одну шину, которая работает быстрее, чем шина ISA. Шина ISA не используется в современных компьютерах. Шина PCI шире, чем ISA, и функционирует с более высокой тактовой частотой. Шина USB обычно используется в качестве шины ввода-вывода для периферийных устройств малого быстродействия (например, мыши, клавиатуры, запоминающих устройств и т. п.).

4.2.4. Шина ISA

Шина IBM PC была неофициальным стандартом систем с процессором 8088, поскольку практически все производители клонов скопировали ее, чтобы иметь возможность использовать в своих системах платы ввода-вывода от различных поставщиков. Шина содержала 62 сигнальные линии, из них 20 для адреса ячейки памяти, восемь для данных и по одной для сигналов считывания информации из памяти, записи информации в память, считывания с устройства ввода-вывода и записи на устройство ввода-вывода. Имелись и сигналы для запроса прерываний и их разрешения, а также для прямого доступа к памяти. Шина была очень примитивной.

Шина IBM PC встраивалась в материнскую плату персонального компьютера. На плате было несколько разъемов, расположенных на расстоянии 2 см друг от друга. В разъемы вставлялись различные платы. На платах имелись позолоченные выводы (по 31 с каждой стороны), которые физически подходили под разъемы. Через них осуществлялся электрический контакт с разъемом.

Когда компания *IBM* разрабатывала компьютер PC/AT с процессором 80286, она столкнулась с некоторыми трудностями. Если бы компания разработала совершенно новую 16-битную шину, многие потенциальные покупатели не стали бы приобретать этот компьютер, поскольку ни одна из сменных плат, выпускаемых другими компаниями, не подошла бы к новой

машине. С другой стороны, если оставить старую шину, то новый процессор не сможет реализовать все свои возможности (например, возможность обращаться к 16 Мбайт памяти и передавать 16-битные слова).

В результате было принято решение расширить старую шину. Сменные платы персональных компьютеров содержали краевой разъем (62 контакта), но этот краевой разъем проходил не по всей длине платы. Поэтому на плате поместили еще один краевой разъем, смежный с главным. Кроме того, схемы PC/AT были разработаны таким образом, чтобы можно было подсоединять платы обоих типов.

Второй краевой разъем шины PC/AT содержит 36 линий. Из них 31 предназначена для дополнительных адресных линий, информационных линий, линий прерывания, каналов ПДП (прямого доступа к памяти), а также для питания и «земли». Остальные связаны с различиями между 8-битными и 16-битными передачами. Когда компания *IBM* выпустила серию компьютеров PS/2, пришло время начать разработку шины заново. С одной стороны, это решение было обусловлено чисто техническими причинами (шина PC к тому времени уже устарела). Но, с другой стороны, оно было вызвано желанием воспрепятствовать компаниям, выпускавшим клоны, которые в то время заполонили компьютерный рынок. Поэтому компьютеры PS/2 с высокой и средней производительностью были оснащены абсолютно новой шиной MCA (*MicroChannel Architecture*), которая была защищена патентами.

Компьютерная промышленность отреагировала на такой шаг введением своего собственного стандарта, шины ISA (*Industry Standard Architecture* — стандартная промышленная архитектура), которая, по существу, представляет собой шину PC/AT, работающую при частоте 8,33 МГц. Преимущество такого подхода состоит в том, что при этом сохраняется совместимость с существующими машинами и платами. Отметим, что в основе этого стандарта лежит шина, разработанная компанией *IBM*. *IBM* когда-то необдуманно предоставила права на производство этой шины многим компаниям, чтобы как можно больше производителей имели возможность выпускать платы для компьютеров IBM. Однако впоследствии компании *IBM* пришлось об этом сильно пожалеть. Эта шина до сих пор используется во всех персональных компьютерах с процессором Intel, хотя обычно кроме нее там есть еще одна или несколько других шин.

Позднее шина ISA была расширена до 32 разрядов. У нее появились некоторые новые особенности (например, возможность параллельной обработки). Такая шина называлась EISA (*Extended Industry Standard Architecture* — расширенная архитектура промышленного стандарта). Для нее было разработано несколько плат.

4.2.5. Шина PCI

В первых компьютерах IBM PC большинство приложений имели дело с текстами. Постепенно с появлением Windows вошли в употребление графические интерфейсы пользователя. Ни одно из этих приложений не давало большой нагрузки на шину ISA. Однако с течением времени появилось множество различных приложений, в том числе игр, для которых потребовалось полноэкранное видеоизображение, и ситуация коренным образом изменилась.

Давайте произведем небольшое вычисление. Рассмотрим монитор 1024×768 для цветного движущегося изображения (3 байта/пиксель). Одно экранное изображение содержит 2,25 Мбайт данных. Для показа плавных движений требуется 30 кадров в секунду, и, следовательно, скорость передачи данных должна быть 67,5 Мбайт/с.

В действительности дело обстоит гораздо хуже, поскольку, чтобы передать изображение, данные должны перейти с жесткого диска, компакт-диска или DVD-диска через шину в память. Затем данные должны поступить в графический адаптер (тоже через шину). Таким образом, пропускная способность шины должна быть 135 Мбайт/с, и это только для передачи видеоизображения. Но в компьютере есть еще центральный процессор и другие устройства, которые тоже должны пользоваться шиной, поэтому пропускная способность должна быть еще выше.

Максимальная частота передачи данных шины ISA — 8,33 МГц. Она способна передавать 2 байта за цикл, поэтому ее максимальная пропускная способность составляет 16,7 Мбайт/с. Шина EISA может передавать 4 байта за цикл. Ее пропускная способность достигает 33,3 Мбайт/с. Ясно, что ни одна из них совершенно не соответствует тому, что требуется для полноэкранного видео.

В 1990 г. компания *Intel* разработала новую шину с гораздо более высокой пропускной способностью, чем у шины EISA. Эту шину назвали PCI (*Peripheral Component Interconnect* — взаимодействие периферийных компонентов). Компания *Intel* запатентовала шину PCI и сделала все патенты

всеобщим достоянием, так что любая компания могла производить периферические устройства для этой шины без каких-либо выплат за право пользования патентом. Компания *Intel* также сформировала промышленный консорциум *Special Interest Group*, который должен был заниматься дальнейшим усовершенствованием шины PCI. Все эти действия привели к тому, что шина PCI стала чрезвычайно популярной. Фактически в каждом компьютере Intel (начиная с Pentium), а также во многих других компьютерах содержится шина PCI. Даже компания *Sun* выпустила версию UltraSPARC, в которой используется шина PCI (это компьютер UltraSPARC III).

Первая шина PCI передавала 32 бита за цикл и работала с частотой 33 МГц (время цикла 30 нс), общая пропускная способность составляла 133 Мбайт/с. В 1993 г. появилась шина PCI 2.0, а в 1995 г. — PCI 2.1. Шина PCI 2.2 подходит и для портативных компьютеров (где требуется экономия заряда батареи). Шина PCI работает с частотой 66 МГц, способна передавать 64 бита за цикл, а ее общая пропускная способность составляет 528 Мбайт/с. При такой производительности полноэкранное видеоизображение вполне достижимо (предполагается, что диск и другие устройства системы справляются со своей работой). Во всяком случае, шина PCI не будет ограничивать производительность системы.

Хотя 528 Мбайт/с — достаточно высокая скорость передачи данных, все же здесь есть некоторые проблемы. Во-первых, этого недостаточно для шины памяти. Во-вторых, эта шина несовместима со всеми старыми картами ISA. По этой причине компания *Intel* решила разрабатывать компьютеры с тремя и более шинами, как показано на рисунке 4.6. Здесь мы видим, что центральный процессор может обмениваться информацией с основной памятью через специальную шину памяти и что шину ISA можно связать с шиной PCI. Такая архитектура используется фактически во всех компьютерах Pentium II, поскольку она удовлетворяет всем требованиям.

Ключевыми компонентами данной архитектуры являются мосты между шинами (эти микросхемы выпускает компания *Intel* — отсюда такой интерес к проекту). Мост PCI связывает центральный процессор, память и шину PCI. Мост ISA связывает шину PCI с шиной ISA, а также поддерживает один или два диска IDE.

Практически все системы Pentium II выпускаются с одним или несколькими свободными слотами PCI для подключения дополнительных высокоскоростных периферийных устройств и с одним или несколькими слотами ISA для подключения низкоскоростных периферийных устройств.

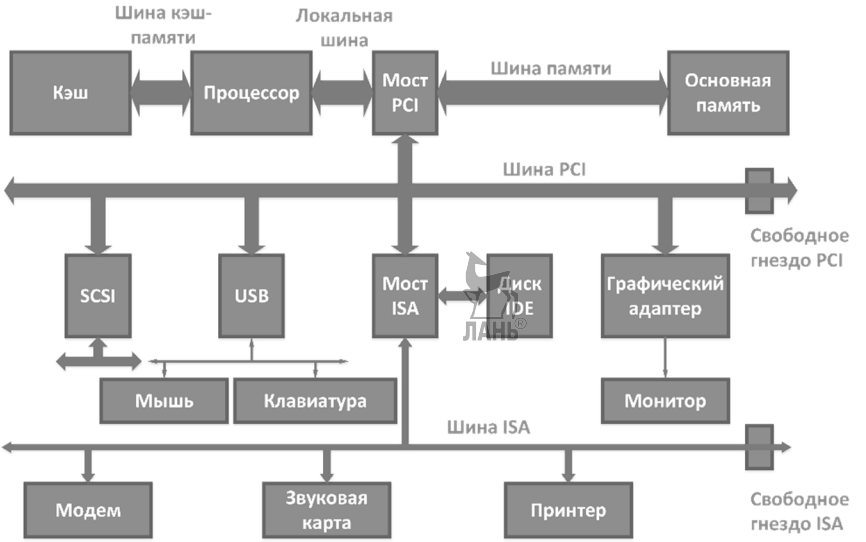


Рис. 4.6. Архитектура типичной системы Pentium

Преимущество системы, изображенной на рисунке 4.6, состоит в том, что шина между центральным процессором и памятью имеет высокую пропускную способность, шина PCI также обладает высокой пропускной способностью и хорошо подходит для связи с быстрыми периферийными устройствами (SCSI-дисками, графическими адаптерами и т. п.), и при этом еще могут использоваться старые платы ISA. На рисунке также изображена шина USB.

Мы проиллюстрировали систему с одной шиной PCI и одной шиной ISA. На практике может использоваться и по несколько шин каждого типа. Существуют специальные мосты, которые связывают две шины PCI, поэтому в больших системах может содержаться несколько отдельных шин PCI (две и более). В системе также может быть несколько мостов (два и более), которые связывают шину PCI и шину ISA, что дает возможность использовать несколько шин ISA.

4.2.6. Шина USB

Шина PCI очень хорошо подходит для подсоединения высокоскоростных периферических устройств, но использовать интерфейс PCI для низкоскоростных устройств ввода-вывода (например, мыши и клавиатуры) было бы слишком дорого.

Изначально каждое стандартное устройство ввода-вывода подсоединялось к компьютеру особым образом, при этом для добавления новых устройств использовались свободные слоты ISA и PCI. К сожалению, такая схема имеет некоторые недостатки. Например, каждое новое устройство ввода-вывода часто снабжено собственной платой ISA или PCI. Пользователь при этом должен сам установить переключатели и перемычки на плате и удостовериться, что установленные параметры не конфликтуют с другими платами. Затем пользователь должен открыть системный блок, аккуратно вставить плату, закрыть системный блок, а затем включить компьютер. Для многих этот процесс очень сложен и часто приводит к ошибкам.

Кроме того, число слотов ISA и PCI очень мало (обычно их два или три). Платы Plug and Play исключают установку переключателей, но пользователь все равно должен открывать компьютер и вставлять туда плату. К тому же количество слотов шины ограничено.

В середине 1990-х гг. представители семи компаний (*Compaq*, *DEC*, *IBM*, *Intel*, *Microsoft*, *NEC* и *Nothern Telecom*) собрались вместе, чтобы разработать шину, оптимально подходящую для подсоединения низкоскоростных устройств. Потом к ним примкнули сотни других компаний. Результатом их работы стала шина USB (*Universal Serial Bus* — универсальная последовательная шина), которая сейчас широко используется в персональных компьютерах.

Некоторые требования, изначально составляющие основу проекта:

1) пользователи не должны устанавливать переключатели и перемычки на платах и устройствах;

2) пользователи не должны открывать компьютер, чтобы установить новые устройства ввода-вывода;

3) должен существовать только один тип кабеля, подходящий для подсоединения всех устройств;

4) устройства ввода-вывода должны получать питание через кабель;

5) необходима возможность подсоединения к одному компьютеру до 127 устройств;

6) система должна поддерживать устройства реального времени (например, звук, телефон);

7) должна быть возможность устанавливать устройства во время работы компьютера;

8) должна отсутствовать необходимость перезагружать компьютер после установки нового устройства;

9) производство новой шины и устройств ввода-вывода для нее не должно требовать больших затрат.

Шина USB удовлетворяет всем этим условиям. Она разработана для низкоскоростных устройств (клавиатур, мышей, фотоаппаратов, сканеров, цифровых телефонов и т. д.). Общая пропускная способность шины составляет 1,5 Мбайт/с. Этого достаточно для большинства таких устройств. Предел был выбран для того, чтобы снизить стоимость шины.

Шина USB состоит из центрального хаба 1, который вставляется в разъем главной шины (рис. 4.7). Этот центральный хаб (часто называемый корневым концентратором) содержит разъемы для кабелей, которые могут подсоединяться к устройствам ввода-вывода или к дополнительным хабам, чтобы обеспечить большее количество разъемов. Таким образом, топология шины USB представляет собой дерево с корнем в центральном хабе, который находится внутри компьютера.

Коннекторы кабеля со стороны устройства отличаются от коннекторов со стороны хаба, чтобы пользователь случайно не подсоединил кабель другой стороной. Когда подсоединяется новое устройство ввода-вывода, центральный хаб (концентратор) распознает это и прерывает работу операционной системы. Затем операционная система запрашивает новое устройство, что оно собой представляет и какая пропускная способность шины для него требуется. Если операционная система решает, что для этого устройства пропускной способности достаточно, она приписывает ему уникальный адрес (1–127) и загружает этот адрес и другую информацию в регистры конфигурации внутри устройства. Таким образом, новые устройства могут подсоединяться «на лету», при этом пользователю не нужно устанавливать новые платы ISA или PCI. Неинициализированные платы начинаются с адреса 0, поэтому к ним можно обращаться. Многие устройства снабжены встроенными сетевыми концентраторами для дополнительных устройств. Например, монитор может содержать два хаба для правой и левой колонок.

Шина USB представляет собой ряд каналов от центрального хаба к устройствам ввода-вывода. Каждое устройство может разбить свой канал максимум на 16 подканалов для различных типов данных (например, аудио и видео). В каждом канале или подканале данные перемещаются от центрального концентратора к устройству или обратно. Между двумя устройствами ввода-вывода обмена информацией не происходит.

Ровно через каждую миллисекунду (±0,05 мс) центральный концентратор передает новый кадр, чтобы синхронизировать все устройства во времени. Кадр состоит из пакетов, первый из которых передается от концентратора к устройству. Следующие пакеты кадра могут передаваться в том же направлении, а могут и в противоположном (от устройства к хабу). На рисунке 4.7 показаны четыре последовательных кадра.

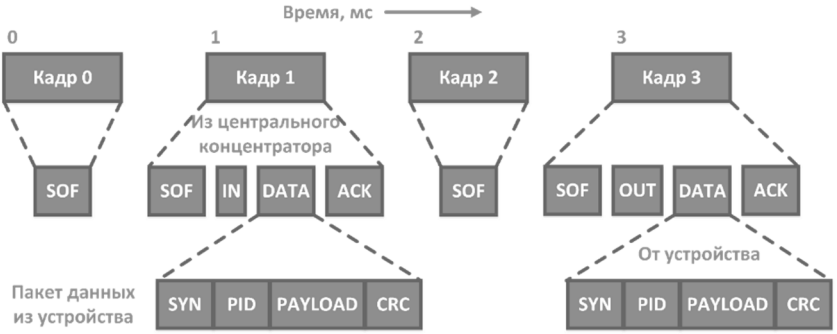


Рис. 4.7. Передача кадров цетральным концентратором шины USB

Рассмотрим рисунок 4.7. В кадрах 0 и 2 не происходит никаких действий, поэтому в них содержится только пакет SOF (*Start of Frame* — начало кадра). Этот пакет всегда посылается всем устройствам. Кадр 1 — упорядоченный опрос (например, сканеру посылается запрос на передачу битов сканированного им изображения).

Кадр 3 состоит из отсылки данных какому-нибудь устройству (например, принтеру). Шина USB поддерживает четыре типа кадров: кадры управления, изохронные кадры, кадры передачи больших массивов данных и кадры прерывания. Кадры управления используются для конфигурации устройств, передачи команд устройствам и запросов об их состоянии. Изохронные кадры предназначены для устройств реального времени (микрофонов, акустических систем и телефонов), которые должны принимать и посылать данные через

равные временные интервалы. Задержки хорошо прогнозируются, но в случае ошибки такие устройства не производят повторной передачи. Кадры следующего типа используются для передач большого объема данных от устройств и к устройствам без требований реального времени (например, принтеров). Наконец, кадры последнего типа нужны для того, чтобы осуществлять прерывания, поскольку шина USB не поддерживает прерывания. Например, вместо того, чтобы вызывать прерывание всякий раз, когда происходит нажатие клавиши, операционная система может вызывать прерывания каждые 50 млс и «собирать» все задержанные нажатия клавиш.

Кадр состоит из одного или нескольких пакетов. Пакеты могут посылаться в обоих направлениях. Существует четыре типа пакетов: маркеры, пакеты данных, пакеты квитирования и специальные пакеты. Маркеры передаются от концентратора к устройству и предназначены для управления системой. Пакеты SOF, IN и OUT на рисунке 4.7 — маркеры. Пакет SOF (*Start of Frame* — начало кадра) является первым в любом кадре и показывает начало кадра. Если никаких действий выполнять не нужно, пакет SOF — единственный в кадре. Пакет IN — это запрос. Этот пакет требует, чтобы устройство выдало определенные данные. Поля в пакете IN содержат информацию, какой именно канал запрашивается, и таким образом устройство определяет, какие именно данные выдавать (если оно обращается с несколькими потоками данных). Пакет OUT объявляет, что далее последует передача данных для устройства. Последний тип маркера, SETUP (он не показан на рисунке), используется для конфигурации.

Кроме маркеров существует еще три типа пакетов. Это пакеты DATA (используются для передачи 64 байтов информации в обоих направлениях), пакеты квитирования и специальные пакеты. Формат пакета данных показан на рисунке 4.7. Он состоит из 8-битного поля синхронизации, 8-битного указателя типа пакета (PID), полезной нагрузки и 16-битного CRC (*Cyclic Redunduncy Code* — циклический избыточный код) для обнаружения ошибок. Есть три типа пакетов квитирования: ACK (предыдущий пакет данных был принят правильно), NAC (найдена ошибка CRC) и STALL (подождите, пожалуйста, я сейчас занят).

Центральный концентратор должен отсылать новый кадр каждую миллисекунду, даже если не происходит никаких действий. Кадры 0 и 2 содержат только один пакет SOF, что говорит о том, что ничего не происходит. Кадр 1 представляет собой опрос, поэтому он начинается с пакетов SOF и

IN, которые передаются от компьютера к устройству ввода-вывода, а затем следует пакет DATA от устройства к компьютеру. Пакет ACK сообщает устройству, что данные были получены без ошибок. В случае ошибки устройство получает пакет NACK, после чего данные передаются заново (отметим, что изохронные данные повторно не передаются). Кадр 3 похож по структуре на кадр 1, но в нем поток данных направлен от компьютера к устройству.

4.3. Средства сопряжения схем

Обычная компьютерная система малого или среднего размера состоит из микросхемы процессора, микросхем памяти и нескольких контроллеров ввода-вывода. Все эти микросхемы соединены шиной. Мы уже рассмотрели память, центральные процессоры и шины. Теперь настало время изучить микросхемы ввода-вывода. Именно через эти микросхемы компьютер обменивается информацией с внешними устройствами.

4.3.1. Микросхемы ввода-вывода

В настоящее время существует множество различных микросхем ввода-вывода. Новые микросхемы появляются постоянно. Из наиболее распространенных можно назвать UART, USART, контроллеры CRT (CRT — электронно-лучевая трубка), дисковые контроллеры и PIO. UART (*Universal Asynchronous Receiver Transmitter* — универсальный асинхронный приемопередатчик) — это микросхема, которая может считывать байт из шины данных и передавать этот байт по битам на линию последовательной передачи к терминалу или получать данные от терминала. Скорость работы микросхем UART различна — от 50 до 19 200 бит/с; ширина знака — от 5 до 8 битов; 1 или 2 стоповых бита. Микросхема может обеспечивать проверку на четность или на нечетность, контроль по четности может также отсутствовать, все это находится под управлением программ. Микросхема USART (*Universal Synchronous Asynchronous Receiver Transmitter* — универсальный синхронно-асинхронный приемопередатчик) может осуществлять синхронную передачу, используя ряд протоколов. Она также выполняет все функции UART.

4.3.2. Микросхемы PIO

Типичным примером микросхемы PIO (*Parallel Input/Output* — параллельный ввод-вывод) является Intel 8255A (рис. 4.8). Она содержит 24 линии ввода-вывода и может сопрягаться с любыми устройствами, совместимыми с TTL-схемами (например, клавиатурами, переключателями, индикаторами, принтерами). Программа центрального процессора может записать 0 или 1 на любую линию или считать входное состояние любой линии, обеспечивая высокую гибкость. Микросхема PIO часто заменяет целую плату с микросхемами МИС и СИС (особенно во встроенных системах).

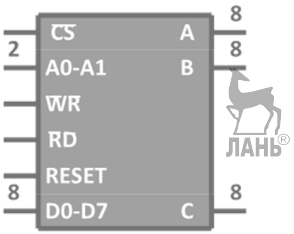


Рис. 4.8. Микроскхема 8255A

Центральный процессор может конфигурировать микросхему 8255A различными способами, загружая регистры состояния микросхемы, мы остановимся на некоторых наиболее простых режимах работы. Можно представить данную микросхему в виде трех 8-битных портов *A*, *B* и *C*. С каждым портом связан 8-битный регистр. Чтобы установить линии на порт, центральный процессор записывает 8-битное число в соответствующий регистр, и это 8-битное число появляется на выходных линиях и остается там до тех пор, пока регистр не будет перезаписан.

Чтобы использовать порт для входа, центральный процессор просто считывает соответствующий регистр.

Другие режимы работы предусматривают квитирование связи с внешними устройствами. Например, чтобы передать данные устройству, микросхема 8255A может представить данные на порт вывода и подождать, пока устройство не выдаст сигнал о том, что данные получены и можно посылать еще. В данную микросхему включены необходимые логические схемы для фиксирования таких импульсов и передачи их центральному процессору.

Из рисунка 4.8 видно, что, помимо 24 выводов для трех портов, микросхема 8255A содержит восемь линий, непосредственно связанных с шиной

данных, линию выбора элемента памяти, линии чтения и записи, две адресные линии и линию для переустановки микросхемы. Две адресные линии выбирают один из четырех внутренних регистров, три из которых соответствуют портам *A*, *B* и *C*.

Четвертый регистр — регистр состояния. Он определяет, какие порты используются для входа, а какие — для выхода, а также выполняет некоторые другие функции. Обычно две адресные линии соединяются с двумя младшими битами адресной шины.

4.3.3. Декодирование адреса

Рассмотрим простой 16-битный встроенный компьютер, состоящий из центрального процессора, стираемого программируемого ПЗУ объемом 2К × 8 байт для хранения программы, ОЗУ объемом 2К × 8 байт для хранения данных и микросхемы PIO. Такая небольшая система может встраиваться в дешевую игрушку или простой прибор. Вместо стираемого программируемого ПЗУ может использоваться обычное ПЗУ.

Микросхема PIO может быть выбрана одним из двух способов: как устройство ввода-вывода или как часть памяти. Если микросхема нам нужна в качестве устройства ввода-вывода, мы должны выбрать ее, используя внешнюю линию шины, которая показывает, что мы обращаемся к устройству ввода-вывода, а не к памяти.

Если применяется другой подход, так называемый ввод-вывод с распределением памяти, то следует присвоить микросхеме 4 байта памяти для трех портов и регистра управления. Выбор в какой-то степени произволен. Выбирается ввод-вывод с распределением памяти, поскольку этот метод наглядно иллюстрирует некоторые проблемы сопряжения.

Стираемому программируемому ПЗУ требуется 2К адресного пространства, ОЗУ требуется также 2К адресного пространства, а микросхеме PIO нужно 4 байта. Поскольку в примере адресное пространство составляет 64К, следует выбрать, где поместить данные три устройства.

5. АРХИТЕКТУРА СИСТЕМЫ КОМАНД

Уровень архитектуры команд расположен между микроархитектурным уровнем и уровнем операционной системы, как показано на рисунке 5.1. Исторически этот уровень развился прежде всех остальных уровней и изначально был единственным. В наши дни этот уровень очень часто называют архитектурой машины, а иногда (что неправильно) языком ассемблера. Уровень архитектуры команд имеет особое значение: он является связующим звеном между программным и аппаратным обеспечением. Конечно, можно было бы сделать так, чтобы аппаратное обеспечение сразу непосредственно выполняло программы, написанные на C, C++, FORTRAN 90 или других языках высокого уровня, но это не очень хорошая идея. Преимущество компиляции перед интерпретацией было бы тогда потеряно. Кроме того, из чисто практических соображений компьютеры должны уметь выполнять программы, написанные на разных языках, а не только на одном.

В сущности, все разработчики считают, что нужно транслировать программы, написанные на различных языках высокого уровня, в общую промежуточную форму — на уровень архитектуры команд — и соответственно конструировать аппаратное обеспечение, которое может непосредственно выполнять программы этого уровня (уровня архитектуры команд). Уровень архитектуры команд связывает компиляторы и аппаратное обеспечение. Это язык, который понятен и компиляторам, и аппаратному обеспечению. На рисунке 5.1 показана взаимосвязь компиляторов, уровня архитектуры команд и аппаратного обеспечения.

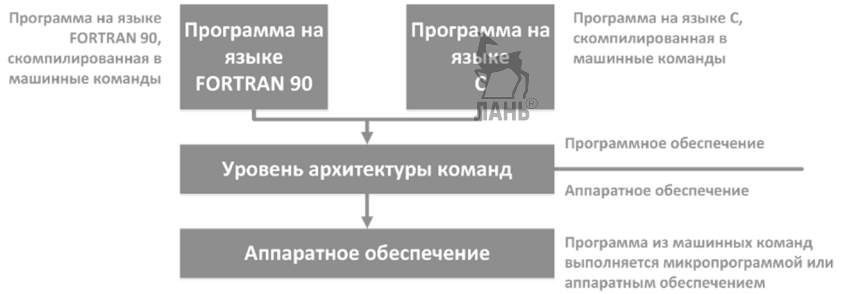


Рис. 5.1. Уровень команд — это промежуточное звено между компиляторами и аппартным обеспечением

В идеале при создании новой машины разработчики архитектуры команд должны консультироваться и с составителями компиляторов, и с теми, кто конструирует аппаратное обеспечение, чтобы выяснить, какими особенностями должен обладать уровень команд. Если составители компилятора требуют наличия какой-то особенности, которую инженеры не могут реализовать, то такая идея не пройдет.

Точно так же если разработчики аппаратного обеспечения хотят ввести в компьютер какую-либо новую особенность, но составители программного обеспечения не знают, как построить программу, чтобы использовать эту особенность, то такой проект никогда не будет воплощен. После долгих обсуждений и моделирования появится уровень команд, оптимизированный для нужных языков программирования, который и будет реализован.

Но все это в теории. На практике же, когда появляется новая машина, возникает вопрос обратной программной совместимости.

Этот факт заставляет компьютерных разработчиков сохранять один и тот же уровень команд в разных моделях или, по крайней мере, делать его обратно совместимым. Под обратной совместимостью понимается способность новой машины выполнять старые программы без изменений. Тем не менее новая машина может содержать новые команды и другие особенности, которые могут использоваться новым программным обеспечением. Разработчики должны делать уровень команд совместимым с предыдущими моделями, но они вправе творить все что угодно с аппаратным обеспечением, поскольку едва ли кого-нибудь из покупателей волнует, что представляет реальное собой аппаратное обеспечение и какие действия оно выполняет. Они могут переходить от микропрограммной разработки к непосредственному выполнению, добавлять конвейеры, суперскалярные устройства и т. п., при условии, что сохранится обратная совместимость с предыдущим уровнем команд. Основная цель — убедиться, что старые программы работают на новой машине. Тогда возникает проблема построения лучших машин, но с обратной совместимостью.

5.1. Общий обзор уровня архитектуры команд

Хорошо разработанный уровень архитектуры команд имеет огромные преимущества перед плохим, особенно в отношении вычислительных возможностей и стоимости. Производительность эквивалентных машин с раз-

личными уровнями команд может различаться на 25%. Рынок несколько затрудняет (хотя и не делает невозможным) устранение старой архитектуры команд и введение новой. Тем не менее иногда появляются новые уровни команд универсального назначения, а на специализированных рынках (например, на рынке встроенных систем или на рынке мультимедийных процессоров) они возникают гораздо чаще. Следовательно, необходимо понимать принципы разработки этого уровня. Существует два основных фактора оценки архитектуры команд.

Во-первых, хорошая архитектура должна определять набор команд, которые можно эффективно реализовать в современной и будущей технике, что приводит к рентабельным разработкам на несколько поколений. Плохой проект реализовать сложнее. При плохо разработанной архитектуре команд может потребоваться большее количество вентилей для процессора и больший объем памяти для выполнения программ. Кроме того, машина может работать медленнее, поскольку такая архитектура команд ухудшает возможности перекрывания операций, поэтому для достижения более высокой производительности здесь потребуется более сложный проект. Разработка, в которой используются особенности конкретной техники, может повлечь за собой производство целого поколения компьютеров, и эти компьютеры сможет опередить только более продвинутая архитектура команд.

Во-вторых, хорошая архитектура команд должна обеспечивать ясную цель для оттранслированной программы. Регулярность и полнота вариантов — важные черты, которые не всегда свойственны архитектуре команд. Эти качества важны для компилятора, которому трудно сделать лучший выбор из нескольких возможных, особенно когда некоторые очевидные на первый взгляд варианты не разрешены архитектурой команд. Если говорить кратко, поскольку уровень команд является промежуточным звеном между аппаратным и программным обеспечением, он должен быть удобен и для разработчиков аппаратного обеспечения, и для составителей программного обеспечения.

5.1.1. Свойства уровня команд

В принципе уровень команд — это то, каким представляется компьютер программисту машинного языка. Поскольку сейчас ни один нормальный человек не пишет программы на машинном языке, это определение необходимо изменить. Программа уровня архитектуры команд — это то,

что выдает компилятор (в данный момент мы игнорируем вызовы операционной системы и символический язык ассемблера). Чтобы произвести программу уровня команд, составитель компилятора должен знать, какая модель памяти используется в машине, какие регистры, типы данных и команды имеются в наличии и т. д. *Вся эта информация в совокупности и определяет уровень архитектуры команд*.

В соответствии с этим определением такие вопросы, как: программируется микроархитектура или нет, конвейеризирован компьютер или нет, является он суперскалярным или нет и т. д., не относятся к уровню архитектуры команд, поскольку составитель компилятора не видит всего этого. Однако это замечание не совсем справедливо, поскольку некоторые из этих свойств влияют на производительность, а производительность является видимой для программиста. Рассмотрим, например, суперскалярную машину, которая может выдавать back-to-back команды в одном цикле при условии, что одна команда целочисленная, а одна — с плавающей точкой. Если компилятор чередует целочисленные команды и команды с плавающей точкой, то производительность заметно улучшится. Таким образом, детали суперскалярной операции видны на уровне команд и границы между различными уровнями размыты.

Для одних архитектур уровень команд определяется формальным документом, который обычно выпускается промышленным консорциумом, для других — нет. Например, V9 SPARC (*Version 9 SPARC*) и JVM имеют официальные определения. Цель такого официального документа — дать возможность различным производителям выпускать машины данного конкретного вида, чтобы эти машины могли выполнять одни и те же программы и получать при этом одни и те же результаты. В случае с системой SPARC подобные документы нужны для того, чтобы различные предприятия могли выпускать идентичные микросхемы SPARC, отличающиеся друг от друга только производительностью и ценой. Чтобы эта идея работала, поставщики микросхем должны знать, что делает микросхема SPARC (на уровне команд). Следовательно, в документе говорится о том, какая модель памяти, какие регистры присутствуют, какие действия выполняют команды и т. д., а не о том, что представляет собой микроархитектура.

В таких документах содержатся нормативные разделы, в которых излагаются требования, и информативные разделы, которые предназначены для

того, чтобы помочь читателю, но не являются частью формального определения. В нормативных разделах описаны требования и запреты. Например, такое высказывание, как «выполнение зарезервированного кода операции должно вызывать системное прерывание» означает, что если программа выполняет код операции, который не определен, то он должен вызывать системное прерывание, а не просто игнорироваться. Может быть и альтернативный подход: результат выполнения зарезервированного кода операции определяется реализацией.

Это значит, что составитель компилятора не может просчитать какие-то конкретные действия, предоставляя конструкторам свободу выбора. К описанию архитектуры часто прилагаются тестовые комплекты для проверки, действительно ли данная реализация соответствует техническим требованиям.

Совершенно ясно, почему V9 SPARC имеет документ, в котором определяется уровень команд: это нужно для того, чтобы все микросхемы V9 SPARC могли выполнять одни и те же программы. По той же причине существует специальный документ для JVM: чтобы интерпретаторы (или такие микросхемы, как picojava II) могли выполнять любую допустимую программу JVM. Для уровня команд процессора Pentium II такого документа нет, поскольку компания *Intel* не хочет, чтобы другие производители смогли запускать микросхемы Pentium II. Компания *Intel* даже обращалась в суд, чтобы запретить производство своих микросхем другими предприятиями.

Другое важное качество уровня команд состоит в том, что в большинстве машин есть по крайней мере два режима. Привилегированный режим предназначен для запуска операционной системы. Он позволяет выполнять все команды. Пользовательский режим предназначен для запуска программных приложений. Этот режим не позволяет выполнять некоторые чувствительные команды (например, те, которые непосредственно манипулируют кэш-памятью).

5.2. Адресация и спецификация команд

Разработка кодов операций является важной частью архитектуры команд. Однако значительное число битов программы используется для того, чтобы определить, откуда нужно брать операнды, а не для того, чтобы узнать, какие операции нужно выполнить. Рассмотрим команду ADD, которая требует спецификации трех операндов: двух источников и одного пункта назначения. (Термин «операнд» обычно используется применительно ко всем трем элементам, хотя пункт назначения — это место, где

сохраняется результат.) Так или иначе, команда ADD должна сообщать, где найти операнды и куда поместить результат. Если адреса памяти 32-битные, то спецификация этой команды требует помимо кода операции еще три 32-битных адреса. Адреса занимают гораздо больше бит, чем коды операции.

Два специальных метода предназначены для уменьшения размера спецификации. Во-первых, если операнд должен использоваться несколько раз, его можно переместить в регистр. В использовании регистра для переменной есть двойная польза: скорость доступа увеличивается, а для определения операнда требуется меньшее количество битов. Если имеется 32 регистра, любой из них можно определить, используя всего лишь 5 битов. Если при выполнении команды ADD применять только регистровые операнды, для определения всех трех операндов понадобится только 15 битов, а если бы эти операнды находились в памяти, понадобилось бы целых 96 битов.

Однако использование регистров может вызвать другую проблему. Если операнд, находящийся в памяти, должен сначала загружаться в регистр, то потребуется большее число битов для определения адреса памяти. Во-первых, для переноса операнда в регистр нужна команда LOAD. Для этого требуется не только код операции, но и полный адрес памяти, а также нужно определить целевой регистр. Поэтому если операнд используется только один раз, помещать его в регистр не стоит.

К счастью, многочисленные измерения показали, что одни и те же операнды используются многократно. Поэтому большинство новых архитектур содержит большое количество регистров, а большинство компиляторов доходит до огромных размеров, чтобы хранить локальные переменные в этих регистрах, устраняя таким образом многочисленные обращения к памяти. Это сокращает и размер, и время выполнения программы.

Второй метод подразумевает определение одного или нескольких операндов неявным образом. Для этого существует несколько технологий. Один из способов — использовать одну спецификацию для входного и выходного операндов. В то время как обычная трехадресная команда ADD использует форму DESTINATION = SOURCE_1 + SOURCE_2.

Двухадресную команду можно сократить до формы REGISER_2 = REGISTER_2 + SOURCE_1.

Недостаток этой команды состоит в том, что содержимое REGISTER_2 не сохранится. Если первоначальное значение понадобится позднее, его нужно сначала скопировать в другой регистр. Компромисс здесь заключается

в том, что двухадресные команды короче, но они не так часто используются. У разных разработчиков разные предпочтения. В Pentium II, например, используются двухадресные команды, а в UltraSPARC II — трехадресные.

Первые компьютеры имели только один регистр, который назывался аккумулятором. Команда ADD, например, всегда прибавляла слово из памяти к аккумулятору, поэтому нужно было определять только один операнд (операнд памяти). Эта технология хорошо работала для простых вычислений, но, когда были нужны промежуточные результаты, аккумулятор приходилось записывать обратно в память, а позднее вызывать снова.

5.3. Способы адресации данных

Один из возможных вариантов интерпретации битов адресного поля для нахождения операнда состоит в том, что они содержат адрес операнда. Помимо огромного поля, необходимого для определения полного адреса памяти, данный метод имеет еще одно ограничение: этот адрес должен определяться во время компиляции. Существуют и другие возможности, которые обеспечивают более короткие спецификации, а также могут определять адреса динамически. В следующих разделах мы рассмотрим некоторые из этих форм, которые называются способами адресации.

5.3.1. Непосредственная адресация

Самый простой способ определения операнда — содержать в адресной части сам операнд, а не адрес операнда или какую-либо другую информацию, описывающую, где находится операнд. Такой операнд называется непосредственным операндом, поскольку он автоматически вызывается из памяти одновременно с командой; следовательно, он сразу непосредственно становится доступным. Один из вариантов команды с непосредственным адресом для загрузки в регистр R1 константы 4 показан на рисунке 5.2.

Рис. 5.2. Команда с нпосредственным адресом
для загрузки константы 4 в регистр 1

При непосредственной адресации не требуется дополнительного обращения к памяти для вызова операнда. Однако у такого способа адресации есть и некоторые недостатки. Во-первых, таким способом можно рабо-

тать только с константами. Во-вторых, число значений ограничено размером поля. Тем не менее эта технология используется во многих архитектурах для определения целочисленных констант.

5.3.2. Прямая адресация

Следующий способ определения операнда — просто дать его полный адрес. Такой способ называется прямой адресацией. Как и непосредственная адресация, прямая адресация имеет некоторые ограничения: команда всегда будет иметь доступ только к одному и тому же адресу памяти. То есть значение может меняться, а адрес — нет. Таким образом, прямая адресация может использоваться только для доступа к глобальным переменным, адреса которых известны во время компиляции. Многие программы содержат глобальные переменные, поэтому этот способ широко используется. Каким образом компьютер узнает, какие адреса непосредственные, а какие прямые, мы обсудим позже.

5.3.3. Регистровая адресация

Регистровая адресация по сути сходна с прямой адресацией, только в данном случае вместо ячейки памяти определяется регистр. Поскольку регистры очень важны (из-за быстрого доступа и коротких адресов), этот способ адресации является самым распространенным на большинстве компьютеров. Многие компиляторы доходят до огромных размеров, чтобы определить, к каким переменным доступ будет осуществляться чаще всего (например, индекс цикла), и помещают эти переменные в регистры.

Такой способ адресации называют регистровой адресацией. В архитектурах с загрузкой с запоминанием, например UltraSPARC II, практически все команды используют исключительно этот способ адресации. Он не используется только в том случае, когда операнд перемещается из памяти в регистр (команда LOAD) или из регистра в память (команда STORE). Даже в этих командах один из операндов является регистром — туда отправляется слово из памяти или оттуда перемещается слово в память.

5.3.4. Косвенная регистровая адресация

При таком способе адресации определяемый операнд берется из памяти или отправляется в память, но адрес не зафиксирован жестко в ко-

манде, как при прямой адресации. Вместо этого адрес содержится в регистре. Если адрес используется таким образом, он называется указателем. Преимущество косвенной адресации состоит в том, что можно обращаться к памяти, не имея в команде полного адреса. Кроме того, при разных выполнениях данной команды можно использовать разные слова памяти.

5.3.5. Индексная адресация

В машине IJVM локальные переменные определяются по смещению от регистра LV. Обращение к памяти по регистру и константе смещения называется индексной адресацией. В машине IJVM при доступе к локальной переменной используется указатель ячейки памяти (LV) в регистре плюс небольшое смещение в самой команде. Есть и другой способ: указатель ячейки памяти в команде и небольшое смещение в регистре. Чтобы показать, как это работает, рассмотрим следующий пример. У нас есть два одномерных массива A и B по 1024 слова в каждом. Нам нужно вычислить A_i «И» B_i для всех пар, а затем соединить все эти 1024 логических произведения операцией «ИЛИ», чтобы узнать, есть ли в этом наборе хотя бы одна пара, не равная нулю. Один из вариантов — поместить адрес массива A в один регистр, а адрес массива B — в другой регистр, а затем последовательно перебирать элементы массивов.

5.3.6. Относительная индексная адресация

В некоторых машинах применяется способ адресации, при котором адрес вычисляется путем суммирования значений двух регистров и смещения (смещение факультативно). Такой подход называется относительной индексной адресацией. Один из регистров — это база, а другой — это индекс. Такая адресация очень удобна при следующей ситуации. Вне цикла мы могли бы поместить адрес элемента A в регистр R5, а адрес элемента B в регистр R6. Тогда мы могли бы заменить две первые команды цикла LOOP на LOOP: MOV R4, (R2 + R5) AND R4, (R2 + R6).

5.3.7. Стековая адресация

Очень желательно сделать машинные команды как можно короче. Конечный предел в сокращении длины адреса — это команды без адресов. Безадресные команды, например ADD, возможны при наличии стека.

6. ОПЕРАЦИОННАЯ СИСТЕМА КОМПЬЮТЕРА

Современный компьютер состоит из ряда уровней, каждый из которых добавляет дополнительные функции к уровню, который находится под ним. Рассмотрен цифровой логический уровень, микроархитектурный уровень и уровень команд. Следующий уровень — уровень операционной системы (ОС).

Операционная система — это программа, которая добавляет ряд команд и особенностей к тем, которые обеспечиваются уровнем команд. Обычно операционная система реализуется главным образом в программном обеспечении, но нет никаких веских причин, по которым ее нельзя было бы реализовать в аппаратном обеспечении (как микропрограммы).

Хотя и уровень операционной системы, и уровень команд абстрактны (в том смысле, что они не являются реальным аппаратным обеспечением), между ними есть важное различие. Набор команд уровня операционной системы — это полный набор команд, доступных для прикладных программистов. Он содержит практически все команды более низкого уровня, а также новые команды, которые добавляет операционная система. Эти новые команды называются системными вызовами. Они вызывают определенную службу операционной системы, в частности одну из ее команд. Обычный системный вызов считывает какие-нибудь данные из файла.

Уровень операционной системы всегда интерпретируется. Когда пользовательская программа выполняет команду операционной системы, например чтение данных из файла, операционная система выполняет эту команду шаг за шагом, точно так же, как микропрограмма выполняет команду ADD. Однако когда программа выполняет команду уровня архитектуры команд, эта команда выполняется непосредственно микроархитектурным уровнем без участия операционной системы.

В этой книге мы можем лишь очень кратко в общих чертах рассказать вам об уровне операционной системы. Мы сосредоточимся на трех важных особенностях. Первая особенность — это виртуальная память. Виртуальная память используется многими операционными системами. Она позволяет создать впечатление, что у машины больше памяти, чем есть на самом деле. Вторая особенность — файл ввода-вывода. Это понятие более высокого уровня, чем команды ввода-вывода. Третья особенность — параллельная

обработка (как несколько процессов могут выполняться, обмениваться информацией и синхронизироваться). Под процессом можно понимать работающую программу и всю информацию о ее состоянии (о памяти, регистрах, счетчике команд, вводе-выводе и т. д.). После обсуждения этих основных характеристик мы покажем, как они применяются к операционным системам двух машин из трех наших примеров: Pentium II (Windows NT) и UltraSPARC II (UNIX). Поскольку picojava II обычно используется для встроенных систем, у этой машины нет операционной системы.

6.1. Технология виртуальной памяти

В первых компьютерах память была очень мала по объему и к тому же дорого стоила. IBM-650, ведущий компьютер того времени (конец 1950-х гг.), содержал всего 2000 слов памяти. Один из первых 60 компиляторов, ALGOL, был написан для компьютера с размером памяти всего 1024 слова. Древняя система с разделением времени прекрасно работала на компьютере PDP-1, общий размер памяти которого составлял всего 4096 18-битных слов для операционной системы и пользовательских программ. В те времена программисты тратили очень много времени на то, чтобы впихнуть свои программы в крошечную память. Часто приходилось использовать алгоритм, который работает намного медленнее другого алгоритма, поскольку лучший алгоритм был слишком большим по размеру и программа, в которой использовался этот лучший алгоритм, могла не поместиться в память компьютера.

Традиционным решением этой проблемы было использование вспомогательной памяти (например, диска). Программист делил программу на несколько частей, так называемых оверлеев, каждый из которых помещался в память. Чтобы прогнать программу, сначала следовало считывать и запускать первый оверлей.

Когда он завершался, нужно было считывать и запускать второй оверлей, и т. д. Программист отвечал за разбиение программы на оверлеи и решал, в каком месте вспомогательной памяти должен был храниться каждый оверлей, контролировал передачу оверлеев между основной и вспомогательной памятью и вообще управлял всем этим процессом без какой-либо помощи со стороны компьютера.

Хотя эта технология широко использовалась на протяжении многих лет, она требовала длительной кропотливой работы, связанной с управлением оверлеями. В 1961 г. группа исследователей из Манчестера (Англия)

предложила метод автоматического выполнения процесса наложения, при котором программист мог вообще не знать об этом процессе. Этот метод, который сейчас называется виртуальной памятью, имел очевидное преимущество, поскольку освобождал программиста от огромного количества нудной работы. Впервые этот метод был использован в ряде компьютеров, выпущенных в 1960-х гг. К началу 1970-х гг. виртуальная память появилась в большинстве компьютеров. В настоящее время даже компьютеры на одной микросхеме, в том числе Pentium II и UltraSPARC II, содержат очень сложные системы виртуальной памяти.

6.1.1. Страничная организация памяти

Группа ученых из Манчестера выдвинула идею о разделении понятий адресного пространства и адресов памяти. Рассмотрим в качестве примера типичный компьютер того времени с 16-битным полем адреса в командах и 4096 словами памяти.

Программа, работающая на таком компьютере, могла обращаться к 65 536 словам памяти (поскольку адреса были 16-битными, а 216 = 65 536). Обратите внимание, что число адресуемых слов зависит только от числа битов адреса и никак не связано с числом реально доступных слов в памяти. Адресное пространство такого компьютера состоит из чисел 0, 1, 2, ..., 65 535, так как это набор всех возможных адресов. Однако в действительности компьютер мог иметь гораздо меньше слов в памяти.

До изобретения виртуальной памяти приходилось проводить жесткое различие между теми адресами, которые меньше 4096, и теми, которые равны или больше 4096. Эти две части рассматривались как полезное адресное пространство и бесполезное адресное пространство соответственно (адреса выше 4095 были бесполезными, поскольку они не соответствовали реальным адресам памяти). Никакого различия между адресным пространством и адресами памяти не проводилось, поскольку между ними подразумевалось взаимно однозначное соответствие.

Идея разделения понятий адресного пространства и адресов памяти состоит в следующем. В любой момент времени можно получить прямой доступ к 4096 словам памяти, но это не значит, что они непременно должны соответствовать адресам памяти от 0 до 4095. Например, мы могли бы сообщить компьютеру, что при обращении к адресу 4096 должно использоваться слово из памяти с адресом 0, при обращении к адресу 4097 — слово

из памяти с адресом 1, при обращении к адресу 8191 — слово из памяти с адресом 4095 и т. д. Другими словами, мы определили отображение из адресного пространства в действительные адреса памяти (рис. 6.1).

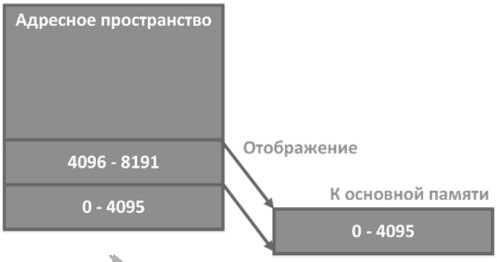


Рис. 6.1. Виртуальные адреса памяти с 4096 по 8191 отображаются в адреса основной памяти с 0 по 4095

Возникает интересный вопрос: а что произойдет, если программа совершит переход в один из адресов с 8192 по 12 287? В машине без виртуальной памяти произойдет ошибка, на экране появится фраза «Несуществующий адрес памяти» и выполнение программы остановится. В машине с виртуальной памятью будет иметь место следующая последовательность шагов:

1) слова с 4096 до 8191 будут размещены на диске;

2) слова с 8192 до 12 287 будут загружены в основную память;

3) отображение адресов изменится: теперь адреса с 8192 до 12 287 соответствуют ячейкам памяти с 0 по 4095;

4) выполнение программы будет продолжаться, как будто ничего ужасного не случилось.

Такая технология автоматического наложения называется страничной организацией памяти, а куски программы, которые считываются с диска, называются страницами.

Возможен и другой, более сложный способ отображения адресов из адресного пространства в реальные адреса памяти. Адреса, к которым программа может обращаться, мы будем называть виртуальным адресным пространством, а реальные адреса памяти в аппаратном обеспечении — физическим адресным пространством.

Схема распределения памяти или таблица страниц соотносит виртуальные адреса с физическими адресами. Предполагается, что на диске достаточно места для хранения полного виртуального адресного пространства

(или, по крайней мере, той его части, которая используется в данный момент).

Программы пишутся так, будто в основной памяти хватит места для размещения всего виртуального адресного пространства, даже если это не соответствует действительности. Программы могут загружать слова из виртуального адресного пространства или записывать слова в виртуальное адресное пространство, несмотря на то, что на самом деле физической памяти для этого не хватает. Программист может писать программы, даже не осознавая, что виртуальная память существует. Просто создается такое впечатление, что объем памяти данного компьютера достаточно велик.

Позднее мы сопоставим страничную организацию памяти с процессом сегментации, при котором программист должен знать о существовании сегментов. Еще раз подчеркнем, что страничная организация памяти создает иллюзию большой линейной основной памяти такого же размера, как адресное пространство. В действительности основная память может быть меньше (или больше), чем виртуальное адресное пространство. То, что память большого размера просто моделируется с помощью страничной организации памяти, нельзя определить по программе (только с помощью контроля синхронизации). При обращении к любому адресу всегда появляются требующиеся данные или нужная команда. Поскольку программист может писать программы и при этом ничего не знать о существовании страничной организации памяти, этот механизм называют прозрачным.

Ситуация, когда программист использует какой-либо виртуальный механизм и даже не знает, как он работает, не нова. В архитектуры команд, например, часто включается команда MUL (команда умножения), даже если в аппаратном обеспечении нет специального устройства для умножения. Иллюзия, что машина может перемножать числа, поддерживается микропрограммой. Точно так же операционная система может создавать иллюзию, что все виртуальные адреса поддерживаются реальной памятью, даже если это неправда. Только разработчикам операционных систем и тем, кто изучает операционные системы, нужно знать, как создается такая иллюзия.

6.1.2. Вызов страниц по требованию и рабочее множество

В основной памяти недостаточно места для всех виртуальных страниц. При обращении к адресу страницы, которой нет в основной памяти, проис-

ходит ошибка из-за отсутствия страницы. В случае такой ошибки операционная система должна считать нужную страницу с диска, ввести новый адрес физической памяти в таблицу страниц, а затем повторить команду, которая вызвала ошибку.

На машине с виртуальной памятью можно запустить программу даже в том случае, если ни одной части программы нет в основной памяти. Просто таблица страниц должна указывать, что абсолютно все виртуальные страницы находятся во вспомогательной памяти. Если центральный процессор пытается вызвать первую команду, он сразу получает ошибку из-за отсутствия страницы, в результате чего страница, содержащая первую команду, загружается в память и вносится в таблицу страниц. После этого начинается выполнение первой команды. Если первая команда содержит два адреса и оба эти адреса находятся на разных страницах, причем обе страницы не являются страницей, в которой находится команда, то произойдет еще две ошибки из-за отсутствия страницы и еще две страницы будут перенесены в основную память до завершения команды. Следующая команда может вызвать еще несколько ошибок и т. д.

Такой метод работы с виртуальной памятью называется вызовом страниц по требованию. Он похож на один из способов кормления младенцев: когда младенец кричит, вы его кормите (в противоположность кормлению по расписанию). При вызове страниц по требованию страницы переносятся в основную память только в случае необходимости, но не заранее.

Вопрос о том, стоит использовать вызов страниц по требованию или нет, имеет смысл только в самом начале при запуске программы. Когда программа проработает некоторое время, нужные страницы уже будут собраны в основной памяти. Если это компьютер с разделением времени и процессы откачиваются обратно, проработав примерно 100 млс, то каждая программа будет запускаться много раз. Для каждой программы распределение памяти уникально и при переключении с одной программы на другую меняется, поэтому в системах с разделением времени такой подход не годится.

Альтернативный подход основан на наблюдении, что большинство команд обращаются к адресному пространству неравномерно. Обычно большинство обращений относятся к небольшому числу страниц. При обращении к памяти можно вызвать команду, вызвать данные или сохранить данные.

В каждый момент времени *t* существует набор страниц, которые использовались при последних *k* обращениях. Деннинг назвал этот набор страниц рабочим множеством.

Поскольку рабочее множество обычно меняется очень медленно, можно, опираясь на последнее перед остановкой программы рабочее множество, предсказать, какие страницы понадобятся при новом запуске программы. Эти страницы можно загрузить заранее перед очередным запуском программы (предполагается, что предсказание будет точным).

6.1.3. Политика замещения страниц

В идеале набор страниц, которые постоянно используются программой (рабочее множество), можно хранить в памяти, чтобы сократить количество ошибок из-за отсутствия страниц. Однако программисты обычно не знают, какие страницы находятся в рабочем множестве, поэтому операционная система периодически должна показывать это множество. Если программа обращается к странице, которая отсутствует в основной памяти, ее нужно вызвать с диска. Однако чтобы освободить для нее место, на диск нужно отправить какую-нибудь другую страницу. Следовательно, нужен алгоритм для определения, какую именно страницу необходимо убрать из памяти.

Выбирать такую страницу просто наугад нельзя. Если, например, выбрать страницу, содержащую команду, выполнение которой вызвало ошибку, то при попытке вызвать следующую команду произойдет еще одна ошибка из-за отсутствия страницы. Большинство операционных систем стараются предсказать, какие из страниц в памяти наименее полезны в том смысле, что их отсутствие не повлияет сильно на ход программы. Иными словами, вместо того чтобы удалять страницу, которая скоро понадобится, следует выбрать такую страницу, которая не будет нужна долгое время.

По одному из алгоритмов удаляется та страница, которая использовалась наиболее давно, поскольку вероятность того, что она будет в текущем рабочем множестве, очень мала. Этот алгоритм называется LRU (*Least Recently Used* — алгоритм удаления наиболее давно использовавшихся элементов). Хотя этот алгоритм работает достаточно хорошо, иногда возникают патологические ситуации. Далее описана одна из них.

Представьте себе программу, выполняющую огромный цикл, который простирается на девять виртуальных страниц, а в физической памяти име-

ется место только для восьми страниц. Когда программа перейдет к странице 7, в основной памяти будут находиться страницы с 0 по 7 (табл. 6.1). Затем совершается попытка вызвать команду из виртуальной страницы 8, что вызывает ошибку из-за отсутствия страницы. Нужно принять решение, какую страницу убрать. По алгоритму LRU будет выбрана виртуальная страница 0, поскольку она использовалась раньше всех.

Таблица 6.1. Ситуация, в которой алгоритм LRU не действует (1)

Виртуальная страница 7
Виртуальная страница 6
Виртуальная страница 5
Виртуальная страница 4
Виртуальная страница 3
Виртуальная страница 2
Виртуальная страница 1
Виртуальная страница 0

Виртуальная страница 0 удаляется, а нужная виртуальная страница помещается на ее место (табл. 6.2).

Таблица 6.2. Ситуация, в которой алгоритм LRU не действует (2)

Виртуальная страница 7
Виртуальная страница 6
Виртуальная страница 5
Виртуальная страница 4
Виртуальная страница 3
Виртуальная страница 2
Виртуальная страница 1
Виртуальная страница 8

После выполнения команд из виртуальной страницы 8 программа возвращается к началу цикла, т. е. к виртуальной странице 0. Этот шаг вызывает еще одну ошибку из-за отсутствия страницы. Только что выброшенную виртуальную страницу 0 приходится вызывать обратно. По алгоритму LRU удаляется страница 1 (табл. 6.3). Через некоторое время программа пытается вызвать команду из виртуальной страницы 1, что опять вызывает ошибку. Затем вызывается страница 1 и удаляется страница 2 и т. д.

Очевидно, что в этой ситуации алгоритм LRU совершенно не работает (другие алгоритмы тоже не работают при сходных обстоятельствах). Од-

нако если расширить размер рабочего множества, число ошибок из-за отсутствия страниц станет минимальным. Можно применять и другой алгоритм — FIFO (*First-in First-out* — первым поступил, первым выводится). FIFO удаляет ту страницу, которая раньше всех загружалась, независимо от того, когда в последний раз производилось обращение к этой странице. С каждым страничным кадром связан отдельный счетчик. Изначально все счетчики установлены на 0.

Таблица 6.3. Ситуация, в которой алгоритм LRU не действует (3)

Виртуальная страница 7
Виртуальная страница 6
Виртуальная страница 5
Виртуальная страница 4
Виртуальная страница 3
Виртуальная страница 2
Виртуальная страница 0
Виртуальная страница 8

После каждой ошибки из-за отсутствия страниц счетчик каждой страницы, находящейся в памяти, увеличивается на 1, а счетчик только что вызванной страницы принимает значение 0. Когда нужно выбрать страницу для удаления, выбирается страница с самым большим значением счетчика. Поскольку она не загружалась в память очень давно, существует большая вероятность, что она больше не понадобится.

Если размер рабочего множества больше, чем число доступных страничных кадров, ни один алгоритм не дает хороших результатов и ошибки из-за отсутствия страниц будут происходить часто. Если программа постоянно вызывает подобные ошибки, то говорят, что наблюдается пробуксовка (*thrashing*). По всей видимости, не нужно объяснять, что пробуксовка очень нежелательна. Если программа использует большое виртуальное адресное пространство, но имеет небольшое медленно изменяющееся рабочее множество, которое помещается в основную память, ничего страшного не произойдет. Это утверждение имеет силу, даже если программа использует в сто раз больше слов виртуальной памяти чем их содержится в физической основной памяти.

Если страница, которую нужно удалить, не менялась с тех пор, как ее считали (а это вполне вероятно, если страница содержит программу, а не данные), то необязательно записывать ее обратно на диск, поскольку точная

копия там уже существует. Однако если эта страница изменилась, то копия на диске уже ей не соответствует и ее нужно туда переписать.

Если научиться определять, изменялась ли страница или не изменялась, то становится возможным избежать ненужных переписываний на диск и сэкономить много времени.

На многих машинах в контроллере управления памятью содержится один бит для каждой страницы, который равен 0 при загрузке страницы и принимает значение 1, когда микропрограмма или аппаратное обеспечение изменяют эту страницу. По этому биту операционная система определяет, менялась данная страница или нет и нужно ли ее перезаписывать на диск или нет.

6.1.4. Размер страниц и фрагментация

Если пользовательская программа и данные время от времени заполняют ровно целое число страниц, то, когда они находятся в памяти, свободного места там не будет. С другой стороны, если они не заполняют ровно целое число страниц, на последней странице останется неиспользованное пространство. Например, если программа и данные занимают 26 000 байтов на машине с 4096 байтами на страницу, то первые 6 страниц будут заполнены целиком, что в сумме даст (6 × 4096) = 24 576 байтов, а последняя страница будет содержать (26 000 – 24 576) = 1424 байта. Поскольку в каждой странице имеется пространство для 4096 байтов, 2672 байта останутся свободными. Всякий раз, когда седьмая страница присутствует в памяти, эти байты будут занимать место в основной памяти, но при этом не будут выполнять никакой функции. Эта проблема называется внутренней фрагментацией (поскольку неиспользованное пространство является внутренним по отношению к какой-то странице).

Если размер страницы составляет n байтов, то среднее неиспользованное пространство в последней странице программы будет $n/2$ байтов — ситуация подсказывает, что нужно использовать страницы небольшого размера, чтобы свести к минимуму количество неиспользованного пространства. С другой стороны, если страницы маленького размера, то потребуется много страниц и большая таблица страниц. Если таблица страниц сохраняется в аппаратном обеспечении, то для хранения большой таблицы страниц нужно много регистров, что повышает стоимость компьютера. Кроме того, при запуске и остановке программы на загрузку и сохранение этих регистров потребуется больше времени.

Более того, маленькие страницы снижают эффективность пропускной способности диска. Поскольку перед началом передачи данных с диска приходится ждать примерно 10 мс (поиск + время вращения), выгоднее совершать большие передачи. При скорости передачи данных 10 Мбайт в секунду передача 8 Кбайт добавляет всего 0,7 мс (по сравнению с передачей 1 Кбайт).

Однако у маленьких страниц есть свои преимущества. Если рабочее множество состоит из большого количества маленьких отделенных друг от друга областей виртуального адресного пространства, при маленьком размере страницы будет реже возникать пробуксовка (режим интенсивной подкачки), чем при большом. Рассмотрим матрицу A 10 000 × 10 000, которая хранится в последовательных 8-байтных словах. При такой записи элементы ряда будут начинаться на расстоянии 80 000 байтов друг от друга.

Программа, выполняющая вычисление над элементами этого ряда, будет использовать 10 000 областей, каждая из которых отделена от следующей 79 992 байтами. Если бы размер страницы составлял 8 Кбайт, то для хранения всех страниц понадобилось бы 80 Мбайт.

При размере страницы в 1 Кбайт для хранения всех страниц потребуется всего 10 Мбайт ОЗУ. При размере памяти в 32 Мбайт и размере страницы в 8 Кбайт программа войдет в режим интенсивной подкачки, а при размере страницы в 1 Кбайт этого не произойдет.

6.1.5. Сегментация

До сих пор речь шла об одномерной виртуальной памяти, в которой виртуальные адреса идут один за другим от 0 до какого-то максимального адреса. По многим причинам гораздо удобнее использовать два или несколько отдельных виртуальных адресных пространств. Например, компилятор может иметь несколько таблиц, которые создаются в процессе компиляции:

1) таблица символов, которая содержит имена и атрибуты переменных;

2) исходный текст, сохраняемый для распечатки;

3) таблица, содержащая все использующиеся целочисленные константы и константы с плавающей точкой;

4) дерево, содержащее синтаксический анализ программы;

5) стек, используемый для вызова процедур в компиляторе.

Каждая из первых четырех таблиц постоянно растет в процессе компиляции. Последняя таблица растет и уменьшается совершенно непредсказуемо. В одномерной памяти эти пять таблиц пришлось бы разместить в виртуальном адресном пространстве в виде смежных областей, как показано на рисунке 6.2.

Рис. 6.2. Наложение таблиц в одномерном адресном пространстве

Если программа содержит очень большое число переменных, то область адресного пространства, предназначенная для таблицы символов, может переполниться, даже если в других таблицах полно свободного места. Компилятор, конечно, может сообщить, что он не способен продолжать работу из-за большого количества переменных, но можно без этого и обойтись, поскольку в других таблицах много свободного места.

Компилятор может забирать свободное пространство у одних таблиц и передавать его другим таблицам, но это похоже на управление оверлеями вручную — некоторое неудобство в лучшем случае и долгая скучная работа в худшем.

На самом деле нужно просто освободить программиста от расширения и сокращения таблиц подобно тому, как виртуальная память исключает необходимость следить за разбиением программы на оверлеи.

Для этого нужно создать много абсолютно независимых адресных пространств, которые называются сегментами. Каждый сегмент состоит из линейной последовательности адресов от 0 до какого-либо максимума. Длина каждого сегмента может быть любой от 0 до некоторого допустимого максимального значения. Разные сегменты могут иметь разную длину (обычно так и бывает). Более того, длина сегмента может меняться во время выполнения программы. Длина стекового сегмента может увеличиваться всякий раз, когда что-либо помещается в стек, и уменьшаться, когда что-либо выталкивается из стека.

Так как каждый сегмент основывает отдельное адресное пространство, разные сегменты могут увеличиваться или уменьшаться независимо друг от друга и не влияя друг на друга. Если стеку в определенном сегменте понадобилось больше адресного пространства (чтобы стек мог увеличиться), он может получить его, поскольку в его адресном пространстве больше не во что «врезаться». Естественно, сегмент может переполниться, но это происходит редко, поскольку сегменты очень большие. Чтобы определить адрес в двухмерной памяти, программа должна выдавать номер сегмента и адрес внутри сегмента. На рисунке 6.3 изображена сегментированная память.

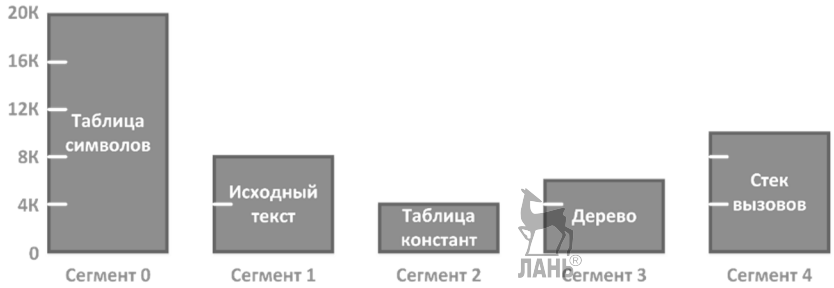


Рис. 6.3. Сегментированная память и независимое изменение размера таблиц

Сегмент является логическим элементом, о котором программист знает и который он использует. Сегмент может содержать процедуру, массив, стек или ряд скалярных переменных, но обычно в него входит только один тип данных.

Сегментированная память имеет и другие преимущества помимо упрощения работы со структурами данных, которые постоянно уменьшаются или увеличиваются. Если каждая процедура занимает отдельный сегмент, в котором первый адрес — это адрес 0, то связывание процедур, которые компилируются отдельно, сильно упрощается. Когда все процедуры программы скомпилированы и связаны, при вызове процедуры из сегмента *n* для обращения к слову 0 будет использоваться адрес (*n*, 0).

Если процедура в сегменте *n* впоследствии изменяется и перекомпилируется, то другие процедуры менять не нужно (поскольку ни один начальный адрес не был изменен), даже если новая версия больше по размеру, чем старая. В одномерной памяти процедуры обычно располагаются друг за другом, и между ними нет никакого адресного пространства. Следовательно, изменение размера одной процедуры может повлиять на начальный адрес других процедур. Это, в свою очередь, требует изменения всех процедур, которые вызывают любую из этих процедур, чтобы попадать в новые начальные адреса. Если в программу включено много процедур, этот процесс будет слишком накладным.

Сегментация облегчает разделение общих процедур или данных между несколькими программами. Если компьютер содержит несколько программ, работающих параллельно (это может быть реальная или смоделированная параллельная обработка), и если все эти программы используют определенные процедуры, снабжать каждую программу отдельной копией расточительно для памяти. А если сделать каждую процедуру отдельным сегментом, их легко можно будет разделять между несколькими программами, что исключит необходимость создавать физические копии каждой разделяемой процедуры. В результате мы сэкономим память.

Разные сегменты могут иметь разные виды защиты. Например, сегмент с процедурой можно определить как «только для выполнения», запретив тем самым считывание из него и запись в него. Для массива с плавающей точкой разрешается только чтение и запись, но не выполнение и т. д. Такая защита часто помогает обнаружить ошибки в программе.

Защита имеет смысл только в сегментированной памяти и не имеет никакого смысла в одномерной (линейной) памяти. Обычно в сегменте не могут содержаться процедура и стек одновременно (только что-нибудь одно). Поскольку каждый сегмент включает в себя объект только одного типа, он

может использовать защиту, подходящую для этого типа. Страничную организацию памяти и сегментацию можно сравнить (табл. 6.4).

Таблица 6.4. Сравнение страничной организации памяти и сегментации

Свойство	Страничная организация памяти	Сегментация
Необходимость знания программистом о наличии	–	+
Количество линейных адресных пространств в наличии	1	Много
Способность виртуального адресного пространства увеличивать размер памяти	+	+
Легкость управления таблицами с изменяющимися размерами	–	+
Назначение технологии	Моделирование памяти большого размера	Обеспечение нескольких адресных пространств

Содержимое страницы в каком-то смысле случайно. Программист может ничего не знать о страничной организации памяти. В принципе можно поместить несколько битов в каждый элемент таблицы страниц для доступа к нему, но, чтобы использовать эту особенность, программисту придется следить за границами страницы в адресном пространстве. Страничное разбиение памяти было придумано для устранения подобных трудностей. Поскольку у пользователя создается иллюзия, что все сегменты постоянно находятся в основной памяти, к ним можно обращаться и не следить при этом за оверлеями.

6.1.6. Виртуальная память и кэширование

На первый взгляд может показаться, что виртуальная память и кэширование никак не связаны, но на самом деле они сходны. При наличии виртуальной памяти вся программа хранится на диске и разбивается на страницы фиксированного размера. Некоторое подмножество этих страниц находится в основной памяти. Если программа главным образом использует страницы

из основной памяти, то ошибки из-за отсутствия страницы будут встречаться редко, и программа будет работать быстро. При кэшировании вся программа хранится в основной памяти и разбивается на блоки фиксированного размера. Некоторое подмножество этих блоков находится в кэш-памяти. Если программа главным образом использует блоки из кэш-памяти, то промахи кэш-памяти будут происходить редко и программа будет работать быстро. Как видим, виртуальная память и кэш-память идентичны, только работают они на разных уровнях иерархии. Естественно, виртуальная память и кэш-память имеют некоторые различия.

Промахи кэш-памяти обрабатываются аппаратным обеспечением, а ошибки из-за отсутствия страниц обрабатываются операционной системой. Блоки кэш-памяти обычно гораздо меньше страниц (например, 64 байта и 8 Кбайт). Кроме того, таблицы страниц индексируются по старшим битам виртуального адреса, а кэш-память индексируется по младшим битам адреса памяти. Тем не менее важно понимать, что различие здесь только в реализации.

7. ПАРАЛЛЕЛЬНЫЕ АРХИТЕКТУРЫ

Скорость работы компьютеров становится все выше, но и требования, предъявляемые к ним, тоже постоянно растут. Астрономы хотят воспроизвести всю историю Вселенной с момента Большого взрыва до самого конца. Фармацевты с помощью компьютеров хотели бы разрабатывать новые лекарственные препараты от конкретных заболеваний, не принося в жертву легионы крыс. Разработчики летательных аппаратов могли бы прийти к более эффективным результатам, если бы всю работу за них выполняли компьютеры, и тогда им не нужно было бы конструировать аэродинамическую трубу. Если говорить коротко, какими бы мощными ни были компьютеры, этого никогда не хватает для решения многих задач (особенно научных, технических и промышленных).

Скорость работы тактовых генераторов постоянно повышается, но скорость коммутации нельзя увеличивать бесконечно. Главной проблемой остается скорость света, в данный момент заставить протоны и электроны перемещаться быстрее невозможно. Из-за высокой теплоотдачи компьютеры превратились в кондиционеры. Наконец, поскольку размеры транзисторов постоянно уменьшаются, в какой-то момент времени каждый транзистор будет состоять из нескольких атомов, поэтому основной проблемой могут стать законы квантовой механики (например, гейзенберговский принцип неопределенности).

Чтобы решать более сложные задачи, разработчики обращаются к компьютерам параллельного действия. Невозможно построить компьютер с одним процессором и временем цикла в 0,001 нс, но зато можно построить компьютер с 1000 процессорами, время цикла каждого из которых составляет 1 нс. И хотя во втором случае мы используем процессоры, которые работают с более низкой скоростью, общая производительность теоретически должна быть такой же.

Параллелизм можно вводить на разных уровнях. На уровне команд, например, можно использовать конвейеры и суперскалярную архитектуру, что позволяет увеличивать производительность примерно в десять раз. Чтобы увеличить производительность в 100, 1000 или 1 000 000 раз, нужно продублировать процессор или, по крайней мере, какие-либо его части и заставить все эти части работать вместе.

Все эти машины состоят из элементов процессора и элементов памяти. Отличаются они друг от друга количеством элементов, их типом и способом взаимодействия элементов. В одних разработках используется небольшое число очень мощных элементов, а в других — огромное число элементов со слабой мощностью. Существуют промежуточные типы компьютеров. В области параллельной архитектуры проведена огромная работа.

7.1. Аспекты разработки параллельных систем

При рассмотрении новой компьютерной системы параллельного действия следует обратить внимание на три основных аспекта:

1) тип, размер и количество процессорных элементов;

2) тип, размер и количество модулей памяти;

3) принцип взаимодействия элементов памяти и процессорных элементов.

Процессорные элементы могут быть самых различных типов — от минимальных АЛУ до полных центральных процессоров, а по размеру один элемент может быть меньше небольшой части микросхемы и больше кубического метра электроники. Очевидно, что если процессорный элемент представляет собой часть микросхемы, то можно поместить в компьютер огромное число таких элементов (например, миллион). Если процессорный элемент представляет собой целый компьютер со своей памятью и устройствами ввода-вывода, цифры будут меньше, хотя были сконструированы такие системы даже с 10 000 процессорами. Сейчас компьютеры параллельного действия конструируются из серийно выпускаемых частей. Разработка компьютеров параллельного действия часто зависит от того, какие функции выполняют эти части и каковы ограничения.

Системы памяти часто разделены на модули, которые работают независимо друг от друга, чтобы несколько процессоров одновременно могли осуществлять доступ к памяти. Эти модули могут быть маленького размера (несколько килобайтов) или большого размера (несколько мегабайтов). Они могут находиться или рядом с процессорами, или на другой плате. Динамическая память (динамическое ОЗУ) работает гораздо медленнее центральных процессоров, поэтому для повышения скорости доступа к памяти обычно используются различные схемы кэш-памяти. Может быть два, три и даже четыре уровня кэш-памяти.

Существуют самые разнообразные процессоры и системы памяти, однако системы параллельного действия различаются в основном тем, как соединены разные части. Схемы взаимодействия можно разделить на две категории: статические и динамические. В статических схемах компоненты просто связываются друг с другом определенным образом. В качестве примеров статических схем можно привести звезду, кольцо и решетку. В динамических схемах все компоненты подсоединены к переключательной схеме, которая может трассировать сообщения между компонентами. У каждой из этих схем есть свои достоинства и недостатки.

Компьютеры параллельного действия можно рассматривать как набор микросхем, которые соединены друг с другом определенным образом. Это один подход.

При другом подходе возникает вопрос, какие именно процессы выполняются параллельно. Здесь существует несколько вариантов. Некоторые компьютеры параллельного действия одновременно выполняют несколько независимых задач. Эти задачи никак не связаны друг с другом и не взаимодействуют. Типичный пример — компьютер, содержащий от 8 до 64 процессоров, представляющий собой большую систему UNIX с разделением времени, с которой могут работать тысячи пользователей. В эту категорию попадают системы обработки транзакций, которые используются в банках (например, банковские автоматы), на авиалиниях (например, системы резервирования) и в больших веб-серверах. Сюда же относятся независимые прогоны моделирующих программ, при которых используется несколько наборов параметров.

Другие компьютеры параллельного действия выполняют одну задачу, состоящую из нескольких параллельных процессов. В качестве примера рассмотрим программу игры в шахматы, которая анализирует данные позиции на доске, порождает список возможных из этих позиций ходов, а затем порождает параллельные процессы, чтобы проанализировать каждую новую ситуацию параллельно. Здесь параллелизм нужен не для того, чтобы обслуживать большое количество пользователей, а для ускорения решения одной задачи.

Далее идут машины с высокой степенью конвейеризации или с большим количеством АЛУ, которые обрабатывают одновременно один поток команд. В эту категорию попадают суперкомпьютеры со специальным аппаратным обеспечением для обработки векторных данных. Здесь решается

одна главная задача, и при этом все части компьютера работают вместе над одним аспектом этой задачи (например, разные элементы двух векторов суммируются параллельно).

Эти три примера различаются по так называемой степени детализации. В многопроцессорных системах с разделением времени блок параллелизма достаточно велик — целая пользовательская программа. Параллельная работа больших частей программного обеспечения практически без взаимодействия между этими частями называется параллелизмом на уровне крупных структурных единиц. Диаметрально противоположный случай (при обработке векторных данных) называется параллелизмом на уровне мелких структурных единиц.

Термин «степень детализации» применяется по отношению к алгоритмам и программному обеспечению, но у него есть прямой аналог в аппаратном обеспечении. Системы с небольшим числом больших процессоров, которые взаимодействуют по схемам с низкой скоростью передачи данных, называются системами с косвенной (слабой) связью. Им противопоставляются системы с непосредственной (тесной) связью, в которых компоненты обычно меньше по размеру, расположены ближе друг к другу и взаимодействуют через специальные коммуникационные сети с высокой пропускной способностью. В большинстве случаев задачи с параллелизмом на уровне крупных структурных единиц лучше всего решаются в системах со слабой связью, а задачи с параллелизмом на уровне мелких структурных единиц лучше всего решаются в системах с непосредственной связью. Однако существует множество различных алгоритмов и множество разнообразного программного и аппаратного обеспечения. Разнообразие степени детализации и возможности различной степени связности систем привели к многообразию параллельных архитектур.

7.2. Информационные модели параллелизма

В любой системе параллельной обработки процессоры, выполняющие разные части одной задачи, должны как-то взаимодействовать друг с другом, чтобы обмениваться информацией. Как именно должен происходить этот обмен? Были предложены и реализованы две разработки: мультипроцессоры и мультикомпьютеры. Мы рассмотрим их далее.

7.2.1. Мультипроцессоры

В первой разработке все процессоры разделяют общую физическую память, как показано на рисунке 7.1*а*. Такая система называется мультипроцессором или системой с совместно используемой памятью. Мультипроцессорная модель распространяется на программное обеспечение.

Рис. 7.1. Мультипроцессор, содержащий 16 процессоров, которые разделяют общую память (*а*); изображение, разбитое на 16 секций, каждую из которых анализирует отдельный процессор (*б*)

Все процессы, работающие вместе на мультипроцессоре, могут разделять одно виртуальное адресное пространство, отображенное в общую память. Любой процесс может считывать слово из памяти или записывать слово в память с помощью команд LOAD и STORE. Больше ничего не требуется. Два процесса могут обмениваться информацией, если один из них будет просто записывать данные в память, а другой будет считывать эти данные.

Благодаря такой возможности взаимодействия двух и более процессов мультипроцессоры весьма популярны. Данная модель понятна программистам и применима к широкому кругу задач. Рассмотрим программу, которая изучает битовое отображение и составляет список всех его объектов. Одна копия отображения хранится в памяти, как показано на рисунке 7.1*б*. Каждый из 16 процессоров запускает один процесс, которому приписана для анализа одна из 16 секций. Если процесс обнаруживает, что один из его объектов переходит через границу секции, этот процесс просто переходит вслед

за объектом в следующую секцию, считывая слова этой секции. В нашем примере некоторые объекты обрабатываются несколькими процессами, поэтому в конце потребуется некоторая координация, чтобы определить количество домов, деревьев и самолетов.

В качестве примеров мультипроцессоров можно назвать Sun Enterprise 10000, Sequent NUMA-Q, SGI Origin 2000 и HP/Convex Exemplar.

7.2.2. Мультикомпьютеры

Во втором типе параллельной архитектуры каждый процессор имеет свою собственную память, доступную только этому процессору. Такая разработка называется мультикомпьютером или системой с распределенной памятью. Она изображена на рисунке 7.2*а*. Мультикомпьютеры обычно (хотя не всегда) являются системами со слабой связью. Ключевое отличие мультикомпьютера от мультипроцессора состоит в том, что каждый процессор в мультикомпьютере имеет свою собственную локальную память, к которой этот процессор может обращаться, выполняя команды LOAD и STORE, но никакой другой процессор не может получить доступ к этой памяти с помощью тех же команд LOAD и STORE. Таким образом, мультипроцессоры имеют одно физическое адресное пространство, разделяемое всеми процессорами, а мультикомпьютеры содержат отдельное физическое адресное пространство для каждого центрального процессора.

Рис. 7.2. Мультикомпьютер, содержащий 16 процессоров, каждый из которых имеет собственную память (*а*); битовое отображение рисунка 7.1, разделенное между участками памяти (*б*)

Поскольку процессоры в мультикомпьютере не могут взаимодействовать друг с другом просто путем чтения из общей памяти и записи в общую память, здесь необходим другой механизм взаимодействия. Они посылают друг другу сообщения, используя сеть межсоединений. В качестве примеров мультикомпьютеров можно назвать IBM SP/2, Intel/Sandia Option Red и Wisconsin COW.

При отсутствии памяти совместного использования в аппаратном обеспечении предполагается определенная структура программного обеспечения. В мультикомпьютере невозможно иметь одно виртуальное адресное пространство, из которого все процессы могут считывать информацию и в которое все процессы могут записывать информацию просто путем выполнения команд LOAD и STORE. Например, если процессор 0 (в верхнем левом углу) на рисунке 7.1, обнаруживает, что часть его объекта попадает в другую секцию, относящуюся к процессору 1, он может продолжать считывать информацию из памяти, чтобы получить хвост самолета. Однако если процессор 0 на рисунке 7.2*а* обнаруживает это, он не может просто считать информацию из памяти процессора 1. Для получения необходимых данных ему нужно сделать что-то другое.

В частности, ему нужно как-то определить, какой процессор содержит необходимые ему данные, и послать этому процессору сообщение с запросом копии данных.

Затем процессор 0 блокируется до получения ответа. Когда процессор 1 получает сообщение, программное обеспечение должно проанализировать его и отправить назад необходимые данные. Когда процессор 0 получает ответное сообщение, программное обеспечение разблокируется и продолжает работу.

В мультикомпьютере для взаимодействия между процессорами часто используются примитивы SEND и RECEIVE. Поэтому программное обеспечение мультикомпьютера имеет более сложную структуру, чем программное обеспечение мультипроцессора. При этом основной проблемой становится правильное разделение данных и разумное их размещение. В мультипроцессоре размещение частей не влияет на правильность выполнения задачи, хотя может повлиять на производительность. Таким образом, мультикомпьютер программировать гораздо сложнее, чем мультипроцессор.

Намного проще и дешевле построить большой мультикомпьютер, чем мультипроцессор с таким же количеством процессоров. Реализация общей памяти, разделяемой несколькими сотнями процессоров, — это весьма сложная задача, а построить мультикомпьютер, содержащий 10 000 процессоров и более, довольно легко.

Таким образом происходит столкновение с дилеммой: мультипроцессоры сложно строить, но легко программировать, а мультикомпьютеры легко строить, но трудно программировать. Поэтому стали предприниматься попытки создания гибридных систем, которые относительно легко конструировать и относительно легко программировать. Это привело к осознанию того, что совместную память можно реализовывать по-разному, и в каждом случае будут какие-то преимущества и недостатки.

Практически все исследования в области архитектур с параллельной обработкой направлены на создание гибридных форм, которые сочетают в себе преимущества обеих архитектур. Здесь важно получить такую систему, которая расширяема, т. е. которая будет продолжать исправно работать при добавлении все новых и новых процессоров.

Один из подходов основан на том, что современные компьютерные системы не монолитны, а состоят из ряда уровней. В данном подходе память совместного использования реализована в аппаратном обеспечении в виде реального мультипроцессора. В данной разработке имеется одна копия операционной системы с одним набором таблиц, в частности таблицей распределения памяти. Если процессу требуется больше памяти, он прерывает работу операционной системы, которая после этого начинает искать в таблице свободную страницу и отображает эту страницу в адресное пространство вызывающей программы. Что касается операционной системы, имеется единая память, и операционная система следит, какая страница в программном обеспечении принадлежит тому или иному процессу. Существует множество способов реализации совместной памяти в аппаратном обеспечении.

Второй подход — использовать аппаратное обеспечение мультикомпьютера и операционную систему, которая моделирует разделенную память, обеспечивая единое виртуальное адресное пространство, разбитое на страницы. При таком подходе, который называется DSM (*Distributed Shared Memory* — распределенная совместно используемая память), каждая страница расположена в одном из блоков памяти (рис. 7.2*б*). Каждая машина содержит свою собственную вир-

туальную память и собственные таблицы страниц. Если процессор совершает команду LOAD или STORE над страницей, которой у него нет, происходит прерывание операционной системы. Затем операционная система находит нужную страницу и требует, чтобы процессор, который обладает нужной страницей, преобразовал ее в исходную форму и послал по сети межсоединений. Когда страница достигает пункта назначения, она отображается в память и выполнение прерванной команды возобновляется. По существу, операционная система просто вызывает недостающие страницы не с диска, а из памяти. Но у пользователя создается впечатление, что машина содержит общую разделенную память.

Третий подход — реализовать общую разделенную память на уровне программного обеспечения. При таком подходе абстракцию разделенной памяти создает язык программирования, и эта абстракция реализуется компилятором. Например, модель Linda основана на абстракции разделенного пространства кортежей (записей данных, содержащих наборы полей). Процессы любой машины могут взять кортеж из общего пространства или отправить его в общее пространство. Поскольку доступ к этому пространству полностью контролируется программным обеспечением (системой Linda), никакого специального аппаратного обеспечения или специальной операционной системы не требуется.

Рис. 7.3. Уровни, на которых можно реализовать память совместного использования: аппаратное обеспечение (*а*); операционная система (*б*); программное обеспечение (*в*)

Другой пример памяти совместного использования, реализованной в программном обеспечении, — модель общих объектов в системе Orca. В модели Orca процессы разделяют объекты, а не кортежи, и могут выполнять над ними те или иные процедуры. Если процедура изменяет внутреннее состояние объекта, операционная система должна проследить, чтобы все копии этого объекта на всех машинах одновременно были изменены. Поскольку объекты — это чисто программное понятие, их можно реализовать с помощью программного обеспечения без вмешательства операционной системы или аппаратного обеспечения.

7.3. Технология сетей межсоединений

На рисунке 7.2 показано, что мультикомпьютеры связываются через сети межсоединений. Мультикомпьютеры и мультипроцессоры очень сходны в этом отношении, поскольку мультипроцессоры часто содержат несколько модулей памяти, которые также должны быть связаны друг с другом и с процессорами.

Основная причина сходства коммуникационных связей в мультипроцессоре и мультикомпьютере заключается в том, что в обоих случаях применяется передача сообщений. Даже в однопроцессорной машине, когда процессору нужно считать или записать слово, он устанавливает определенные линии на шине и ждет ответа. Это действие представляет собой то же самое, что и передача сообщений: инициатор посылает запрос и ждет ответа. В больших мультипроцессорах взаимодействие между процессорами и удаленной памятью почти всегда состоит в том, что процессор посылает в память сообщение, так называемый пакет, который запрашивает определенные данные, а память посылает процессору ответный пакет. Сети межсоединений могут состоять максимум из пяти компонентов:

– центральные процессоры;
– модули памяти;
– интерфейсы;
– каналы связи;
– коммутаторы.

Интерфейсы — это устройства, которые вводят и выводят сообщения из центральных процессоров и модулей памяти. Во многих разработках интерфейс представляет собой микросхему или плату, к которой подсоединя-

ется локальная шина каждого процессора и которая может передавать сигналы процессору и локальной памяти (если таковая есть). Часто внутри интерфейса содержится программируемый процессор со своим собственным ПЗУ, которое принадлежит только этому процессору. Обычно интерфейс способен считывать и записывать информацию в различные модули памяти, что позволяет ему перемещать блоки данных.

Каналы связи — это каналы, по которым перемещаются биты. Каналы могут быть электрическими или оптико-волоконными, последовательными (шириной 1 бит) или параллельными (шириной более 1 бита). Каждый канал связи характеризуется максимальной пропускной способностью (это максимальное число битов, которое он способен передавать в секунду). Каналы могут быть симплексными (передавать биты только в одном направлении), полудуплексными (передавать информацию в обоих направлениях, но не одновременно) и дуплексными (передавать биты в обоих направлениях одновременно).

Коммутаторы — это устройства с несколькими входными и несколькими выходными портами. Когда на входной порт приходит пакет, некоторые биты в этом пакете используются для выбора выходного порта, в который посылается пакет. Размер пакета может составлять 2 или 4 байта, но может быть и значительно больше (например, 8 Кбайт).

Сети межсоединений можно сравнить с улицами города. Улицы похожи на каналы связи. Каждая улица может быть с односторонним и двусторонним движением, она характеризуется определенной «скоростью передачи данных» (имеется в виду ограничение скорости движения) и имеет определенную ширину (число рядов). Перекрестки похожи на коммутаторы. На каждом перекрестке прибывающий пакет (пешеход или машина) выбирает, в какой выходной порт (улицу) поступить дальше в зависимости от того, каков конечный пункт назначения.

При разработке и анализе сети межсоединений важно учитывать несколько ключевых моментов. Во-первых, это топология (т. е. способ расположения компонентов). Во-вторых, это то, как работает система переключения и как осуществляется связь между ресурсами. В-третьих, какой алгоритм выбора маршрута используется для доставки сообщений в пункт назначения. Далее мы рассмотрим каждый из этих пунктов.

7.3.1. Топология

Топология сети межсоединений определяет, как расположены каналы связи и коммутаторы (это, например, может быть кольцо или решетка). Топологии можно изображать в виде графов, в которых дуги соответствуют каналам связи, а узлы — коммутаторам (рис. 7.4). С каждым узлом в сети (или в соответствующем графе) связан определенный ряд каналов связи. Математики называют число каналов степенью узла, инженеры — коэффициентом разветвления. Чем больше степень, тем больше вариантов маршрута и тем выше отказоустойчивость. Если каждый узел содержит k дуг и соединение сделано правильно, то можно построить сеть межсоединений так, чтобы она оставалась полносвязной, даже если $(k - 1)$ каналов повреждены.

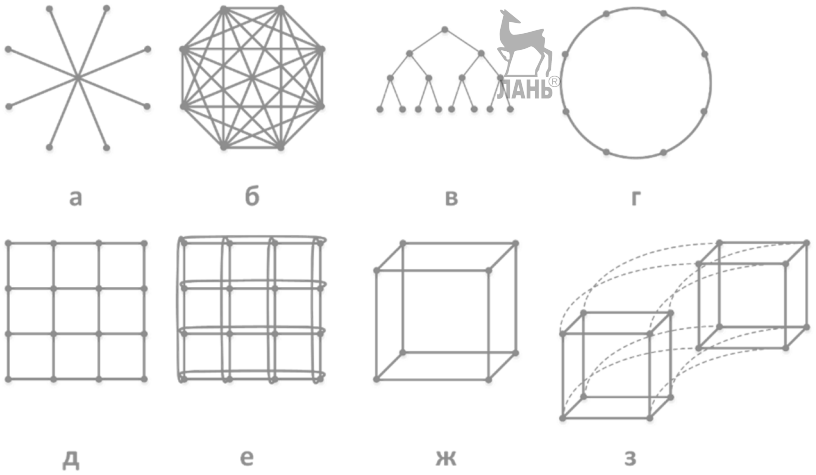


Рис. 7.4. Топология сетей межсоединений; звезда (*а*); полное межсоединение (*б*); дерево (*в*); кольцо (*г*); решетка (*д*); двойной тор (*е*); куб (*ж*) и гиперкуб (*з*)

Следующее свойство сети межсоединений — это ее диаметр. Если расстоянием между двумя узлами мы будем считать число дуг, которые нужно пройти, чтобы попасть из одного узла в другой, то диаметром графа будет расстояние между двумя узлами, которые расположены дальше всех друг от друга. Диаметр сети определяет самую большую задержку при передаче пакетов от одного процессора к другому или от процессора к памяти, по-

скольку каждая пересылка через канал связи занимает определенное количество времени. Чем меньше диаметр, тем выше производительность. Также имеет большое значение среднее расстояние между двумя узлами, поскольку от него зависит среднее время передачи пакета.

Еще одно важное свойство сети межсоединений — это ее пропускная способность, т. е. количество данных, которое она способна передавать в секунду. Очень важная характеристика — бисекционная пропускная способность. Чтобы ее вычислить, нужно мысленно разделить сеть межсоединений на две равные (с точки зрения числа узлов) несвязанные части путем удаления ряда дуг из графа. Затем нужно вычислить общую пропускную способность дуг, которые мы удалили. Существует множество способов разделения сети межсоединений на две равные части. Бисекционная пропускная способность — минимальная из всех возможных. Предположим, что бисекционная пропускная способность составляет 800 бит/с. Тогда, если между двумя частями много взаимодействий, общую пропускную способность в худшем случае можно сократить до 800 бит/с. По мнению многих разработчиков, бисекционная пропускная способность — это самая важная характеристика сети межсоединений. Часто основная цель при разработке сети межсоединений — сделать бисекционную пропускную способность максимальной.

Сети межсоединений можно характеризовать по их размерности. Размерность определяется по числу возможных вариантов перехода из исходного пункта в пункт назначения. Если выбора нет (т. е. существует только один путь из каждого исходного пункта в каждый конечный пункт), то сеть нульмерная. Если есть два возможных варианта (например, если можно пойти либо направо, либо налево), то сеть одномерна. Если есть две оси и пакет может направиться направо или налево либо вверх или вниз, то такая сеть двумерна и т. д.

На рисунке 7.4 показано несколько топологий. Здесь изображены только каналы связи (это линии) и коммутаторы (это точки). Модули памяти и процессоры (они на рисунке не показаны) подсоединяются к коммутаторам через интерфейсы. На рисунке 7.4*а* изображена нульмерная конфигурация звезда, где процессоры и модули памяти прикрепляются к внешним узлам, а переключение совершает центральный узел. Такая схема очень проста, но в большой системе центральный коммутатор будет главным критическим параметром, который ограничивает производительность системы.

И с точки зрения отказоустойчивости это очень неудачная разработка, поскольку одна ошибка в центральном коммутаторе может разрушить всю систему.

На рисунке 7.4*б* изображена другая нульмерная топология — полное межсоединение (*full interconnect*). Здесь каждый узел непосредственно связан с каждым имеющимся узлом. В такой разработке пропускная способность между двумя секциями максимальна, диаметр минимален, а отказоустойчивость очень высока (даже при утрате шести каналов связи система все равно будет полностью взаимосвязана). Однако для k-узлов требуется $k(k-1)/2$ каналов, а это совершенно неприемлемо для больших значений k.

На рисунке 7.4*в* изображена третья нульмерная топология — дерево. Здесь основная проблема состоит в том, что пропускная способность между секциями равна пропускной способности каналов. Обычно у верхушки дерева наблюдается очень плотный информационный трафик, поэтому верхние узлы становятся препятствием для повышения производительности. Можно разрешить эту проблему, увеличив пропускную способность верхних каналов. Например, самые нижние каналы будут иметь пропускную способность S, следующий уровень — пропускную способность $2S$, а каждый канал верхнего уровня — пропускную способность $4S$. Такая схема называется толстым деревом (fat tree). Она применялась в коммерческих мультикомпьютерах Thinking Machines CM-5.

Кольцо (рис. 7.4*г*) — это одномерная топология, поскольку каждый отправленный пакет может пойти направо или налево. Решетка (или сетка) (рис. 7.4*д*) — это двумерная топология, которая применяется во многих коммерческих системах. Она отличается регулярностью и применима к системам большого размера, а диаметр составляет квадратный корень от числа узлов (т. е. при расширении системы диаметр увеличивается незначительно). Двойной тор (рис. 7.4*е*) является разновидностью решетки. Это решетка, у которой соединены края. Она характеризуется большей отказоустойчивостью и меньшим диаметром, чем обычная решетка, поскольку теперь между двумя противоположными узлами всего два транзитных участка.

Куб (рис. 7.4*ж*) — это правильная трехмерная топология. На рисунке изображен куб 2×2×2, но в общем случае он может быть $k \times k \times k$. На рисунке 7.4*з* показан четырехмерный куб, полученный из двух трехмерных

кубов, которые связаны между собой. Можно сделать пятимерный куб, соединив вместе четыре четырехмерных куба.

Чтобы получить шесть измерений, нужно продублировать блок из четырех кубов и соединить соответствующие узлы, и т. д.; *n*-мерный куб называется гиперкубом. Эта топология используется во многих компьютерах параллельного действия, поскольку ее диаметр находится в линейной зависимости от размерности. Другими словами, диаметр — это логарифм по основанию 2 от числа узлов, поэтому 10-мерный гиперкуб имеет 1024 узла, но диаметр равен всего 10, что дает очень незначительные задержки при передаче данных. Решетка 32×32, которая также содержит 1024 узла, имеет диаметр 62, что более чем в шесть раз превышает диаметр гиперкуба. Однако чем меньше диаметр гиперкуба, тем больше разветвление и число каналов (и, следовательно, тем выше стоимость). Тем не менее в системах с высокой производительностью чаще всего используется именно гиперкуб.

7.3.2. Коммутация

Сеть межсоединений состоит из коммутаторов и проводов, соединяющих их. На рисунке 7.5 изображена небольшая сеть межсоединений с четырьмя коммутаторами. В данном случае каждый коммутатор имеет четыре входных порта и четыре выходных порта. Кроме того, каждый коммутатор содержит несколько центральных процессоров и схемы соединения (на рисунке они показаны не полностью). Задача коммутатора — принимать пакеты, которые приходят на любой входной порт, и отправлять пакеты из соответствующих выходных портов.

Каждый выходной порт связан с входным портом другого коммутатора через последовательный или параллельный канал (на рис. 7.5 это пунктирная линия).

Последовательные каналы передают 1 бит единовременно. Параллельные каналы могут передавать несколько битов сразу. Существуют специальные сигналы для управления каналом. Параллельные каналы характеризуются более высокой производительностью, чем последовательные каналы с такой же тактовой частотой, но в них возникает проблема расфазировки данных (нужно быть уверенным, что все биты прибывают одновременно), и они стоят гораздо дороже.

Существует несколько стратегий переключения. Первая из них — коммутация каналов. Перед тем как послать пакет, весь путь от начального до

конечного пункта резервируется заранее. Все порты и буферы затребованы заранее, поэтому, когда начинается процесс передачи, все необходимые ресурсы гарантированно доступны и биты могут на полной скорости перемещаться от исходного пункта через все коммутаторы к пункту назначения. На рисунке 7.5 показана коммутация каналов, где резервируется канал от процессора 1 к процессору 2 (черная жирная стрелка). Здесь резервируется три входных и три выходных порта.

Рис. 7.5. Сеть межсоединений в форме квадратной решетки с четырьмя коммутаторами

Коммутацию каналов можно сравнить с перекрытием движения транспорта во время парада, когда блокируются все прилегающие улицы. При этом требуется предварительное планирование, но после блокирования прилегающих улиц парад может продвигаться на полной скорости, поскольку никакой транспорт препятствовать этому не будет. Недостаток такого метода состоит в том, что требуется предварительное планирование и любое движение транспорта запрещено, даже если парад (или пакеты) еще не приближается.

Вторая стратегия — коммутация с промежуточным хранением. Здесь не требуется предварительного резервирования. Из исходного пункта посылается целый пакет к первому коммутатору, где он хранится целиком. На

рисунке 7.6*а* исходным пунктом является процессор 1, а весь пакет, который направляется в процессор *2*, сначала сохраняется внутри коммутатора *A*. Затем этот пакет перемещается в коммутатор *C*, как показано на рисунке 7.6*б*. Затем весь пакет целиком перемещается в коммутатор *D* (рис. 7.6*в*). Наконец, пакет доходит до пункта назначения — до процессора 2. Отметим, что никакого предварительного резервирования ресурсов не требуется.

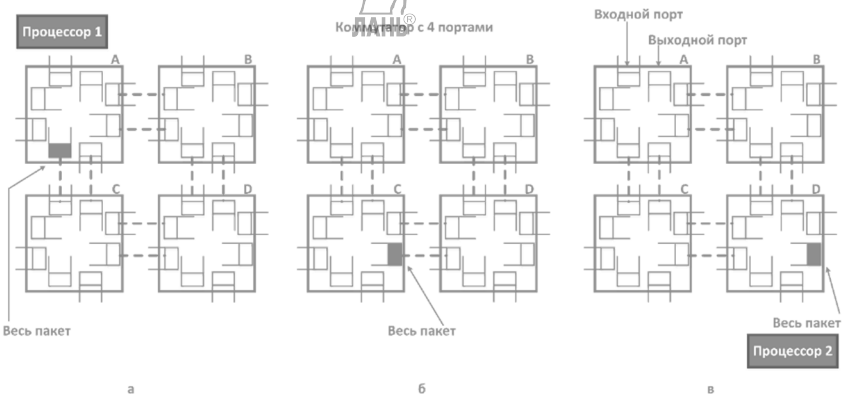


Рис. 7.6. Этапы коммутации с промежуточным хранением на исходном (*а*), промежуточном (*б*) и конечном (*в*) коммутаторе

Коммутаторы с промежуточным хранением должны отправлять пакеты в буфер, поскольку, когда исходный пункт (например, процессор, память или коммутатор) выдает пакет, требующийся выходной порт может быть в данный момент занят передачей другого пакета. Если бы не было буферизации, входящие пакеты, которым нужен занятый в данный момент выходной порт, пропадали бы. Применяется три метода буферизации. При буферизации на входе один или несколько буферов связываются с каждым входным портом в форме очереди типа FIFO («первым вошел, первым вышел»). Если пакет в начале очереди нельзя передать по причине занятости нужного выходного порта, этот пакет просто ждет своей очереди.

Однако если пакет ожидает, когда освободится выходной порт, то пакет, идущий за ним, тоже не может передаваться, даже если нужный ему порт свободен. Ситуация называется блокировкой начала очереди. Проиллюстрируем ситуацию на примере. Представим дорогу из двух рядов. Вереница машин в одном из рядов не может двигаться дальше, поскольку первая машина в этом ряду хочет повернуть налево, но не может из-за движения

машин другого ряда. Даже если второй и всем следующим за ней машинам нужно ехать прямо, первая машина в ряду препятствует их движению.

Проблему можно устранить с помощью буферизации на выходе. В этой системе буферы связаны с выходными портами. Биты пакета по мере пребывания сохраняются в буфере, который связан с нужным выходным портом. Поэтому пакеты, направленные в порт *t*, не могут блокировать пакеты, направленные в порт *n*.

И при буферизации на входе, и при буферизации на выходе с каждым портом связано определенное количество буферов. Если места недостаточно для хранения всех пакетов, то какие-то пакеты придется выбрасывать. Чтобы разрешить эту проблему, можно использовать общую буферизацию, при которой один буферный пул динамически распределяется по портам по мере необходимости. Однако такая схема требует более сложного управления, чтобы следить за буферами, и позволяет одному занятому соединению захватить все буферы, оставив другие соединения ни с чем. Кроме того, каждый коммутатор должен вмещать самый большой пакет и даже несколько пакетов максимального размера, а для этого потребуется ужесточить требования к памяти и снизить максимальный размер пакета.

Хотя метод коммутации с промежуточным хранением гибок и эффективен, здесь возникает проблема возрастающей задержки при передаче данных по сети межсоединений. Предположим, что время, необходимое для перемещения пакета по одному транзитному участку на рисунке 7.6, занимает *T* нс. Чтобы переместить пакет из процессора 1 в процессор 2, нужно скопировать его четыре раза (в *A*, в *C*, в *D* и в процессор 2), и следующее копирование не может начаться, пока не закончится предыдущее, поэтому задержка по сети составляет 4*T*. Чтобы выйти из этой ситуации, нужно разработать гибридную сеть межсоединений, объединяющую в себе коммутацию каналов и коммутацию пакетов. Например, каждый пакет можно разделить на части. Как только первая часть поступила в коммутатор, ее можно сразу направить в следующий коммутатор, даже если оставшиеся части пакета еще не прибыли в этот коммутатор.

Такой подход отличается от коммутации каналов тем, что ресурсы не резервируются заранее. Следовательно, возможна конфликтная ситуация в соревновании за право обладания ресурсами (портами и буферами). При коммутации без буферизации пакетов, если первый блок пакета не может

двигаться дальше, оставшаяся часть пакета продолжает поступать в коммутатор. В худшем случае эта схема превратится в коммутацию с промежуточным хранением. При другом типе маршрутизации, так называемой *wormhole routing* («червоточина»), если первый блок не может двигаться дальше, в исходный пункт передается сигнал остановить передачу, и пакет может оборваться, будучи растянутым на два и более коммутаторов. Когда необходимые ресурсы становятся доступными, пакет может двигаться дальше.

Оба подхода аналогичны конвейерному выполнению команд в центральном процессоре. В любой момент времени каждый коммутатор выполняет небольшую часть работы, и в результате получается более высокая производительность, чем если бы эту же работу выполнял один из коммутаторов.

7.4. Производительность параллельных систем

Цель создания компьютера параллельного действия — сделать так, чтобы он работал быстрее, чем однопроцессорная машина. Если эта цель не достигнута, то никакого смысла в построении компьютера параллельного действия нет. Более того, эта цель должна быть достигнута при наименьших затратах. Машина, которая работает в два раза быстрее, чем однопроцессорная, но стоит в 50 раз дороже последней, не будет пользоваться спросом.

7.4.1. Метрика аппаратного обеспечения

В аппаратном обеспечении наибольший интерес представляет скорость работы процессоров, устройств ввода-вывода и сети. Скорость работы процессоров и устройств ввода-вывода такая же, как и в однопроцессорной машине, поэтому ключевыми параметрами в параллельной системе являются те, которые связаны с межсоединением. Здесь есть два ключевых момента: время ожидания и пропускная способность.

Полное время ожидания — это время, которое требуется на то, чтобы процессор отправил пакет и получил ответ. Если пакет посылается в память, то время ожидания — это время, которое требуется на чтение и запись слова или блока слов.

Если пакет посылается другому процессору, то время ожидания — это время, которое требуется на межпроцессорную связь для пакетов данного

размера. Обычно интерес представляет время ожидания для пакетов минимального размера (как правило, для одного слова или небольшой строки кэш-памяти).

Время ожидания зависит от нескольких факторов. Для сетей с коммутацией каналов, сетей с промежуточным хранением и сетей без буферизации пакетов характерно разное время ожидания. Для коммутации каналов время ожидания составляет сумму времени установки и времени передачи. Для установки схемы нужно выслать пробный пакет, чтобы зарезервировать необходимые ресурсы, а затем передать назад сообщение об этом. После этого можно ассемблировать пакет данных. Когда пакет готов, биты можно передавать на полной скорости, поэтому если общее время установки составляет T_S, размер пакета равен p битам, а пропускная способность b битов в секунду, то время ожидания в одну сторону составит $T_S + p/b$ с.

Если схема дуплексная и никакого времени установки на ответ не требуется, то минимальное время ожидания для передачи пакета размером в p битов и получения ответа размером в p битов составляет $T_S + 2p/b$ с.

При пакетной коммутации не нужно посылать пробный пакет в пункт назначения заранее, но все равно требуется некоторое время установки T_S на компоновку пакета. Здесь время передачи в одну сторону составляет $T_S + p/b$, но за этот период пакет доходит только до первого коммутатора. При прохождении через сам коммутатор получается некоторая задержка T_D, а затем происходит переход к следующему коммутатору и т. д. Время T_D состоит из времени обработки и задержки в очереди (когда нужно ждать, пока не освободится выходной порт). Если имеется n коммутаторов, то общее время ожидания в одну сторону составляет $T_S + n(p/b + T_D) + p/b$, где последнее слагаемое отражает копирование пакета из последнего коммутатора в пункт назначения.

Время ожидания в одну сторону для коммутации без буферизации пакетов и «червоточины» в лучшем случае будет приближаться к $T_S + p/b$, поскольку здесь нет пробных пакетов для установки схемы и нет задержки, обусловленной промежуточным хранением. По существу, это время начальной установки для компоновки пакета плюс время на передачу битов. Следовало бы еще прибавить задержку на распространение сигнала, но она обычно незначительна.

Следующая характеристика аппаратного обеспечения — пропускная способность. Многие программы параллельной обработки, особенно в есте-

ственных науках, перемещают огромные массивы данных, поэтому количество байтов, которое система способна перемещать в секунду, имеет очень большое значение для производительности. Существует несколько показателей пропускной способности.

Другой показатель — суммарная пропускная способность — вычисляется путем суммирования пропускной способности всех каналов связи. Это число показывает максимальное число битов, которое можно передать сразу. Еще один важный показатель — средняя пропускная способность каждого процессора. Если каждый процессор способен выдавать только 1 Мбайт/с, то от сети с пропускной способностью между секциями в 100 Гбайт/с не будет толка. Скорость взаимодействия будет ограничена тем, сколько данных может выдавать каждый процессор.

На практике приблизиться к теоретически возможной пропускной способности очень трудно. Пропускная способность сокращается по многим причинам. Например, каждый пакет всегда содержит какие-то служебные сигналы и данные, это компоновка, построение заголовка, отправка. При отправке 1024 пакетов по 4 байта каждый мы никогда не достигнем той же пропускной способности, что и при отправке одного пакета на 4096 байтов. К сожалению, для достижения маленького времени ожидания лучше использовать маленькие пакеты, поскольку большие надолго блокируют линии и коммутаторы. В результате возникает конфликт между достижением низкого времени ожидания и высокой пропускной способности. Для одних прикладных задач первое важнее, чем второе, для других — наоборот. Важно знать, что всегда можно купить более высокую пропускную способность (добавив больше проводов или поставив более широкие провода), но нельзя купить низкое время ожидания. Поэтому лучше сначала сделать время ожидания как можно меньшим, а уже потом заботиться о пропускной способности.

7.4.2. Метрика программного обеспечения

Метрика аппаратного обеспечения показывает, на что способно аппаратное обеспечение. Но пользователей интересует совсем другое. Они хотят знать, насколько быстрее будут работать их программы на компьютере параллельного действия по сравнению с однопроцессорным компьютером. Для них ключевым показателем является коэффициент ускорения:

насколько быстрее работает программа в *n*-процессорной системе по сравнению с 1-процессорной системой. Результаты обычно иллюстрируются графиком (рис. 7.7). Здесь представлены несколько разных параллельных программ, которые работают на мультикомпьютере, состоящем из 64 процессоров Pentium Pro. Каждая кривая показывает повышение скорости работы одной программы с *k* процессорами как функцию от *k*. Идеальное повышение скорости показано сплошной линией, где использование *k* процессоров заставляет программу работать в *k* раз быстрее для любого *k*. Лишь немногие программы достигают совершенного повышения скорости, но есть достаточное число программ, которые приближаются к идеалу. Скорость работы *N*-объектной задачи с добавлением новых процессоров увеличивается стремительно; «авари» (африканская игра) ускоряется вполне сносно; но инвертирование матрицы нельзя ускорить более чем в пять раз, сколько бы процессоров мы ни использовали.

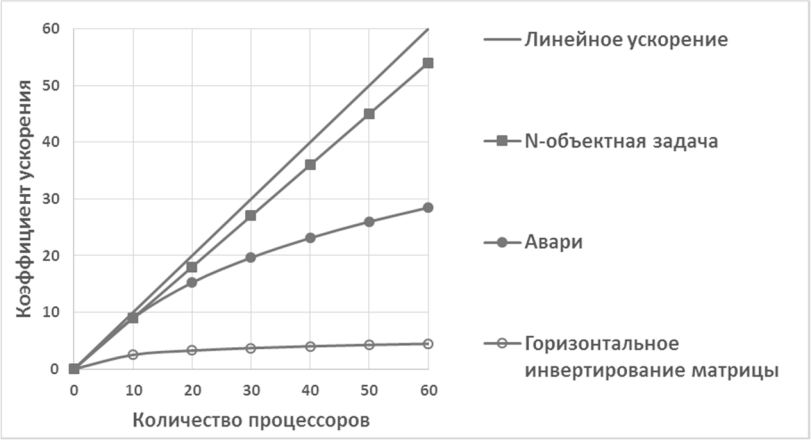


Рис. 7.7. График ускорения. Идеальный коэффициент ускорения показан непрерывной линией

Есть ряд причин, по которым практически невозможно достичь идеального повышения скорости: все программы содержат последовательную часть, они часто имеют фазу инициализации, они обычно должны считывать данные и собирать результаты. Большое количество процессоров здесь не поможет. Предположим, что на однопроцессорном компьютере программа работает *T* с, причем доля (*f*) от этого времени является последовательным кодом, а доля (1 – *f*) потенциально параллелизуется, как показано

на рисунке 7.8*а*. Если второй код можно запустить на n процессорах без издержек, то время выполнения программы в лучшем случае сократится с $(1 - f) \cdot T$ до $(1 - f) \cdot T/n$, как показано на рисунке 7.8*б*. В результате общее время выполнения программы (и последовательной, и параллельной частей) будет $f \cdot T + (1 - f) \cdot T/n$. Коэффициент ускорения — это время выполнения изначальной программы (T), разделенное на это новое время выполнения: $n/d + (n - 1) \cdot f$.

Для $f = 0$ мы можем получить линейное повышение скорости, но для $f > 0$ идеальное повышение скорости невозможно, поскольку в программе имеется последовательная часть. Это явление носит название закона Амдала.

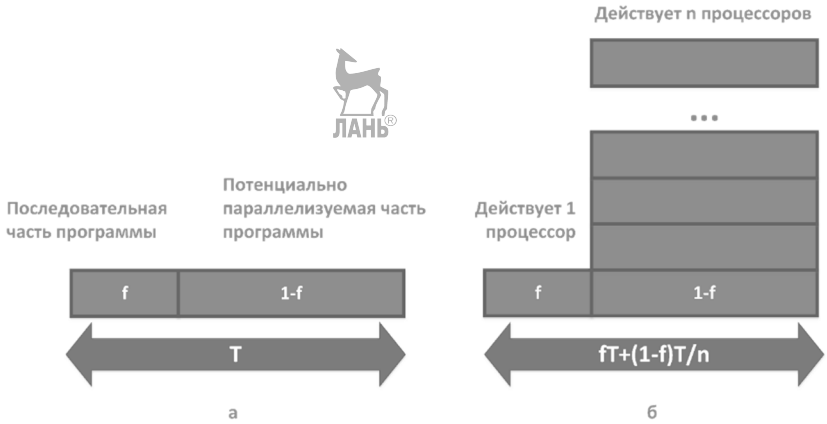


Рис. 7.8. Программа содежит последовательную часть и параллелизуемую часть (*а*); результат параллельной обработки части программы (*б*)

Закон Амдала — это только одна причина, по которой невозможно идеальное повышение скорости. Определенную роль в этом играют и время ожидания в коммуникациях, и ограниченная пропускная способность, и недостатки алгоритмов.

Даже если бы мы имели в наличии 1000 процессоров, не все программы можно написать так, чтобы использовать такое большое число процессоров, а непроизводительные издержки для запуска их всех могут быть очень значительными. Кроме того, многие известные алгоритмы трудно подвергнуть параллельной обработке, поэтому в данном случае приходится использовать субоптимальный алгоритм. Для многих прикладных задач желательно

заставить программу работать в n раз быстрее, даже если для этого потребуется $2n$ процессоров. В конце концов процессоры не такие уж и дорогие.

7.5. Классификация компьютеров параллельного действия

Было предложено и построено множество различных видов параллельных компьютеров, поэтому хотелось бы узнать, можно ли их как-либо категоризировать. Хорошей классификации компьютеров параллельного действия до сих пор не существует. Чаще всего используется классификация Флинна (*M. Flynn*), но даже она является очень грубым приближением (табл. 7.1).

Таблица 7.1. Классификация Флинна компьютеров параллельного действия

Поток команд	Поток данных	Название	Пример
1	1	SISD	Классическая машина фон Неймана
1	Много	SIMD	Векторный суперкомпьютер, массивно-параллельный процессор
Много	1	MISD	Не существует
Много	Много	MIMD	Мультипроцессор, мультикомпьютер

В основе классификации лежат два понятия: потоки команд и потоки данных. Поток команд соответствует счетчику команд. Система с n процессорами имеет n счетчиков команд и, следовательно, n потоков команд. Поток данных состоит из набора операндов. В примере с вычислением температуры, приведенном ранее, было несколько потоков данных, один для каждого датчика.

Потоки команд и данных в какой-то степени независимы, поэтому существует четыре комбинации.

1. SISD (*Single Instruction stream Single Data stream* — один поток команд, один поток данных) — это классический последовательный компьютер фон Неймана. Он содержит один поток команд и один поток данных и может выполнять только одно действие одномоментно.

2. Машины SIMD (*Single Instruction stream Multiple Data stream* — один поток команд, несколько потоков данных) содержат один блок управления,

выдающий по одной команде, но при этом есть несколько АЛУ, которые могут обрабатывать несколько наборов данных одновременно. ILLIAC IV — прототип машин SIMD. Существуют и современные машины SIMD. Они применяются для научных вычислений.

3. Машины MISD (*Multiple Instruction stream Single Data stream* — несколько потоков команд, один поток данных) — несколько странная категория. Здесь несколько команд оперируют одним набором данных. Трудно сказать, существуют ли такие машины. Однако некоторые считают машинами MISD машины с конвейерами.

4. Последняя категория — машины MIMD (*Multiple Instruction stream Multiple Data stream* — несколько потоков команд, несколько потоков данных). Здесь несколько независимых процессоров работают как часть большой системы. В эту категорию попадает большинство параллельных процессоров. И мультипроцессоры, и мультикомпьютеры — это машины MIMD.

Э. Таненбаумом расширена классификация Флинна (рис. 7.9). Машины SIMD распались на две подгруппы. В первую подгруппу попадают многочисленные суперкомпьютеры и другие машины, которые оперируют векторами, выполняя одну и ту же операцию над каждым элементом вектора. Во вторую подгруппу попадают машины типа ILLIAC IV, в которых главный блок управления посылает команды нескольким независимым АЛУ.

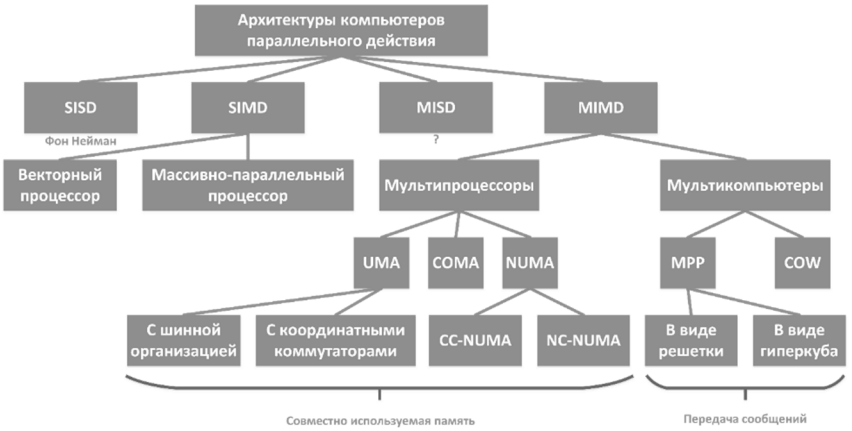


Рис. 7.9. Классификация Таненбаума компьютеров параллельного действия

В классификации Таненбаума категория MIMD распалась на мультипроцессоры (машины с памятью совместного использования) и мультикомпьютеры (машины с передачей сообщений). Существует три типа мультипроцессоров. Они отличаются друг от друга по способу реализации памяти совместного использования. Они называются UMA (*Uniform Memory Access* — архитектура с однородным доступом к памяти), NUMA (*NonUniform Memory Access* — архитектура с неоднородным доступом к памяти) и COMA (*Cache Only Memory Access* — архитектура с доступом только к кэш-памяти). В машинах UMA каждый процессор имеет одно и то же время доступа к любому модулю памяти. Иными словами, каждое слово памяти можно считать с той же скоростью, что и любое другое слово памяти.

Если это технически невозможно, самые быстрые обращения замедляются, чтобы соответствовать самым медленным, поэтому программисты не увидят никакой разницы. Это и значит «однородный». Такая однородность делает производительность предсказуемой, а этот фактор очень важен для написания эффективной программы.

Мультипроцессор NUMA, напротив, не обладает этим свойством. Обычно есть такой модуль памяти, который расположен близко к каждому процессору, и доступ к этому модулю памяти происходит гораздо быстрее, чем к другим. С точки зрения производительности очень важно, куда помещаются программа и данные.

Машины COMA — тоже с неоднородным доступом, но по другой причине. Подробнее каждый из этих трех типов мы рассмотрим, когда будем изучать соответствующие подкатегории.

Во вторую подкатегорию машин MIMD попадают мультикомпьютеры, которые, в отличие от мультипроцессоров, не имеют памяти совместного использования на архитектурном уровне. Другими словами, операционная система в процессоре мультикомпьютера не может получить доступ к памяти, относящейся к другому процессору, просто путем выполнения команды LOAD. Ему приходится отправлять сообщение и ждать ответа. Именно способность операционной системы считывать слово из отдаленного модуля памяти с помощью команды LOAD отличает мультипроцессоры от мультикомпьютеров. Даже в мультикомпьютере пользовательские программы могут обращаться к другим модулям памяти с помощью команд LOAD и STORE, но эту иллюзию создает операционная система, а не аппа-

ратное обеспечение. Разница незначительна, но очень важна. Так как мультикомпьютеры не имеют прямого доступа к отдаленным модулям памяти, они иногда называются машинами NORMA (*NO Remote Memory Access* — без доступа к отдаленным модулям памяти).

Мультикомпьютеры можно разделить на две категории. Первая категория содержит процессоры MPP (*Massively Parallel Processors* — процессоры с массовым параллелизмом) — дорогостоящие суперкомпьютеры, которые состоят из большого количества процессоров, связанных высокоскоростной коммуникационной сетью. В качестве примеров можно назвать Cray T3E и IBM SP/2.

Вторая категория мультикомпьютеров включает рабочие станции, которые связываются с помощью уже имеющейся технологии соединения. Эти примитивные машины называются NOW (*Network of Workstations* — сеть рабочих станций) и COW (*Cluster of Workstattions* — кластер рабочих станций).

ЗАКЛЮЧЕНИЕ

В заключение резюмируем содержание пособия. Компьютерные системы состоят из трех типов компонентов: процессоров, памяти и устройств ввода-вывода. Задача процессора заключается в том, чтобы последовательно вызывать команды из памяти, декодировать и выполнять их. Цикл «вызов → декодирование → выполнение» всегда можно представить в виде алгоритма. Вызов, декодирование и выполнение команд определенной программы иногда выполняются программой-интерпретатором, работающей на более низком уровне.

Для повышения скорости работы во многих компьютерах имеется один или несколько конвейеров, или суперскалярная архитектура с несколькими функциональными блоками, которые действуют параллельно. Широко распространены системы с несколькими процессорами. Компьютеры с параллельной обработкой включают: векторные процессоры, в которых одна и та же операция выполняется одновременно над разными наборами данных; мультипроцессоры, в которых несколько процессоров разделяют общую память; и мультикомпьютеры, в которых у каждого компьютера есть своя собственная память, но при этом компьютеры связаны между собой и пересылают друг другу сообщения.

Память можно разделить на основную и вспомогательную. Основная память используется для хранения программ, которые выполняются в данный момент. Время доступа невелико (максимум несколько десятков наносекунд) и не зависит от адреса, к которому происходит обращение. Кэш-память еще больше сокращает время доступа. Память может быть оснащена кодом с исправлением ошибок для повышения надежности.

Время доступа к вспомогательной памяти, напротив, гораздо больше (от нескольких миллисекунд и более) и зависит от расположения считываемых и записываемых данных. Наиболее распространенные виды вспомогательной памяти — магнитные ленты и диски (IDE, SCSI, SATA), оптические диски (CD, DVD, HD-DVD, BD) и твердотельные накопители (SSD).

Устройства ввода-вывода используются для передачи информации в компьютер и из компьютера. Они связаны с процессором и памятью одной или несколькими шинами. В качестве примеров можно назвать терминалы, мыши, принтеры и модемы. Большинство устройств ввода-вывода используют коды ASCII и UNICODE.

Компьютеры конструируются из интегральных схем, содержащих крошечные переключатели, которые называются вентилями. Обычно используются вентили «И» («AND»), «ИЛИ» («OR»), «НЕ-И» («NOT-AND», «NAND»), «НЕ-ИЛИ» («NOT-OR», «NOR») и «НЕ» («NOT»). Комбинируя отдельные вентили, можно построить простые схемы.

Более сложными схемами являются мультиплексоры, демультиплексоры, кодеры, декодеры, схемы сдвига и АЛУ. С помощью программируемой логической матрицы можно запрограммировать произвольные булевы функции. Если требуется много булевых функций, программируемые логические матрицы обычно более эффективны, чем другие средства. Законы булевой алгебры используются для преобразования схем из одной формы в другую. Во многих случаях таким способом можно произвести более экономичные схемы.

Арифметические действия в компьютерах осуществляются сумматорами. Одноразрядный полный сумматор можно сконструировать из двух полусумматоров. Чтобы построить сумматор для многоразрядных слов, полные сумматоры нужно соединить таким образом, чтобы выход переноса каждого сумматора передавался его левому соседу.

Статическая память состоит из защелок и триггеров, каждый из которых может хранить 1 бит информации. Их можно объединять и получать восьмиразрядные триггеры и защелки либо законченную память для хранения слов. Существуют различные типы памяти: ОЗУ, ПЗУ, программируемое ПЗУ, стираемое ПЗУ, электронно-перепрограммируемое ПЗУ и флеш-память. Статическое ОЗУ не нужно обновлять: оно хранит информацию, пока включен компьютер. Динамическое ОЗУ, напротив, нужно периодически обновлять, чтобы предотвратить утечку информации.

Компоненты компьютерной системы соединяются шинами. Большинство выводов обычного центрального процессора (хотя не все) запускают одну линию шины. Линии шины можно подразделить на адресные, информационные и линии управления. Синхронные шины запускаются задающим генератором. В асинхронных шинах для согласования работы задающего и подчиненного устройств используется система полного квитирования.

Переключатели, индикаторы, принтеры и многие другие устройства ввода-вывода можно связать с компьютером, используя микросхемы ввода-вывода (например, 8255A). Эти микросхемы по желанию можно сделать частью пространства ввода-вывода или частью пространства памяти. Выбор

микросхемы может происходить с помощью полного или частичного декодирования адреса в зависимости от того, какие задачи выполняет компьютер.

Основным компонентом любого компьютера является тракт данных. Он содержит несколько регистров, две или три шины, один или несколько функциональных блоков, например АЛУ, и схему сдвига. Основной цикл состоит из вызова нескольких операндов из регистров и их передачи по шинам к АЛУ и другому функциональному блоку. После выполнения операции результаты сохраняются опять в регистрах.

Тракт данных может управляться задатчиком последовательности, который вызывает микрокоманды из управляющей памяти. Каждая микрокоманда содержит биты, управляющие трактом данных в течение одного цикла. Эти биты определяют, какие операнды нужно выбирать, какую операцию нужно выполнять и что нужно делать с результатами. Кроме того, каждая микрокоманда определяет своего последователя (обычно в ней содержится адрес следующей микрокоманды). Некоторые микрокоманды изменяют этот базовый адрес с помощью операции «ИЛИ».

IJVM — это машина со стековой организацией и с 1-байтными кодами операций, которые помещают слова в стек, выталкивают слова из стека и выполняют различные операции над словами из стека (например, складывают их). Если добавить блок выборки команд для загрузки команд из потока байтов, можно устранить большое количество обращений к счетчику команд, и тогда скорость работы машины сильно повысится. Существует множество способов разработки микроархитектурного уровня. Есть много различных вариантов: двухшинная — трехшинная архитектура, кодированные — декодированные поля микрокоманды, наличие или отсутствие вызова с упреждением и многие другие.

Производительность компьютера можно повысить несколькими способами. Главный способ — использование кэш-памяти. Кэш-память прямого отображения и ассоциативная кэш-память с множественным доступом широко используются для того, чтобы ускорить обращения к памяти. Кроме того, применяется прогнозирование ветвления (как статическое, так и динамическое), исполнение с изменением последовательности и спекулятивное выполнение команд.

Уровень архитектуры команд — это «машинный язык». На этом уровне машина имеет память с байтовой организацией или с пословной организацией, состоящую из нескольких десятков мегабайтов и содержащую такие команды, как MOVE, ADD и т. д.

В большинстве современных компьютеров память организована в виде последовательности байтов, при этом 4 или 8 байтов группируются в слова. Обычно в машине имеется от 8 до 32 регистров, каждый из которых содержит одно слово.

Команды обычно имеют один, два или три операнда, обращение к которым происходит с помощью различных способов адресации: непосредственной, прямой, регистровой, индексной и т. д. Команды обычно могут перемещать данные, выполнять унарные и бинарные операции (в том числе арифметические и логические), совершать переходы, вызывать процедуры, осуществлять циклы, а иногда и выполнять некоторые операции ввода-вывода. Типичные команды перемещают слово из памяти в регистр или наоборот, складывают, вычитают, умножают или делят два регистра или регистр и слово из памяти или сравнивают два значения в регистрах или памяти. Обычно в компьютере содержится не более 200 команд.

Для осуществления передачи управления на втором уровне используется ряд элементарных действий: переходы, вызовы процедур, вызовы сопрограмм, ловушки и прерывания. Переходы нужны для того, чтобы остановить одну последовательность команд и начать новую. Процедуры нужны для того, чтобы изолировать какой-то блок программы, который можно вызывать из различных мест этой же программы. Сопрограммы позволяют двум потокам управления работать одновременно. Ловушки используются для сообщения об исключительных ситуациях (например, о переполнении). Прерывания позволяют осуществлять процесс ввода-вывода параллельно с основным вычислением, при этом центральный процессор получает сигнал, как только ввод-вывод завершен.

Операционную систему можно считать интерпретатором определенных особенностей архитектуры, которых нет на уровне команд. Главными среди них являются виртуальная память, виртуальные команды ввода-вывода и средства параллельной обработки.

Виртуальная память нужна для того, чтобы позволить программам использовать больше адресного пространства, чем есть у машины на самом деле, или чтобы обеспечить удобный механизм для защиты и разделения

памяти. Виртуальную память можно реализовать в виде чистого разбиения на страницы, чистой сегментации или того и другого вместе. При страничной организации памяти адресное пространство разбивается на равные по размеру виртуальные страницы. Одни из них отображаются на физические страничные кадры, другие — нет. Отсылка к отображенной странице преобразуется контроллером управления памятью в правильный физический адрес. Обращение к неотображенной странице вызывает ошибку из-за отсутствия страницы. Процессоры имеют сложные контроллеры управления памятью, которые поддерживают виртуальную память и страничную организацию.

Самой важной абстракцией ввода-вывода на этом уровне является файл. Файл состоит из последовательности байтов или логических записей, которые можно читать и записывать, не зная при этом о том, как работают диски и другие устройства ввода-вывода. Доступ к файлам может осуществляться последовательно, непоследовательно по номеру записи и непоследовательно по ключу. Для группировки файлов используются директории. Файлы могут храниться в последовательных секторах, а могут быть разбросаны по диску. В последнем случае требуются специальные структуры данных для нахождения всех блоков файла. Чтобы следить за свободным пространством на диске, можно использовать список или битовое отображение.

В системах часто поддерживается параллельная обработка. Для этого путем разделения времени в одном процессоре моделируется несколько процессов.

Неконтролируемое взаимодействие различных процессов может привести к состоянию гонок. Чтобы избежать их, вводятся специальные средства синхронизации. Самыми простыми из них являются семафоры.

UNIX и NT являются сложными операционными системами. Обе системы поддерживают страничную организацию памяти и отображаемые в память файлы. Кроме того, они поддерживают иерархические системы файлов, в которых файлы состоят из последовательности байтов. Наконец, обе системы поддерживают процессы и потоки и обеспечивают механизмы их синхронизации.

Компьютеры параллельной обработки можно разделить на две основные категории: SIMD и MIMD. Машины SIMD выполняют одну команду одновременно над несколькими наборами данных. Это массивно-параллельные процессоры и векторные компьютеры. Машины MIMD выполняют

разные программы на разных машинах. Машины MIMD можно подразделить на мультипроцессоры, которые совместно используют общую основную память, и мультикомпьютеры, которые не используют общую основную память. Системы обоих типов состоят из процессоров и модулей памяти, связанных друг с другом различными высокоскоростными сетями, по которым между процессорами и модулями памяти передаются пакеты запросов и ответов. Применяются различные топологии, в том числе решетки, торы, кольца и гиперкубы.

Для всех мультипроцессоров ключевым вопросом является модель согласованности памяти. Из наиболее распространенных моделей можно назвать согласованность по последовательности, процессорную согласованность, слабую согласованность и свободную согласованность. Мультипроцессоры можно строить с использованием отслеживающей шины, например в соответствии с протоколом MESI. Кроме того, возможны различные сети, а также машины на основе каталога NUMAHCOMA.

Мультипроцессоры можно разделить на системы MPP и COW, хотя граница между ними произвольна. К системам MPP относятся Cray T3E и Intel/Sandia Option Red. В них используются запатентованные высокоскоростные сети межсоединений. Системы COW, напротив, строятся из таких стандартных деталей, как Ethernet, ATM и Myrinet.

Мультикомпьютеры часто программируются с использованием пакета с передачей сообщений (например, PVM или MPI). Оба пакета поддерживают библиотечные вызовы для отправки и получения сообщений. Оба работают поверх существующих операционных систем.

Альтернативный подход — использование памяти совместного использования на прикладном уровне (например, система DSM со страничной организацией, пространство кортежей в системе Linda, объекты в системах Orca и Globe). Система DSM моделирует совместно используемую память на уровне страниц, и в этом она сходна с машиной NUMA. Системы Linda, Orca и Globe создают иллюзию совместно используемой памяти с помощью кортежей, локальных объектов и глобальных объектов соответственно.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. *Баула*, *В. Г.* Введение в архитектуру ЭВМ и системы программирования. — М. : МГУ, 2003.
2. *Владимиров*, *Д. А.* Булевы алгебры. — М. : Наука, 1969. — 320 с.
3. *Глушков*, *В. М.* О развитии структур многопроцессорных вычислительных машин, интерпретирующих языки высокого уровня / В. М. Глушков, С. Б. Погребинский, З. Л. Рабинович // Управляющие системы и машины. — 1978. — № 6. — С. 61–66.
4. *Гуров*, *С. И.* Булевы алгебры, упорядоченные множества, решетки: определения, свойства, примеры. — М. : Либроком, 2013. — 352 с.
5. Основы информатики и вычислительной техники : проб. учеб. пособие : в 2 ч. / А. П. Ершов, В. М. Монахов, С. А. Бешенков [и др.]. — М. : Просвещение, 1985. — Ч. 1. — 96 с.
6. Основы информатики и вычислительной техники : проб. учеб. пособие : в 2 ч. / А. П. Ершов, В. М. Монахов, С. А. Бешенков [и др.] — М. : Просвещение, 1985. — Ч. 2. — С. 143.
7. *Иванов*, *Б. Н.* Дискретная математика. Алгоритмы и программы: расширенный курс. — М. : Известия, 2011. — 512 с.
8. *Каган*, *Б. М.* Электронные вычислительные машины и системы. — М. : Энергоатомиздат, 1991. — 592 с.
9. *Кузнецов*, *О. П.* Дискретная математика для инженера / О. П. Кузнецов, Г. М. Адельсон-Вельский. — М. : Энергоатомиздат, 1988. — 480 с.
10. Организация вычислений в многопроцессорных вычислительных системах / В. С. Михалевич, Ю. В. Капитонова, А. А. Летичевский [и др.] // Кибернетика. — 1984. — № 3. — С. 1–10.
11. *Олифер*, *В. Г.* Компьютерные сети: принципы, технологии, протоколы : учебник для вузов / В. Г. Олифер, Н. А. Олифер. — СПб. : Питер, 2006. — 958 с.
12. *Олифер*, *В. Г.* Сетевые операционные системы : учебник / В. Г. Олифер, Н. А. Олифер. — СПб. : Питер, 2009. — 669 с.
13. *Рабаи*, *Ж. М.* Цифровые интегральные схемы. Методология проектирования / Ж. М. Рабаи, А. Чандракасан, Б. Николич — М. : Вильямс, 2007. — 912 с.

14. Прикладная теория цифровых автоматов / К. Г. Самофалов, А. М. Романкевич, В. Н. Валуйский [и др.]. — Киев : Вища школа, 1987. — 375 с.
15. *Таненбаум*, *Э.* Архитектура компьютера. — СПб. : Питер, 2007. — 844 с.
16. *Угрюмов*, *Е. П.* Цифровая схемотехника : учеб. пособие. — СПб. : БХВ-Петербург, 2004.
17. *Угрюмов*, *Е. П.* Элементы и узлы ЭЦВМ. — М. : Высшая школа, 1976. — 232 с.
18. *Хамахер*, *К.* Организация ЭВМ / К. Хамахер, З. Вранешич, С. Заки. — СПб. : Питер, 2003. — 848 с.

Антон Евгеньевич ЖУРАВЛЕВ

ОРГАНИЗАЦИЯ И АРХИТЕКТУРА ЭВМ
ВЫЧИСЛИТЕЛЬНЫЕ СИСТЕМЫ

Учебное пособие

Издание второе, стереотипное

Зав. редакцией
литературы по информационным технологиям
и системам связи *О. Е. Гайнутдинова*

ЛР № 065466 от 21.10.97
Гигиенический сертификат 78.01.10.953.П.1028
от 14.04.2016 г., выдан ЦГСЭН в СПб

Издательство «ЛАНЬ»
lan@lanbook.ru; www.lanbook.com;
196105, Санкт-Петербург, пр. Юрия Гагарина, д. 1, лит. А
Тел.: (812) 412-92-72, 336-25-09.
Бесплатный звонок по России: 8-800-700-40-71

Подписано в печать 23.07.21.
Бумага офсетная. Гарнитура Школьная. Формат 60×90 $^{1}/_{16}$.
Печать офсетная. Усл. п. л. 9,00. Тираж 30 экз.

Заказ № 910-21.

Отпечатано в полном соответствии
с качеством предоставленного оригинал-макета
в АО «Т8 Издательские Технологии».
109316, г. Москва, Волгоградский пр., д. 42, к. 5.