КЛАССИКА COMPUTER SCIENCE

АРХИТЕКТУРА КОМПЬЮТЕРА

И ПРОЕКТИРОВАНИЕ КОМПЬЮТЕРНЫХ СИСТЕМ

ЧЕТВЕРТОЕ ИЗДАНИЕ

Д. ПАТТЕРСОН
ДЖ. ХЕННЕССИ

ПИТЕР

СЕРИЯ

КЛАССИКА COMPUTER SCIENCE

David A. Patterson
John L. Hennessy

COMPUTER ORGANIZATION AND DESIGN

4th Edition

ELSEVIER

КЛАССИКА COMPUTER SCIENCE

Д. Паттерсон, Дж. Хеннесси

АРХИТЕКТУРА КОМПЬЮТЕРА

И ПРОЕКТИРОВАНИЕ КОМПЬЮТЕРНЫХ СИСТЕМ

ЧЕТВЕРТОЕ ИЗДАНИЕ

ПИТЕР®

Москва · Санкт-Петербург · Нижний Новгород · Воронеж
Ростов-на-Дону · Екатеринбург · Самара · Новосибирск
Киев · Харьков · Минск
2012

ББК 32.973.2-02
УДК 004.3
П20

Паттерсон Д., Хеннесси Дж.
П20 Архитектура компьютера и проектирование компьютерных систем. Классика Computers Science. 4-е изд. — СПб.: Питер, 2012. — 784 с.: ил.

ISBN 978-5-459-00291-1

Книга, выходящая уже в 4-м издании, посвящена структурной организации компьютера и отражает революционные изменения, происходящие в области аппаратного обеспечения, в частности стремительный переход от однопроцессорных систем к многоядерным микропроцессорам. В издании подробно описывается архитектура компьютера и устройство всех его компонентов: процессора, банков памяти, средств ввода-вывода и хранения данных. Отличительной особенностью книги является демонстрация взаимодействий между аппаратными средствами и системным программным обеспечением. Особое внимание уделяется многоядерным вычислительным системам и параллельному программированию. Многочисленные упражнения и задачи, приводимые после каждой темы, помогают закрепить материал. Книга рассчитана на широкий круг читателей: от студентов, изучающих компьютерные технологии, до опытных разработчиков, которые хотят освоить современные концепции многопроцессорного программирования.

ББК 32.973.2-02
УДК 004.3

Права на издание получены по соглашению с Elsevier Inc. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 978-0123744937 англ.
ISBN 978-5-459-00291-1

© 2009 by Elsevier Inc
© Перевод на русский язык ООО Издательство «Питер», 2012
© Издание на русском языке, оформление ООО Издательство «Питер», 2012

Краткое оглавление

Предисловие 9

Глава 1. Компьютерные абстракции и технологии 18

Глава 2. Инструкции: язык компьютера 97

Глава 3. Арифметика для компьютеров 245

Глава 4. Процессор 332

Глава 5. Объемная и быстродействующая: анализ иерархии памяти 498

Глава 6. Хранение информации и другие темы, связанные с вводом-выводом 629

Глава 7. Многоядерность, мультипроцессорные системы и кластеры ... 697

Оглавление

Предисловие **9**

Глава 1. Компьютерные абстракции и технологии **18**

1.1. Введение 18
1.2. Что находится ниже вашей программы 26
1.3. Что скрывается под крышкой корпуса компьютера 30
1.4. Производительность 47
1.5. Барьер потребляемой мощности 60
1.6. Коренное изменение: переход от одного к нескольким процессорам 62
1.7. Реальное оборудование: производство и оценочное тестирование AMD Opteron X4 67
1.8. Заблуждения и недоразумения 74
1.9. Заключительные комментарии 79
1.10. Упражнения 81

Глава 2. Инструкции: язык компьютера **97**

2.1. Введение 97
2.2. Операции, осуществляемые компьютерным оборудованием 99
2.3. Операнды компьютерного оборудования 104
2.4. Числа со знаком и без знака 111
2.5. Представление инструкций в компьютере 118
2.6. Логические операции 125
2.7. Инструкции для принятия решения 128
2.8. Поддержка процедур в компьютерном оборудовании 134
2.9. Общение с людьми 144
2.10. Адресация MIPS для 32-разрядных непосредственных значений и адресов 150
2.11. Параллелизм и инструкции: синхронизация 159
2.12. Трансляция и запуск программы 162
2.13. Объединение всего ранее рассмотренного в упражнении по сортировке на языке C 173
2.14. Сравнение массивов и указателей 181
2.15. Реальное оборудование: инструкции ARM 185
2.16. Реальное оборудование: инструкции x86 190
2.17. Заблуждения и недоразумения 202
2.18. Заключительные комментарии 204

2.19. Упражнения ... 207
Ответы на вопросы для самопроверки ... 243

Глава 3. Арифметика для компьютеров ... 245

3.1. Введение ... 245
3.2. Сложение и вычитание ... 246
3.3. Умножение ... 252
3.4. Деление ... 258
3.5. Числа с плавающей точкой ... 269
3.6. Параллелизм и компьютерная арифметика: ассоциативность ... 297
3.7. Реальное оборудование: вычисления чисел с плавающей точкой в x86 299
3.8. Заблуждения и недоразумения ... 303
3.9. Заключительные комментарии ... 307
3.10. Упражнения ... 316
Ответы на вопросы для самопроверки ... 331

Глава 4. Процессор ... 332

4.1. Введение ... 332
4.2. Соглашения по логическому проектированию ... 337
4.3. Создание операционного блока ... 341
4.4. Простая схема реализации ... 350
4.5. Обзор конвейеризации ... 365
4.6. Операционный блок и система управления, оснащенные конвейером ... 381
4.7. Конфликты данных: сравнение препровождения данных и задержки ... 403
4.8. Конфликты управления ... 416
4.9. Исключения ... 427
4.10. Параллелизм и расширенный параллелизм на уровне инструкций ... 435
4.11. Реальное оборудование: конвейер AMD Opteron X4 (Barcelona) ... 451
4.12. Заблуждения и недоразумения ... 454
4.13. Заключительные комментарии ... 455
4.14. Упражнения ... 456
Ответы на вопросы для самопроверки ... 497

Глава 5. Объемная и быстродействующая: анализ иерархии памяти ... 498

5.1. Введение ... 499
5.2. Основы кэш-памяти ... 505
5.3. Измерение и повышение производительности кэш-памяти ... 525
5.4. Виртуальная память ... 543
5.5. Общая среда для иерархий памяти ... 573
5.6. Виртуальные машины ... 582
5.7. Использование конечного автомата для управления простой кэш-памятью ... 587
5.8. Параллелизм и иерархии памяти: целостность данных в кэш-памяти ... 592
5.9. Реальное оборудование: иерархии памяти AMD Opteron X4 (Barcelona) и Intel Nehalem ... 597
5.10. Заблуждения и недоразумения ... 603
5.11. Заключительные комментарии ... 607
5.12. Упражнения ... 609
Ответы на вопросы для самопроверки ... 628

Глава 6. Хранение информации и другие темы, связанные с вводом-выводом ... 629

6.1. Введение ... 630
6.2. Безотказность, надежность и готовность ... 633
6.3. Дисковое запоминающее устройство ... 636
6.4. Флэш-накопители ... 642
6.5. Соединение процессоров, памяти и устройств ввода-вывода ... 644
6.6. Организация интерфейса устройств ввода-вывода с процессором, памятью и операционной системой ... 650
6.7. Оценки производительности ввода-вывода: примеры, связанные с дисками и файловыми системами ... 660
6.8. Разработка систем ввода-вывода ... 663
6.9. Параллелизм и ввод-вывод: избыточные массивы недорогих дисков ... 664
6.10. Реальное оборудование: сервер Sun Fire x4150 ... 672
6.11. Заблуждения и недоразумения ... 679
6.12. Заключительные комментарии ... 685
6.13. Упражнения ... 686
Ответы на вопросы для самопроверки ... 696

Глава 7. Многоядерность, мультипроцессорные системы и кластеры ... 697

Мультипроцессор или кластерная организация ... 697
7.1. Введение ... 698
7.2. Сложности создания программ, выполняемых в параллельном режиме ... 701
7.3. Мультипроцессоры с общей памятью ... 705
7.4. Кластеры и другие мультипроцессоры с передачей сообщений ... 708
7.5. Аппаратная многопоточность ... 713
7.6. SISD, MIMD, SIMD, SPMD и использование векторов ... 717
7.7. Введение в графические процессоры ... 723
7.8. Введение в топологию мультипроцессорных сетей ... 731
7.9. Контрольные задачи для мультипроцессоров ... 735
7.10. Roofline: Простая модель производительности ... 738
7.11. Реальное оборудование: выполнение контрольных задач для четырех многоядерных процессоров с использованием модели Roofline ... 748
7.12. Заблуждения и недоразумения ... 758
7.13. Заключительные комментарии ... 760
7.14. Упражнения ... 763
Ответы на вопросы для самопроверки ... 777

Предисловие

> Самое прекрасное, что мы можем испытать, — это непостижимость. Она является источником всего настоящего искусства и науки.
>
> *Альберт Эйнштейн. «Во что я верю»*

Об этой книге

Мы уверены, что обучение в сфере науки и техники компьютерной обработки данных должно отражать текущее состояние дел и давать введение в принципы, формирующие информационные технологии. Мы также полагаем, что читатели в любой специализированной области компьютерной обработки данных должны понимать организационные парадигмы, определяющие возможности, производительность и, в конечном счете, успех компьютерных систем.

Современная компьютерная технология требует от профессионалов любой компьютерной специальности умения разбираться как в аппаратном, так и в программном обеспечении. Взаимодействие аппаратного и программного обеспечения на множестве уровней также предлагает структуру для понимания фундаментальных основ вычислительной техники. Независимо от того, чем вы интересуетесь: оборудованием или программным обеспечением, информатикой или электроникой, — в компьютерной организации и конструировании используются одни и те же основные идеи. Поэтому в данной книге акценты расставлены на демонстрации взаимоотношений между аппаратным и программным обеспечением и на концентрации вашего внимания на концепциях, положенных в основу современных компьютеров.

Недавний скачок от однопроцессорных систем к многоядерным микропроцессорам подтверждает разумность такого подхода, выбранного еще в первом издании. Программисты, конечно, могли бы проигнорировать тот или иной совет и всецело положиться на разработчиков компьютеров, но для создателей компиляторов и проектировщиков микросхем эти времена закончились. Чтобы программы работали быстрее, они должны использовать параллельную обработку. Хотя целью позволить программистам работать, даже ничего не подозревая о параллельной природе исходного оборудования, задались многие исследователи, на реализацию этой мечты уйдут долгие годы. Мы считаем, что по крайней мере в следующем десятилетии большинству программистов придется разбираться в интерфейсе аппаратного и программного обеспечения, если они хотят добиться эффективного выполнения программ на параллельных компьютерах.

Аудитория читателей этой книги включает тех, кто имеет небольшой опыт в программировании на ассемблере или в разработке алгоритмов и нуждается в понимании основ компьютерной организации, а также тех, кто уже имеет определенную подготовку в программировании на ассемблере и в разработке алгоритмов и собирается изучить вопросы конструирования компьютера или разобраться в том, как работает система и почему она работает именно таким образом.

О других книгах

Некоторые читатели уже знакомы с книгой «Computer Architecture: A Quantitative Approach», которую часто называют «Хеннесси и Паттерсон». (Эту же книгу, в свою очередь, часто называют «Паттерсон и Хеннесси».) Причиной, побудившей нас написать предыдущую книгу, было желание дать описание принципов компьютерной архитектуры с использованием незыблемых основных технических принципов и количественных соотношений стоимости—производительности. Чтобы создать атмосферу реальных конструкций, мы использовали подход, сочетающий примеры и оценки, основанные на изучении коммерческих систем. Наша цель заключалась в демонстрации того, что компьютерную архитектуру можно изучить, используя не описательный подход, а количественные методологии. Книга была предназначена для настоящего компьютерного профессионала, желающего получить детальное представление о компьютерах.

Большинство читателей этой книги не собирались становиться разработчиками компьютеров. И тем не менее производительность и снижение энергопотребления будущих программных систем будут сильно зависеть от того, насколько хорошо разработчики программного обеспечения понимают основные технологии работы компьютерного оборудования в той или иной системе. Поэтому твердое знание основ, представляемых в данной книге, будет необходимо создателям компиляторов, разработчикам операционных систем, программистам баз данных и большинству других разработчиков программного обеспечения. Аналогичным образом и конструкторам оборудования следует четко понимать влияние результатов их работы на работу прикладных программ.

Мы понимали, что эта книга должна была представлять собой нечто большее, чем подгруппа материалов из «Computer Architecture», и весь материал книги тщательно пересмотрен и соответствует интересам разнообразной аудитории читателей. Мы настолько были довольны результатами своей работы, что пересмотрели последующие издания «Computer Architecture», чтобы удалить из них основную часть вводного материала.

Изменения, внесенные в четвертое издание

Выпуская четвертое издание этой книги, мы ставили перед собой пять главных задач: в связи с происходящей в микропроцессорах многоядерной революцией выде-

лить во всей книге темы параллельного аппаратного и программного обеспечения; сократить существующий материал, чтобы оставить место для тем, касающихся параллелизма; улучшить в целом педагогический аспект; обновить техническое содержимое, чтобы оно отражало изменения, происшедшие в промышленности со дня публикации в 2004 году третьего издания; и привлечь внимание к практическим заданиям в наш век развития Интернета.

Перед более подробным рассмотрением целей данной книги давайте посмотрим на приведенную на следующей странице таблицу. В ней показаны пути изучения оборудования и программного обеспечения по всему материалу книги. Главы 1, 4, 5 и 7 фигурируют в обоих основных путях, независимо от начального опыта читателя или круга его интересов. Глава 1 представляет собой новое введение в тематику, включающее рассмотрение важности вопроса энергопотребления и того, как им мотивируется переход от одноядерных к многоядерным микропроцессорам. Эта глава также включает материал, касающийся производительности и использования контрольных задач, который в третьем издании был представлен отдельной главой. Главу 2, скорее всего, можно считать обзорным материалом для тех, кого больше интересует оборудование, но она также является обязательной к прочтению теми, кого больше интересует программное обеспечение, особенно она полезна тем читателям, которые интересуются углубленным изучением компиляторов и объектно-ориентированных языков программирования. В эту главу включен материал из главы 3 третьего издания, поэтому вся архитектура MIPS, за исключением инструкций для работы с числами с плавающей точкой, теперь изложена в одной главе. Глава 3 предназначена для читателей, интересующихся конструированием операционного блока или получением дополнительных сведений об арифметике чисел с плавающей точкой. Возможно, кто-то пропустит чтение главы 3, поскольку она им не нужна или же потому, что она имеет обзорный характер. В главе 4 с целью объяснения, что такое конвейеризированные процессоры, объединены две главы третьего издания. В разделах 4.1, 4.5 и 4.10 приводится обзорная информация для тех, кто больше интересуется программным обеспечением. Тем же, кто больше интересуется оборудованием, эта глава может показаться слишком простой. Главу 6 важно прочитать тем, кого больше интересует программное обеспечение, а всем остальным ее нужно прочитать при наличии времени. Последняя глава, посвященная многоядерности, мультипроцессорам и кластерам, должна быть прочитана всеми.

Главной целью авторов было сделать параллелизм из второстепенного материала предыдущего издания первостепенным материалом издания настоящего. Наиболее ярким примером этого может послужить глава 7. В частности, в этой главе представлена модель производительности Roofline и показано ее значение для оценки четырех недавно выпущенных многоядерных архитектур при выполнении двух базовых компонентов набора контрольных задач. Эта модель способна продемонстрировать в самую суть многоядерных микропроцессоров, как и модель трех «С» применительно к устройствам кэш-памяти.

Глава	Разделы	Тем, кто больше интересуется	
		программным обеспечением	аппаратным обеспечением
1. Компьютерные абстракции и технологии	1.1–1.9	Читать внимательно	Читать внимательно
2. Инструкции: язык компьютера	2.1–2.18	Читать внимательно	Просмотреть или читать
3. Арифметика для компьютеров	3.1–3.9	Просмотреть или читать	Просмотреть или читать
4. Процессор	4.1 (Введение)	Читать внимательно	Читать внимательно
	4.2 (Соглашения по логическому проектированию)		Читать внимательно
	4.3–4.4 (Простая схема реализации)	Просмотреть или читать	Читать внимательно
	4.5 (Обзор конвейеризации)	Читать внимательно	Читать внимательно
	4.6 (Операционный блок, оснащенный конвейером и система управления)	Просмотреть или читать	Читать внимательно
	4.7–4.9 (Конфликты, исключения)		Читать внимательно
	4.10–4.13 (Параллелизм, реальное оборудование, заблуждения)	Читать внимательно	Читать внимательно
5. Объемная и быстродействующая: анализ иерархии памяти	5.1–5.11	Читать внимательно	Читать внимательно
6. Хранение информации и другие темы, связанные с вводом-выводом	6.1–6.13	Читать внимательно	Прочитать при наличии времени
7. Многоядерность, мультипроцессорные системы и кластеры	7.1–7.13	Читать внимательно	Читать внимательно

Учитывая важность параллелизма, было бы неразумно ждать рассмотрения этого вопроса вплоть до последней главы, поэтому посвященные ему разделы есть в каждой из предшествующих шести глав:

- Глава 1: Параллелизм и потребляемая мощность. Здесь показано, как тепловые барьеры, связанные с повышением потребляемой мощности заставили промышленность перейти на параллелизм и как он помогает решить эту проблему.
- Глава 2: Параллелизм и инструкции: синхронизация. В этой главе рассматриваются блокировки совместно используемых переменных, особенно MIPS-

инструкции связанной загрузки — Load Linked и условного сохранения — Store Conditional.

- Глава 3: Параллелизм и компьютерная арифметика: ассоциативность. В этой главе рассматриваются изменения точности представления чисел и вычисления с плавающей точкой.
- Глава 4: Параллелизм и расширенный параллелизм на уровне инструкций. Здесь рассматриваются расширенный параллелизм на уровне инструкций (ILP) — суперскаляры, предположения, слово инструкции очень большой длины (VLIW), развертывание цикла и выполнение с изменением последовательности, а также взаимосвязанность глубины конвейера и потребляемой мощности.
- Глава 5: Параллелизм и иерархии памяти: целостность данных в кэш-памяти. Здесь представлены понятия целостности, последовательности и протоколы отслеживания кэш-памяти.
- Глава 6: Параллелизм и ввод-вывод: избыточные массивы недорогих дисков. Здесь приводится описание RAID-массивов в качестве параллельных систем ввода-вывода, а также в качестве систем ввода-вывода высокой степени готовности.

Глава 7 заканчивается описанием причин, по которым эта атака на параллелизм должна стать успешнее предыдущих.

Вторая цель заключалась в сокращении материала, чтобы появилось место для нового материала по параллелизму. Первый шаг заключался в тщательной ревизии всех без исключения разделов, накопившихся за три издания, чтобы убедиться в их необходимости. Коренные изменения выразились в объединении глав и в исключении некоторых тем. Марк Хилл (Mark Hill) предложил исключить реализацию мультициклового процессора и вместо нее добавить в главу, посвященную иерархии памяти, мультицикловый контроллер кэш-памяти. Это позволило представить процессор в одной главе вместо двух, улучшив материал о процессоре за счет исключения ненужных тем. Материал о производительности из отдельной главы третьего издания был добавлен в материал главы 1.

Третья цель заключалась в усилении педагогического воздействия этой книги. Глава 1 стала более содержательной, включив в себя вопросы производительности, интегральных микросхем и проблем энергопотребления, ее материал подготавливает почву для всей остальной книги. Главы 2 и 3 изначально были написаны в эволюционной манере, начиная с «одноклеточной» архитектуры и заканчивая в завершении главы 3 полноценной MIPS-архитектурой. Такой неторопливый стиль не подходит современному читателю. В этом издании весь материал, касающийся набора целочисленных инструкций, собран в главе 2, превращая главу 3 в необязательную для многих читателей, и каждый раздел теперь содержит материал, не зависящий от материала других разделов. Читателю теперь не нужно читать все предыдущие разделы. Глава 2 больше, чем в предыдущих изданиях, походит на справочное руководство. Глава 4 теперь лучше справляется со своей задачей, поскольку весь материал о процессоре выделен в отдельную главу, а мультицикловая реализация теперь не отвлекает внимание от основной темы. В главе 5 появился новый раздел, посвященный созданию контроллеров кэш-памяти.

Книга посвящена быстро развивающейся области знаний, и, как всегда, в нашем новом издании важной целью было обновление технического содержимого. В качестве примера на протяжении всей книги фигурирует процессор AMD Opteron X4 модели 2356 (кодовое название «Barcelona»), упоминания о нем можно найти в главах 1, 4, 5 и 7. В главы 1 и 6 добавлены результаты, полученные с помощью нового мощного комплекта контрольных задач от корпорации SPEC. В главу 2 добавлен раздел, посвященный ARM-архитектуре, которая в настоящий момент является наиболее популярной в мире 32-разрядной архитектурой набора инструкций. В главу 5 добавлен новый раздел, посвященный виртуальным машинам, роль которых снова возрастает. В главе 5 приведены подробные измерения производительности кэш-памяти многоядерного процессора Opteron X4 и некоторые детали, касающиеся его конкурента, Intel Nehalem, о которых не было объявлено вплоть до выпуска этой книги. В главе 6 впервые дано описание флэш-памяти, а также удивительно компактного сервера от компании Sun, где в устройство размером 1U втиснуты 8 ядер, 16 DIMM-модулей и 8 дисков. Эта глава также включает последние результаты долговременных испытаний дисков на отказ. В главе 7 рассматривается множество тем, посвященных параллелизму, включая многопоточность, SIMD, вектор, графические процессорные устройства (GPU), модели производительности, контрольные задачи (benchmarks), мультипроцессорные сети, и, вдобавок к Opteron X4, дается описание трех многоядерных процессоров: Intel Xeon модели e5345 (Clovertown), IBM Cell модели QS20 и Sun Microsystems T2 модели 5120 (Niagara 2).

Еще одна задача — сделать практические задания в эпоху Интернета полезными для преподавателей, ведь домашние задания являются важным методом изучения материала. (Хотя ответы публикуются почти сразу же после выхода книги.) Во-первых, привлеченные специалисты поработали над созданием совершенно новых упражнений для каждой имеющейся в этой книге главы. Во-вторых, большинство упражнений имеют количественные описания, подкрепленные таблицами, предоставляющими несколько альтернативных количественных параметров, необходимых для ответа на вопрос. Огромное количество упражнений плюс гибкость, с которой преподаватель может выбирать варианты этих упражнений, усложнят студентам поиск готовых решений в Сети. Преподаватели также могут по своему желанию изменить эти количественные параметры, в очередной раз расстроив тех студентов, которые привыкли полагаться на Интернет в поисках решений не подвергающихся изменениям упражнений. Мы полагаем, что этот новый подход будет ценным дополнением к книге. Пожалуйста, сообщите, насколько хорошо он у вас сработает, — независимо от того, кто вы, — студент или преподаватель!

Мы сохранили ценные элементы книги из предыдущих изданий. Чтобы книга была полезнее в качестве справочника, мы, как и раньше, вынесли определения новых понятий на поля при первом же их упоминании. Раздел книги «Представление о производительности программ» помогает читателям понять, что такое производительность программ и как ее можно повысить, а раздел книги «Интерфейс аппаратного и программного обеспечения» — разобраться в сильных и слабых сторонах этого интерфейса. Раздел «Общее представление» оставлен для

того, чтобы читатель мог разглядеть лес, а не множество деревьев. Раздел «Самопроверка» помогает читателям убедиться в усвоении материала.

Поддержка преподавателей

Мы собрали большое количество вспомогательного материала, помогающего преподавателям вести курс с использованием данной книги. Решения упражнений, короткие тесты по главам, иллюстрации из книги, примечания и слайды для лекций и другие материалы можно получить у издателя. Более подробную информацию можно найти на веб-сайте textbooks.elsevier.com/9780123744937.

Заключительные комментарии

Если вы прочитаете раздел благодарностей, то увидите, что мы прошли длинный путь исправления ошибок. Поскольку книга не раз переиздавалась, у нас была возможность внести множество поправок.

Это издание отмечено перерывом в долговременном сотрудничестве между Хеннесси и Паттерсоном, начавшемся в 1989 году. К сожалению, должность президента в одном из самых больших в мире университетов не позволила Хеннесси принимать активное участие в работе над новым изданием. Оставшийся в одиночестве автор почувствовал себя жонглером, всегда выступавшим с партнером, которого внезапно вытолкнули на манеж для сольного выступления. Следовательно, люди, упомянутые в разделе благодарностей, и коллеги из Беркли сыграли еще более значительную роль в формировании содержимого данной книги. Тем не менее на сей раз ответственность за весь материал, который вы собираетесь читать, несет только один автор.

Благодарности участникам выпуска четвертого издания

Я хочу еще раз выразить свою признательность Джиму Ларусу (Jim Larus) из Microsoft Research за его готовность поделиться своим опытом в программировании на языке ассемблера, а также пригласить читателей этой книги воспользоваться симулятором, который он разработал и поддерживает.

Я также очень благодарен за вклад многим специалистам, разработавшим новые упражнения для этого нового издания. Составление хороших упражнений — дело непростое, и каждый автор упорно работал над созданием довольно сложных и привлекательных задач:

- Глава 1: Хавьер Бругейра (Javier Bruguera), Университет Сантьяго де Компостелле.
- Глава 2: Джон Оливер (John Oliver), Калифорнийский политехнический государственный университет, Сан-Луис-Обиспо, с добавлениями от Николь Кайян (Nicole Kaiyan, Университет Аделаиды) и Милоша Првуловича (Milos Prvulovic, Технологический институт Джорджии).

- Глава 3: Мэтью Фарренс (Matthew Farrens, Калифорнийский университет в Дэвисе).
- Глава 4: Милош Првулович (Milos Prvulovic, Технологический институт Джорджии).
- Глава 5: Цзичуань Чанг (Jichuan Chang), Джакоб Леверич (Jacob Leverich), Кевин Лим (Kevin Lim) и Партасарати Ранганатан (Parthasarathy Ranganathan), все из компании Hewlett-Packard, с добавлениями от Николь Кайян (Nicole Kaiyan, Университет Аделаиды).
- Глава 6: Перри Александер (Perry Alexander, Канзасский университет).
- Глава 7: Дэвид Каэли (David Kaeli, Северо-Восточный университет).

Весь груз по редактированию и оценке всех этих новых упражнений взвалил на себя Петер Ашенден (Peter Ashenden).

Спасибо Дэвиду Огесту (David August) и Пракашу Прабху (Prakash Prabhu) из Принстонского университета за их работу над контрольными вопросами к главам.

В подборе значительного количества технического материала, используемого для написания этой книги, мне помогли коллеги из Силиконовой долины:

- AMD занимался подробностями и показателями Opteron X4 (Barcelona): Уильям Брэнтли (William Brantley), Василиос Ляшковитис (Vasileios Liaskovitis), Чак Мур (Chuck Moore) и Брайан Уалдекер (Brian Waldecker);
- Intel — предвыпускной информации по Intel Nehalem: Фэй Бриггс (Faye Briggs);
- Micron – предысторией флэш-памяти в главе 6: Дин Клейн (Dean Klein);
- Sun Microsystems — инструкциями для контрольных задач SPEC2006 в главе 2, а также Sun Server x4150 в главе 6: Ян Фишер (Yan Fisher), Джон Фаулер (John Fowler), Дэррил Гоув (Darryl Gove), Пол Джойс (Paul Joyce), Шеник Мет (Shenik Mehta), Пьер Рейнс (Pierre Reynes), Димитрий Штюве (Dimitry Stuve), Дургам Вайя (Durgam Vahia) и Дэвид Вивер (David Weaver);
- U.C. Berkeley: Крст Асанович (Krste Asanovic) (он подбросил идею сравнения параллелизма на уровне программ и параллелизма на уровне оборудования в главе 7), Джеймс Деммел (James Demmel) и Вевел Кэхэн (Velvel Kahan) (они прокомментировали параллелизм и вычисления с плавающей точкой), Жангкси Тэн (Zhangxi Tan) (он разработал контроллер кэш-памяти и написал Verilog-описание для него в главе 5), Сэм Уильямс (Sam Williams) (он предложил roofline-модель и оценку показателей многоядерных процессоров в главе 7) и все остальные коллеги в Par Lab, давшие массу предложений и отзывов по вопросам параллелизма, встречающимся по всей книге.

Я признателен многим преподавателям, которые ответили на опросы издателей, провели обзор наших предложений и входили в группу опроса для проведения анализа и изучения реакции на наши планы относительно этого издания.

Особенно хочется поблагодарить специалистов из Беркли, которые дали отзыв по главе 7, по самым сложным для написания частям этого издания: Крста Асановича (Krste Asanovic), Кристофера Бэттена (Christopher Batten), Растислава Бодика (Rastilav Bodik), Брайана Катанзаро (Bryan Catanzaro), Джайка Чонга (Jike Chong), Коушика Дейта (Kaushik Data), Грэга Гиблинга (Greg Giebling), Аника Джейна, Дже Ли (Jae Lee), Василия Волкова и Сэмьюэла Уилльямса (Samuel Williams).

Более двух сотен человек помогали создать это четвертое издание, которое, я надеюсь, станет нашей лучшей из всех выпущенных до сих пор книг. Пользуйтесь!

Дэвид А. Паттерсон

От издательства

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

Мы будем рады узнать ваше мнение!

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

Глава 1

Компьютерные абстракции и технологии

> Цивилизация развивается за счет расширения количества важных операций, которые можно выполнять, не думая о них.
>
> *Алфред Норт Уайтхед. «Введение в математику»*

1.1. Введение

Добро пожаловать в эту книгу! Мы рады представившейся возможности поделиться с вами теми возвышенными чувствами, которые вызывают у нас мир компьютерных систем. В этом мире нет места скуке и застою и новые идеи не прячут под сукно, напротив! Компьютеры являются продуктом невероятно бурно развивающейся индустрии информационных технологий, все направления которой составляют почти 10% валового национального продукта США и все расчеты в которой отчасти строятся на ускоренном совершенствовании компьютерных технологий, обещанном законом Мура. Эта весьма необычная индустрия поглощает инновации на одном дыхании. За последние 25 лет было создано множество новых компьютеров, чье появление представлялось не иначе как революционными переворотами в компьютерной индустрии, и недолговечность этих революций обусловливалась лишь тем, что кто-то другой создавал еще более передовой компьютер.

Эта инновационная гонка, начало которой было положено созданием электронных вычислительных машин в конце 40-х годов прошлого века, привела к беспрецедентным успехам. Если бы, к примеру, транспортная индустрия развивалась такими же темпами, сегодня мы смогли бы добраться из Нью-Йорка до Лондона примерно за секунду, заплатив за это всего несколько центов. Представьте на мгновение, как подобное усовершенствование могло бы изменить жизнь человечества: можно было бы жить на Таити, работать в Сан-Франциско, а по вечерам выбираться в Москву, посмотреть балет в Большом театре, – вы только оцените последствия такого изменения!

Компьютеры привели цивилизацию к третьей, информационной революции, которая встала в один ряд с промышленной и сельскохозяйственной. Преумножение интеллектуальной мощи и расширение возможностей человечества в этой сфере естественным образом отразились на его повседневной жизни, изменив способы

получения новых знаний. Появилось новое направление научных исследований, объединившее усилия специалистов по вычислительной технике и теоретиков и практиков в проведении новых исследований на передовых рубежах астрономии, биологии, химии и физики, не говоря уже обо всех остальных науках.

Компьютерная революция продолжается. С каждым ее шагом достигнутая мощность компьютерных вычислений в очередной раз удесятеряет возможности вычислительных машин. Прикладные задачи, казавшиеся еще недавно экономически неосуществимыми, неожиданно превращаются в повседневную практику. В недавнем прошлом следующие области применения компьютерных технологий относились к разряду научной фантастики:

- *Компьютеры в автомобилях.* Компьютерные системы управления в автомобилях до начала 80-х годов прошлого века, пока не произошло существенного улучшения ситуации с ценами и производительностью микропроцессоров, считались бредовой идеей. Сегодня компьютерное управление способствует уменьшению вредных выбросов, увеличивает эффективность использования топлива путем управления двигателем и существенно повышает безопасность, предотвращая заносы и своевременно приводя в действие подушки безопасности, защищающие пассажиров в случае аварийного столкновения автомобиля с препятствием.
- *Сотовые телефоны.* Разве кто-нибудь раньше мог хотя бы мечтать о том, что развитие компьютерных систем приведет к появлению сотовых телефонов, позволяющих связаться людям, находящимся в разных уголках планеты?
- *Проект по изучению генома человека.* Стоимость компьютерного оборудования для построения карты и анализа цепочек молекулы человеческой ДНК составляет сотни миллионов долларов. Вряд ли кто-нибудь мог всерьез рассматривать этот проект, если бы компьютеры стоили в десятки или в сотни раз дороже, как это было еще 10–20 лет назад. Более того, стоимость продолжает падать, и у вас может появиться возможность обзавестись картой своего собственного генома, позволяющей конкретизировать адресность медицинской помощи.
- *Всемирная паутина (World Wide Web).* Еще не существовавшая на время первого издания данной книги Всемирная паутина изменила жизнь всего нашего общества. Для многих она просто заменила библиотеки.
- *Поисковые машины.* По мере того как Всемирная паутина становилась все более распространенной и всеобъемлющей, существенно возрастала важность поиска нужной информации. Сегодня использование поисковых машин играет настолько важную роль в жизни многих людей, что без них они окажутся в очень трудном положении.

Совершенно очевидно, что успехи в развитии этой технологии оказывают влияние практически на любые аспекты развития нашего общества. Совершенствование компьютерного оборудования позволяет программистам создавать замечательные программы, благодаря которым компьютеры получают повсеместное распространение. Современная фантастика предрекает потрясающие примеры их завтрашнего использования: она уже сейчас предсказывает появление виртуальных миров, полноценное распознавание речи и персонализацию охраны здоровья.

Классификация применения компьютерной техники и характеристики классов компьютеров

Хотя во всех компьютерах, начиная с «умных» бытовых приборов и сотовых телефонов и заканчивая самыми большими суперкомпьютерами, используется одинаковый набор технологий аппаратного обеспечения (см. разделы 1.3 и 1.7), различие в сферах их применения ведет к разным конструктивным требованиям и к разным методам использования базовых технологий аппаратного обеспечения. В общем, по характеру использования компьютеры делятся на три разных класса.

Настольные компьютеры составляют, наверное, наиболее известный вид компьютерной техники. Их типичным представителем является персональный компьютер, которым читатели данной книги наверняка интенсивно пользуются в повседневной жизни. Для настольных компьютеров характерно предоставление отдельным пользователям хорошей вычислительной производительности при низкой стоимости и использование компьютерных программ независимых производителей. Этому классу компьютеров, история развития которого насчитывает всего лишь около 30 лет, обязано развитие многих вычислительных технологий!

Серверы представляют собой современную разновидность всего того, что когда-то было универсальными компьютерами (мэйнфреймами), мини-компьютерами и суперкомпьютерами. Доступ к ним обычно осуществляется только через сеть. Серверы предназначены для выполнения больших объемов работы, которая может состоять либо из одного комплекса прикладных задач — обычно это задачи научного или технического направления, либо из обработки множества мелких заданий, как, например, в случае создания крупного веб-сервера. Эти приложения основаны на программном обеспечении из сторонних источников (например, на системе управления базами данных или на моделирующей системе), но, зачастую, модифицированном и подстроенном под конкретные функции. Серверы создаются с применением тех же основных технологий, которые используются и для настольных компьютеров, но с предоставлением существенно больших возможностей по наращиванию вычислительной мощности и средств ввода-вывода. Как правило, в серверах также уделяется большое внимание функциональной надежности, поскольку их отказы, по сравнению с отказами обычного однопользовательского настольного компьютера, обходятся куда дороже.

Настольный компьютер
Компьютер, разработанный для индивидуального пользования, часто имеющий в своем составе графический дисплей, клавиатуру и мышь.

Сервер
Компьютер, используемый для запуска большого числа программ для множества пользователей, часто обслуживаемых одновременно, и, как правило, доступный только по сети.

Серверы имеют широкий спектр цен и возможностей. Самые дешевые серверы могут стоить чуть больше, чем настольные компьютеры за вычетом цены дисплея или клавиатуры, и цена может составить несколько тысяч долларов. Эти самые дешевые серверы обычно используются для хранения файлов, небольших бизнес-приложений или занимаются обслуживанием сети (см. раздел 6.10). На другом конце ценового диапазона находятся **суперкомпьютеры**, которые в настоящее время содержат от

нескольких сотен до нескольких тысяч процессоров и, как правило, используют память объемом в несколько **терабайт** и внешние хранилища данных объемом в несколько **петабайт**. Их стоимость колеблется от нескольких миллионов до нескольких сотен миллионов долларов. Суперкомпьютеры обычно применяются для выполнения самых сложных научных и технических расчетов, например составления прогнозов погоды, производства нефтеразведочных расчетов, определения структуры белка и решения других крупномасштабных задач.

Несмотря на то что такие суперкомпьютеры олицетворяют собой пик вычислительных возможностей, с точки зрения суммарного дохода они составляют относительно небольшую часть всего компьютерного рынка.

Хотя **центры обработки данных** Интернета, используемые такими компаниями, как eBay и Google, не называются суперкомпьютерами, они также содержат тысячи процессоров, терабайты памяти и петабайты устройств хранения информации. Обычно они рассматриваются в качестве больших компьютерных кластеров (см. главу 7).

Встроенные компьютеры представляют собой самый большой класс компьютеров и имеют самый широкий спектр применения и показателей производительности. К встроенным компьютерам относятся микропроцессоры, которые можно найти в вашем автомобиле, компьютеры в сотовых телефонах, компьютеры в видеоиграх и телевизорах и сети процессоров, содействующие управлению современными самолетами или грузовыми судами. Встроенные компьютерные системы сконструированы для запуска одного приложения или набора взаимосвязанных приложений, которые обычно интегрированы с аппаратной частью и поставляются в виде единой системы, поэтому, несмотря на огромное количество встроенных компьютеров, большинство пользователей даже не понимает, что ими пользуется!

На рис. 1.1 показано, что в течение последних нескольких лет рост количества сотовых телефонов, работа которых основана на применении встроенных компьютеров, был намного выше роста количества настольных компьютеров. Следует иметь в виду, что встроенные компьютеры также можно найти в цифровых телевизорах и телевизионных абонентских приставках, автомобилях, цифровых камерах, музыкальных плеерах, видеоиграх и во многих других им подобных бытовых приборах, которые все больше увеличивают разрыв между количеством встроенных и настольных компьютеров.

К встраиваемым устройствам часто предъявляются весьма специфические потребительские

Суперкомпьютер
Класс компьютеров с наибольшей производительностью и стоимостью. Компьютеры этого класса предназначаются для работы в качестве серверов и обычно стоят миллионы долларов.

Терабайт
Изначально это 1 099 511 627 776 (2^{40}) байт, хотя в некоторых системах обмена данными и вторичных системах хранения данных терабайт переопределяется в 1 000 000 000 0 00 (10^{12}) байт.

Петабайт
В зависимости от ситуации это либо 1000, либо 1024 терабайта.

Центр обработки данных
Помещение или здание, сконструированное для обеспечения питания, охлаждения и сетевых потребностей большого количества серверов.

Встроенный компьютер
Компьютер, находящийся внутри другого устройства и используемый для запуска одного заранее установленного приложения или совокупности программ.

требования, сочетающие в себе минимальную производительность со строгими ограничениями по стоимости или потребляемой мощности. В качестве примера рассмотрим музыкальный плеер: для него самое главное, чтобы процессор обладал минимально необходимым быстродействием, позволяющим справиться со строго ограниченными функциями и, кроме того, снижающим до минимума стоимость и потребляемую мощность. Несмотря на свою низкую стоимость, встроенные компьютеры имеют более низкую терпимость к отказам, потому что последствия этих отказов могут варьироваться от испорченного настроения (когда ломается новый телевизор) до возникновения самых крайних ситуаций (например, при отказе компьютера на самолете или корабле). Во встроенных устройствах, предназначенных для бытового применения, например в цифровых бытовых приборах, гарантия надежности работы достигается в первую очередь за счет их простоты: основной акцент делается на наиболее полноценном выполнении одной-единственной функции. В крупных встроенных системах часто используются технологии дублирования, позаимствованные у серверов (см. раздел 6.9). Хотя в данной книге рассматриваются в основном компьютеры общего назначения, большинство излагаемых в ней концепций применимы, непосредственно или с незначительными изменениями, и к встроенным компьютерам.

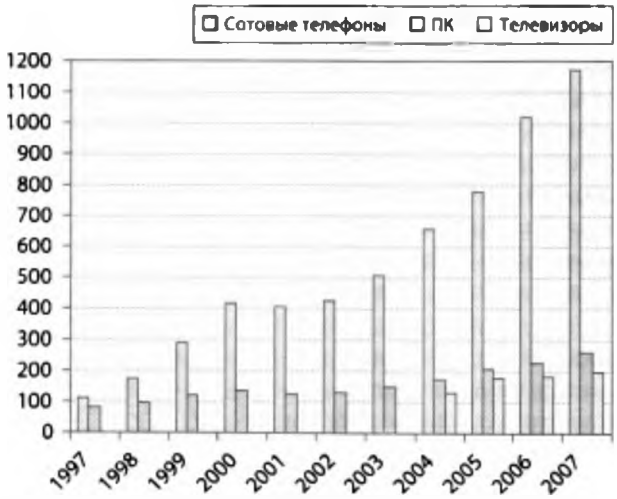


Рис. 1.1. Количество сотовых телефонов, персональных компьютеров и телевизоров, произведенных за период с 1997 по 2007 год. (Сведения о телевизорах получены только начиная с 2004 года.) В 2006 году торговле было поставлено более миллиарда новых сотовых телефонов. В 1997 году продажи сотовых телефонов превышали продажи ПК только в 1,4 раза, но к 2007 году это превышение возросло до 4,5 раза. Общее количество используемой техники в 2004 году оценивалось примерно в 2 млрд телевизоров, 1,8 млрд сотовых телефонов, и 0,8 млрд персональных компьютеров. Так как население земного шара в 2004 году составляло около 6,4 млрд человек, на каждые восемь человек на планете приходился примерно один ПК, 2,2 сотовых телефона, и 2,5 телевизора. Проведенный в 2006 году опрос американских семей показал, что они в среднем являются владельцами 12 электронных устройств, включая три телевизора, два персональных компьютера и несколько других устройств вроде игровых приставок, MP3-плееров и сотовых телефонов

Уточнение. Уточнения — это небольшие подразделы, используемые по всему тексту для предоставления более подробной информации по конкретной теме. Все, кому это неинтересно, могут пропустить уточнение, поскольку последующий материал не будет связан с его содержанием.

Многие встроенные процессоры разработаны с использованием *процессорных ядер*, версии процессора, написанной на языке описания аппаратного обеспечения, таком как Verilog или VHDL (см. главу 4). Ядро позволяет конструктору объединить другое специализированное оборудование с процессорным ядром для изготовления единой микросхемы.

О чем можно узнать, читая эту книгу

Серьезные программисты всегда заботились о производительности своих программ, поскольку быстрое получение пользователем результатов является важным показателем при создании удачных программ. В 60-х и 70-х годах прошлого века основным сдерживающим фактором производительности компьютеров был объем их памяти. Поэтому программисты часто придерживались простого правила: минимизировать пространство используемой памяти, чтобы ускорить работу программ. В последнее десятилетие прогресс в разработке компьютеров и в технологии создания памяти существенно уменьшил важность фактора использования небольшого объема памяти во многих приложениях, кроме тех, которые используются во встроенных компьютерных системах.

Программисты, стремящиеся повысить производительность, теперь должны разбираться в вопросах, пришедших на смену простым моделям памяти 60-х годов: параллельном устройстве процессоров и иерархическом устройстве памяти. Программисты, задавшиеся целью создать конкурентоспособные компиляторы, операционные системы, базы данных и даже простые приложения, нуждаются в совершенствовании своих знаний устройства компьютеров.

Мы рады представившейся возможности объяснить, что находится внутри такой революционной машины, как компьютер, распутывая хитросплетения того программного обеспечения, которое работает на уровень ниже ваших программ, и того оборудования, которое скрыто в корпусе вашего компьютера. Мы верим, что после прочтения данной книги вы сумеете ответить на следующие вопросы:

- Как программы, написанные на языках высокого уровня, например на C или Java, транслируются в язык компьютерного оборудования и как это оборудование выполняет полученную в итоге программу? Усвоение этих понятий позволяет сформировать основу понимания тех аспектов аппаратного и программного обеспечения, которые оказывают влияние на производительность программ.
- Что представляет собой интерфейс программного и аппаратного обеспечения и как программы инструктируют аппаратную часть для выполнения необходимых функций? Без усвоения этих понятий невозможно разобраться в том, как нужно создавать различные виды программ.
- Чем определяется производительность программного обеспечения и как программист может ее повысить? Нам предстоит понять, что производительность

зависит от исходной программы, от транслятора, переводящего программу на язык компьютера, и от эффективности работы оборудования при выполнении этой программы.

- Какие технологии могут использоваться разработчиками компьютерного оборудования для повышения производительности? В этой книге будут представлены основные концепции современной разработки компьютеров.
- Каковы причины и следствия недавнего перехода от последовательной обработки к параллельной? В этой книге приводится мотивация, описываются современные аппаратные устройства поддержки параллельных вычислений и дается обзор нового поколения **«многоядерных» микропроцессоров** (см. главу 7).

Без знания ответов на эти вопросы улучшение производительности ваших программ на современных компьютерах или определение, какие свойства делают один компьютер лучше другого для решения конкретных задач, станет вместо научного подхода, построенного на глубоком понимании сути вопроса и анализе данных, сложным процессом проб и ошибок.

В первой главе закладывается фундамент понимания материала всей остальной книги. Эта глава позволяет ознакомиться с основными идеями и определениями, расставляет по своим местам основные компоненты программного и аппаратного обеспечения, показывает, как оценить производительность и мощность, дает представление об интегральных микросхемах (чье появление инициировало компьютерную революцию), и объясняет причины перехода к многоядерным системам.

В этой и в следующих главах вам, наверное, попадется немало новых слов или тех слов, которые вы уже слышали, но не уверены в правильности понимания их значения. Не стоит переживать! При описании современных компьютеров действительно применяется масса специальных терминов, но эти термины оказывают реальную помощь, позволяя нам точнее описать какую-нибудь функцию или возможность. Кроме того, разработчики компьютеров (включая и авторов этой книги) *любят* использовать **аббревиатуры**, которые *проще* понять, когда известно значение их букв!

Чтобы помочь вам запомнить и распознать в дальнейшем те или иные термины, рядом с местом их первого появления в тексте будут находиться *врезки* с их определениями. Пообвыкнув в употреблении терминов, вы научитесь их выговаривать, впечатляя своих друзей правильным применением таких аббревиатур, как BIOS, CPU, DIMM, DRAM, PCIE, SATA и многих других.

Многоядерный микропроцессор
Микропроцессор, состоящий из нескольких процессоров («ядер»), находящихся в одной интегральной микросхеме.

Аббревиатура
Слово, составленное из первых букв слов в строке. Например: **RAM** является аббревиатура для строки «Random Access Memory», то есть «память с произвольным доступом», или «оперативная память», а **CPU** является аббревиатурой для строки «Central Processing Unit», то есть «центральное процессорное устройство», или просто «центральный процессор».

Чтобы придать особое значение вопросу влияния программного и аппаратного обеспечения, используемого для запуска программы, на ее производительность, по всей книге используются специальные подразделы *Представление о производительности программ*, в которых обобщаются все важные моменты, помогающие проникнуть в суть этого вопроса. Ниже приводится первый из таких подразделов.

Представление о производительности программ

Производительность программ зависит от сочетания эффективности алгоритмов, используемых в программе, программных систем, используемых для создания и трансляции программы в инструкции машины, и эффективности компьютера в выполнении этих инструкций, которые могут включать в себя операции ввода-вывода (I/O). В следующей таблице обобщаются факторы, оказывающие влияние аппаратного и программного обеспечения на производительность.

Компонент аппаратного или программного обеспечения	Как этот компонент влияет на производительность	Где освещается эта тема?
Алгоритм	Определяет как количество инструкций на уровне исходных кодов, так и количество выполняемых операций ввода-вывода	В других книгах!
Языки программирования, компиляторы и архитектура	Определяют количество инструкций компьютера для каждой инструкции на уровне исходного кода	Главы 2 и 3
Процессор и система памяти	Определяют скорость выполнения инструкций	Главы 4, 5 и 7
Система ввода-вывода (оборудование и операционная система)	Определяет, насколько быстро могут быть выполнены операции ввода-вывода	Глава 6

Самопроверка

Подразделы *Самопроверка* предназначены для того, чтобы помочь читателям оценить, насколько ими усвоены основные понятия, представленные в главе, и убедиться в том, что им понятны значения этих понятий. Некоторые вопросы этих подразделов требуют простых ответов, но часть из них предназначена для обсуждения в группе. Ответы на конкретные вопросы могут быть найдены в конце главы. Вопросы подраздела *Самопроверка* появляются только в конце раздела, и если вы уверены в том, что усвоили весь материал, их можно пропустить.

1. В разделе 1.1 показано, что количество встроенных процессоров, проданных в каждом году, существенно превышает количество проданных процессоров для настольных компьютеров. Можете ли вы подтвердить или опровергнуть это наблюдение на основе своего собственного опыта? Попробуйте подсчитать количество встроенных процессоров в своем доме. Как это количество соотносится с количеством настольных компьютеров в вашем доме?
2. Как уже упоминалось, на производительность программы влияет как программное, так и аппаратное обеспечение. Можете ли вы привести примеры, когда главной причиной снижения производительности стало что-нибудь из нижеперечисленного?

- Выбранный алгоритм
- Язык программирования или компилятор
- Операционная система
- Процессор
- Система и устройства ввода-вывода

1.2. Что находится ниже вашей программы

> В Париже на меня просто смотрели в изумлении, когда я говорил им что-нибудь по-французски, и мне ни разу не удалось заставить этих идиотов понимать их собственный язык.
>
> *Марк Твен. «Простаки за границей»*

Обычное приложение, например текстовый процессор или серьезная система управления базами данных, может состоять из нескольких миллионов строк кода и зависеть от использования сложных программных библиотек, реализующих непростые функции, разработанные для поддержки приложений. Далее будет показано, что компьютерное оборудование может выполнять исключительно простые низкоуровневые инструкции. Для перехода от сложных приложений к простым инструкциям нужны еще несколько уровней программного обеспечения, интерпретирующего или транслирующего высокоуровневые операции в простые компьютерные инструкции.

На рис. 1.2 показано, что эти уровни программного обеспечения организованы преимущественно в виде иерархии, в которой приложение находится в самом внешнем кольце, а всевозможное **системное программное обеспечение** располагается между аппаратным обеспечением и прикладными программами.

Существует множество видов системных программ, но два из них играют главную роль в каждой современной компьютерной системе: это операционная система и компилятор. **Операционная система** является посредником между пользовательской программой и оборудованием и предоставляет различные службы и управляющие функции. К самым важным из них относятся:

- обработка основных операций ввода и вывода;
- распределение средств хранения информации и памяти;
- обеспечение защищенного совместного использования компьютера сразу несколькими приложениями.

Примерами современных операционных систем могут служить Linux, MacOS и Windows.

Компиляторы выполняют другую жизненно важную функцию: трансляцию программы, написанной на языке высокого уровня, например

Системное программное обеспечение
Программное обеспечение, предоставляющее службы общего пользования, включая операционные системы, компиляторы, загрузчики и сборщики (ассемблеры).

Операционная система
Программа, управляющая ресурсами компьютера в интересах запущенных на нем программ.

Компилятор
Программа, транслирующая инструкции на языке высокого уровня в инструкции языка ассемблера.

на C, C++, Java или Visual Basic, в инструкции, которые может выполнять оборудование. Учитывая сложность современных языков программирования и простоту инструкций, выполняемых оборудованием, транслятор языка высокого уровня в инструкции оборудования представляет собой весьма непростую программу. Краткий обзор процесса трансляции будет дан в этой главе, а более подробно этот вопрос будет рассмотрен в главе 2.

Рис. 1.2. Упрощенное представление иерархических уровней аппаратного и программного обеспечения, показанное в виде концентрических кругов: центральную часть которых занимает оборудование, а периферическую — прикладные программы. В сложных приложениях зачастую имеется несколько собственных уровней прикладного программного обеспечения. Например, система управления базами данных может запускаться над системным программным обеспечением, являясь базой для приложения, которое, в свою очередь, запускается над базой данных

Переход от языка высокого уровня к языку оборудования

Чтобы общаться с электронным оборудованием, нужно передавать ему электрические сигналы. Простейшими сигналами, понятными компьютеру, являются сигналы *включено* и *выключено*, поэтому в компьютерном алфавите только две «буквы». Точно так же как 26 букв английского алфавита не накладывают никаких ограничений на то, сколько текста может быть написано с их помощью, две буквы компьютерного алфавита не ограничивают возможности компьютера. Двумя символами для этих двух «букв» служат цифры 0 и 1, и компьютерный язык часто представляется в виде чисел по основанию 2, или *двоичных чисел*. Мы обращаемся к каждой «букве» как к **двоичной цифре** — **binary digit**, или **биту** — **bit** (сокращение от binary digit). Компьютеры послушно выполняют наши команды, которые называются инструкциями. Инструкции, представляющие собой всего лишь наборы

битов, которые компьютер понимает и которым он повинуется, могут рассматриваться как числа. Например, биты

```
1000110010100000
```

предписывают компьютеру сложить два числа. В главе 2 объясняется, почему мы используем числа для инструкций *и* данных, и мы не хотели бы забегать вперед, но использование чисел как для инструкций, так и для данных является основой работы всей вычислительной техники.

Первые программисты общались с компьютером посредством двоичных чисел, но это было настолько утомительно, что они быстро изобрели новую систему записи, близкую к образу человеческого мышления. Поначалу эти записи транслировались в двоичный код вручную, но этот процесс был не менее утомителен. Используя компьютер с целью содействия программированию самого компьютера, первопроходцы изобрели программы перевода символьной нотации в двоичную. Первые такие программы были названы **ассемблерами** (то есть сборщиками). Такая программа переводила символьное представление инструкций в их двоичное представление. Например, программист мог написать следующее:

```
add A,B
```

а ассемблер мог перевести эту запись в код:

```
1000110010100000
```

Двоичная цифра
Также называется битом. Одна из двух цифр (0 или 1), используемых для представления чисел по основанию 2, являющаяся информационным компонентом.

Инструкция
Команда, которую понимает и выполняет компьютерное оборудование.

Ассемблер
Программа, транслирующая символьное представление инструкций в их двоичное представление.

Язык ассемблера
Символьное представление инструкций машины.

Машинный язык
Двоичное представление инструкций машины.

Языки программирования высокого уровня
Переносимые языки, например C, C++, Java или Visual Basic, которые состоят из слов и алгебраических форм записи и могут транслироваться компилятором в язык ассемблера.

Эта инструкция предписывает компьютеру сложить два числа A и B. Имя, придуманное для этого символьного языка, — **ассемблер** — используется до сих пор. В отличие от него двоичный язык, понимаемый машиной, называется **машинным языком**.

Несмотря на существенные усовершенствования, язык ассемблера по-прежнему далек от той формы записи, которую ученый предпочел бы для описания потока жидкости или которую бухгалтер мог бы использовать для выведения сальдо. Язык ассемблера требует от программиста записывать по одной строке для каждой инструкции, которую будет выполнять компьютер, заставляя его мыслить по-компьютерному.

Мысль о том, что можно написать программу для трансляции более мощного языка в инструкции компьютера, на заре компьютерной эры была революционной. Сегодняшние программисты могут производительно трудиться благодаря тому, что были созданы **языки программирования высокого уровня** и компиляторы, которые транслируют программы,

написанные на этих языках, в инструкции. На рис. 1.3 показана взаимосвязь между этими программами и языками.

Компилятор позволяет программисту написать следующее выражение на языке высокого уровня:

```
A + B
```

Рис. 1.3. Программа на языке C компилируется в программу на языке ассемблера, а затем ассемблируется в двоичный машинный язык. Хотя здесь показана двухэтапная трансляция с языка высокого уровня в двоичный машинный язык, некоторые компиляторы исключают промежуточный этап и выдают двоичный машинный код напрямую. Более подробно эти языки и такая программа будут рассмотрены в главе 2

Компилятор скомпилирует его в инструкцию на языке ассемблера:

```
add A,B
```

Как показано выше, ассемблер транслирует эту инструкцию в двоичную, которая предпишет компьютеру сложить два числа A и B.

Языки программирования высокого уровня предлагают ряд важных преимуществ. В первую очередь, они позволяют программисту размышлять на более естественном для него языке, используя английские слова[1] и алгебраические формы записи, превращающиеся в программы, которые больше похожи на текст, чем на таблицы криптографических символов (см. рис. 1.3). Более того, они позволяют подстраивать языки под конкретные задачи. Благодаря этому Фортран был разработан для научных вычислений, Кобол — для обработки деловой информации, Лисп — для работы с символами, и т. д. Существуют также проблемно-ориентированные языки для более узкого контингента пользователей, например для тех, кто интересуется моделированием текучих сред.

Вторым преимуществом языков программирования является повышение производительности труда программистов. Одним из редких случаев единодушия в среде разработчиков программного обеспечения стало утверждение, что время на разработку программ уменьшается, когда они пишутся на языках, которые требуют меньшего количества строк для выражения замысла. Лаконичность — очевидное преимущество языков высокого уровня над языком ассемблера.

И завершающее преимущество языков программирования заключается в том, что они позволяют программам быть независимыми от компьютера, на котором они были разработаны, поскольку компиляторы и ассемблеры могут транслировать программы на языках высокого уровня в язык двоичных инструкций любого компьютера. Эти три преимущества настолько сильны, что сегодня на языке ассемблера пишется лишь незначительный объем программ.

1.3. Что скрывается под крышкой корпуса компьютера

Давайте снимем крышку корпуса вашего компьютера и изучим базовые компоненты оборудования. Эти компоненты в любом компьютере выполняют одни и те же основные функции: ввод данных, вывод данных, обработку данных и хранение данных. То, как выполняются эти функции, и является основной темой данной книги, поэтому все последующие главы связаны с различными частями этих четырех задач.

Когда мы в этой книге подходим к какому-нибудь серьезному положению, которое является настолько важным, что мы надеемся на то, что вы запомните его навсегда, мы выделяем это положение в виде подраздела *Самое важное*. В этой книге присутствует около десятка таких элементов, и первым из них будет элемент, рассказывающий о пяти компонентах компьютера, которые выполняют задачи ввода, вывода, обработки и хранения данных.

---

[1] Существуют языки программирования, использующие слова других языков, в том числе русского. Однако среди них нет широко распространенных языков программирования общего назначения. — *Примеч. ред.*

Самое важное

К пяти классическим компонентам компьютера относятся: устройства ввода, вывода, память, операционный блок и блок управления, причем два последних компонента иногда объединяются и называются процессором. На рис. 1.4 показана стандартная структура компьютера. Эта структура не зависит от технологии изготовления оборудования: любую часть любого компьютера из прошлого или настоящего можно отнести к одной из этих пяти категорий. Чтобы помочь вам постоянно держать перед глазами эту структуру, пять компонентов компьютера показаны на первых страницах каждой из следующих глав, с выделением той части, которой посвящена конкретная глава.

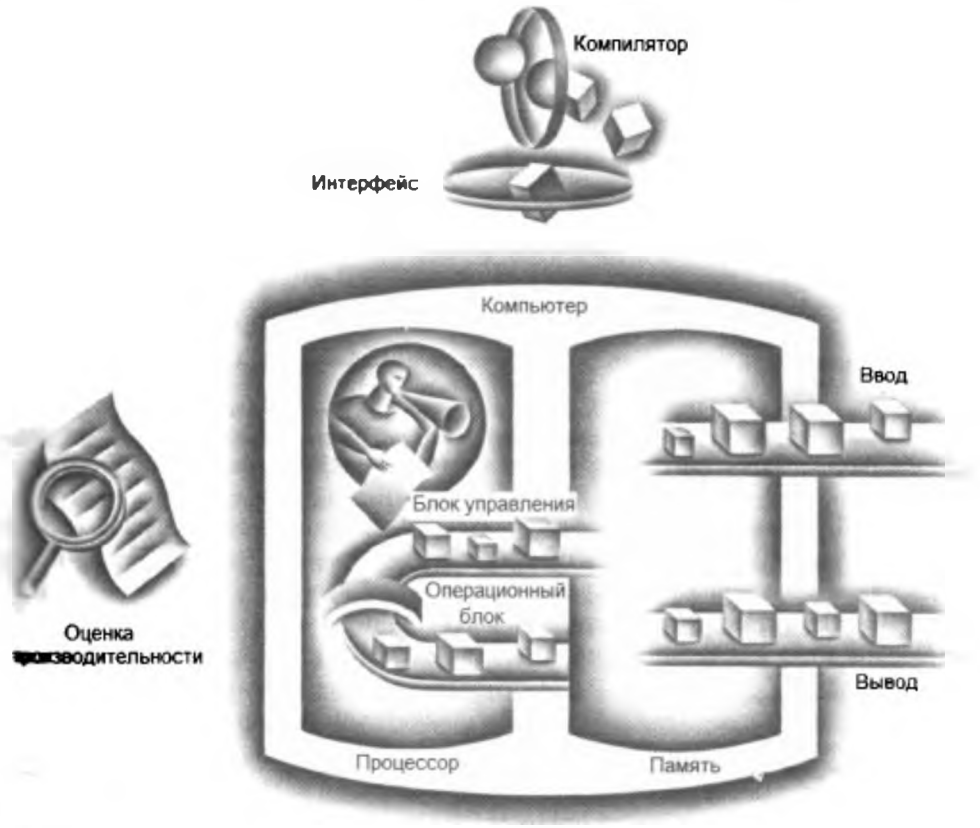


Рис. 1.4. Структура компьютера из пяти классических компонентов. Процессор получает инструкции и данные из памяти. Устройство ввода записывает данные в память, а вывода читает данные из памяти. Блок управления посылает сигналы, определяющие действия операционного блока, памяти, ввода и вывода

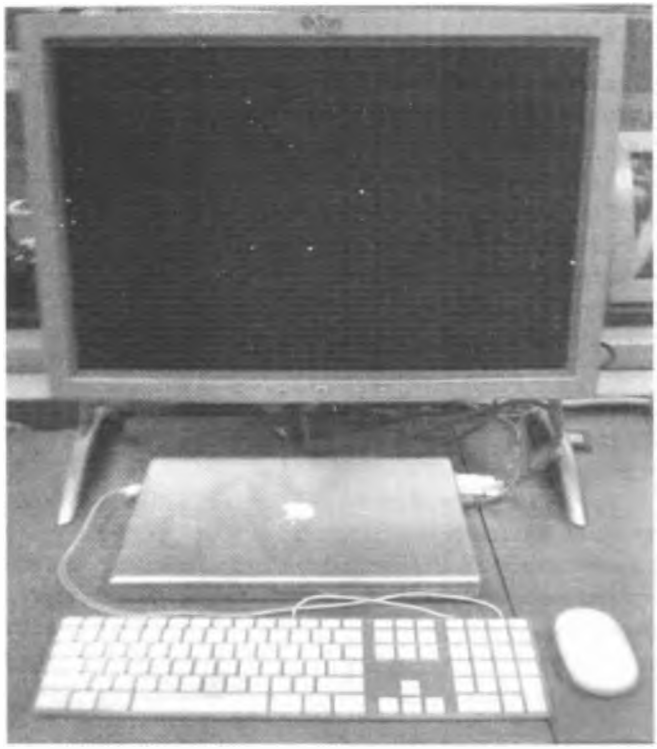


Рис. 1.5. Настольный компьютер. Жидкокристаллический экран (LCD) в качестве первичного устройства вывода и клавиатура с мышью в качестве первичных устройств ввода. С правой стороны виден кабель Ethernet, подключающий портативный компьютер к сети и Интернету. Сам портативный блок состоит из процессора, блока памяти и дополнительных устройств ввода-вывода. На рисунке показан ноутбук Macbook Pro 15", подключенный к внешнему дисплею

На рис. 1.5 показан компьютер с клавиатурой, беспроводной мышью и экраном. На этой фотографии видны два ключевых компьютерных компонента: **устройства ввода**, такие как клавиатура и мышь, и **устройства вывода**, к которым относится экран. В соответствии с их названиями ввод «подпитывает» компьютер данными, а вывод представляет собой результаты вычислений, отправленные пользователю. Некоторые устройства, например сети и диски, предоставляют компьютеру как ввод, так и вывод.

Устройство ввода
Устройство, через которое компьютер снабжается информацией, например клавиатура или мышь.

Устройство вывода
Устройство, которое передает результаты вычисления пользователю или другому компьютеру.

Более подробно устройства ввода-вывода (I/O) рассматриваются в главе 6, но нам нужен ознакомительный тур по компьютерному оборудованию, начинающийся с внешних устройств ввода-вывода.

Анатомия мыши

> Идея мыши пришла мне в голову во время слушаний на компьютерной конференции. Речь докладчика была настолько скучной, что я размечтался и придумал это устройство.
>
> *Дуг Энгельбарт*

> Благодаря дисплею я посадил самолет на палубу движущегося авианосца, видел, как ядерная частица попадала в электростатическую ловушку, летал на ракете с околосветовой скоростью и наблюдал за компьютером, раскрывающим самые сокровенные детали своей работы.
>
> *Айван Сазерленд, «отец» компьютерной графики, американский ученый, 1984 год*

Хотя сегодня многие пользователи воспринимают мышь как нечто само собой разумеющееся, идея использования такого указывающего устройства, как мышь, впервые была продемонстрирована в качестве исследовательского прототипа Дугом Энгельбартом (Doug Engelbart) в 1967 году.

Компьютер Alto, вдохновивший на создание всех рабочих станций, а также операционных систем Macintosh и Windows, включал мышь как указывающее устройство еще в 1973 году. В 90-е годы прошлого века мышью и новым пользовательским интерфейсом на основе графических дисплеев обзавелись все настольные компьютеры.

Самая первая мышь была электромеханической, в ней использовался большой шар, перекатывающийся по поверхности. При этом счетчики координат x и y наращивали свои значения. Скорость приращения показателя каждого счетчика говорила о том, насколько быстро перемещалась мышь.

На смену электромеханическим мышам практически повсеместно пришли более новые, оптические мыши. Эти устройства по своей сути являются миниатюрными оптическими процессорами, включающими светодиод, обеспечивающий подсветку, крошечную черно-белую камеру и простой оптический процессор. Светодиод подсвечивает поверхность под мышью, камера благодаря этой подсветке делает 1500 снимков в секунду. Последовательные снимки отправляются оптическому процессору, который сравнивает изображения и определяет, в какую сторону и насколько быстро была перемещена мышь. Замена электромеханической мыши электрооптической является примером типичного явления, когда удешевление и повышение надежности электроники приводит к тому, что чисто электронные решения заменяют устаревшие электромеханические технологии. Чуть позже будет показан еще один пример — флэш-память.

Что находится за стеклом дисплея

Наверное, самым замечательным устройством ввода-вывода является дисплей. Все ноутбуки и КПК, калькуляторы, сотовые телефоны и практически все настольные

компьютеры сейчас оснащены **жидкокристаллическими (ЖК) дисплеями** (liquid crystal displays, LCD), завоевали популярность благодаря малой толщине и небольшому энергопотреблению. Жидкокристаллический дисплей сам по себе не является источником света, а управляет световым потоком. Обычный ЖК-дисплей состоит из молекул, образующих закрученные спирали, либо отклоняющие свет, попадающий на дисплей из источника позади него, либо, что используется реже, отраженный свет. Под воздействием электрического тока частицы выпрямляются. Поскольку жидкие кристаллы находятся между двумя экранами, поляризация которых смещена относительно друг друга на 90 градусов, свет не может пройти, пока молекулы находятся в скрученном состоянии. Сегодня большинство ЖК-дисплеев используют **активную матрицу** с тонкопленочными транзисторными переключателями для каждого пиксела, которые обеспечивают точное управление электрическим током и улучшают четкость изображения. Красно-зелено-синяя маска, связанная с каждой точкой на экране, определяет интенсивность трех цветовых компонентов в итоговом изображении. В цветных активных матрицах ЖК-дисплеев на каждую точку приходится три транзисторных переключателя.

Изображение составляется из матрицы элементов изображения, или **пикселов** (**pixel** — сокращение от **pic**ture **el**ement), которая может быть представлена в виде матрицы битов, называемой *битовым (двоичным) отображением (bit map)*. В зависимости от величины экрана и разрешения в 2008 году матрицы дисплея имели диапазон размеров от 640 × 480 до 2560 × 1600 пикселов. Цветной дисплей может использовать по 8 бит на каждый из трех цветов (красный, синий и зеленый), итого 24 бита на пиксел, что дает возможность отобразить несколько миллионов цветовых оттенков.

Используемая в компьютерах аппаратная поддержка графики представляет собой *обновляемый растровый буфер*, или *буфер кадра*, в котором хранится его битовое отображение. Изображение для представления на экране хранится в буфере кадра, а конфигурация битов для каждого пиксела считывается с частотой обновления. На рис. 1.6 показан упрощенный буфер кадра, состоящий всего лишь из четырех битов на пиксел.

Задача битового отображения — точное представление того, что находится на экране. Сложность графических систем обусловлена тем, что человеческий глаз способен обнаруживать даже самые незначительные изменения на экране.

Жидкокристаллический дисплей
Дисплей, изготовленный по технологии, использующей тонкий слой жидких полимеров, которые могут использоваться для пропуска или блокировки света соответственно приложенному к ним электрическому напряжению.

Дисплей с активной матрицей
Жидкокристаллический дисплей, использующий транзистор для управления пропуском света для каждого отдельно взятого пиксела.

Пиксел
Наименьший отдельный элемент изображения. В состав экранов входят от нескольких сотен тысяч до нескольких миллионов пикселов, сведенных в матрицу.

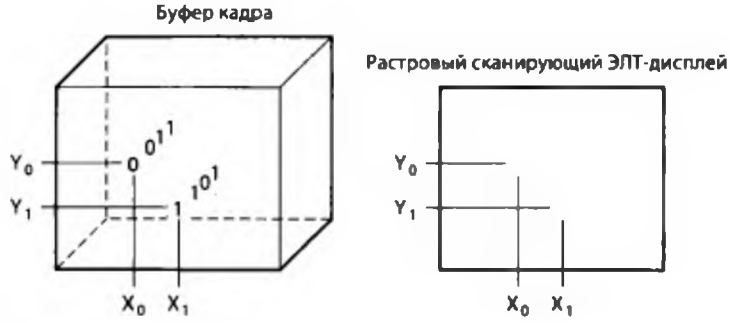


Рис. 1.6. Каждая координата в буфере кадра, показанном слева, определяет оттенок точки с соответствующей координатой для растрового сканирующего ЭЛТ-дисплея, показанного справа. Пиксел (X0, Y0) содержит конфигурацию битов 0011, определяющую более светлый тон, чем конфигурация битов 1101 в пикселе (X1, Y1)

Что скрывается под корпусом компьютера

Если открыть корпус компьютера, станет видна весьма интересная плата из тонкого пластика с десятками небольших серых или черных предметов прямоугольной формы. На рис. 1.7. показано содержимое ноутбука, показанного на рис. 1.5. В верхней части фотографии видна **материнская плата (системная плата)**. В передней части находятся два дисковых привода — жесткий диск слева и DVD-привод справа. Пространство посредине предназначено для батареи ноутбука.

Небольшие объекты прямоугольной формы, находящиеся на материнской плате, являются устройствами, которые определяют развитие нашей передовой технологии. Они называются **интегральными микросхемами**, или, если коротко, **чипами**. Плата состоит из трех частей: из блока, подключенного к упомянутым ранее устройствам ввода-вывода, памяти и процессора.

Память является местом хранения выполняемых программ, она также содержит данные, необходимые этим программам. На рис. 1.8 показано, что память размещается на двух небольших картах, и каждая небольшая карта памяти содержит восемь интегральных микросхем. Память, показанная на рис. 1.8, построена на DRAM-чипах. *DRAM* означает **dynamic random access memory**, то есть динамическая память с произвольным доступом. Несколько DRAM-модулей используются вместе для хранения инструкций и данных программы. В отличие от последовательного доступа к памяти, осуществляемого, к примеру, с использованием магнитной ленты, часть *RAM* аббревиатуры DRAM означает, что доступ к любому фрагменту памяти занимает практически одинаковое количество времени.

Процессор является той самой активной частью материнской платы, которая в точности следует инструкциям программы. Он склады-

Dynamic random access memory (DRAM)
Память, построенная на интегральных микросхемах и предоставляющая произвольный доступ к любому фрагменту.

Модуль памяти с двухрядным расположением выводов (DIMM)
Небольшая плата, содержащая DRAM-чипы. Имеет контакты в виде площадок, расположенных на одном из краев платы по обе ее стороны (то есть в два независимых ряда). (У SIMM-модулей контакты были только с одной стороны.)

вает числа, проверяет их, подает сигналы на активацию устройств ввода-вывода и т. д. Процессор охлаждается вентилятором, и в левой части рис. 1.7 показано что он накрыт радиатором. Иногда процессор называют **ЦПУ (CPU)**, или, в более бюрократичной форме, **центральным процессорным устройством (Central Processor Unit)**.

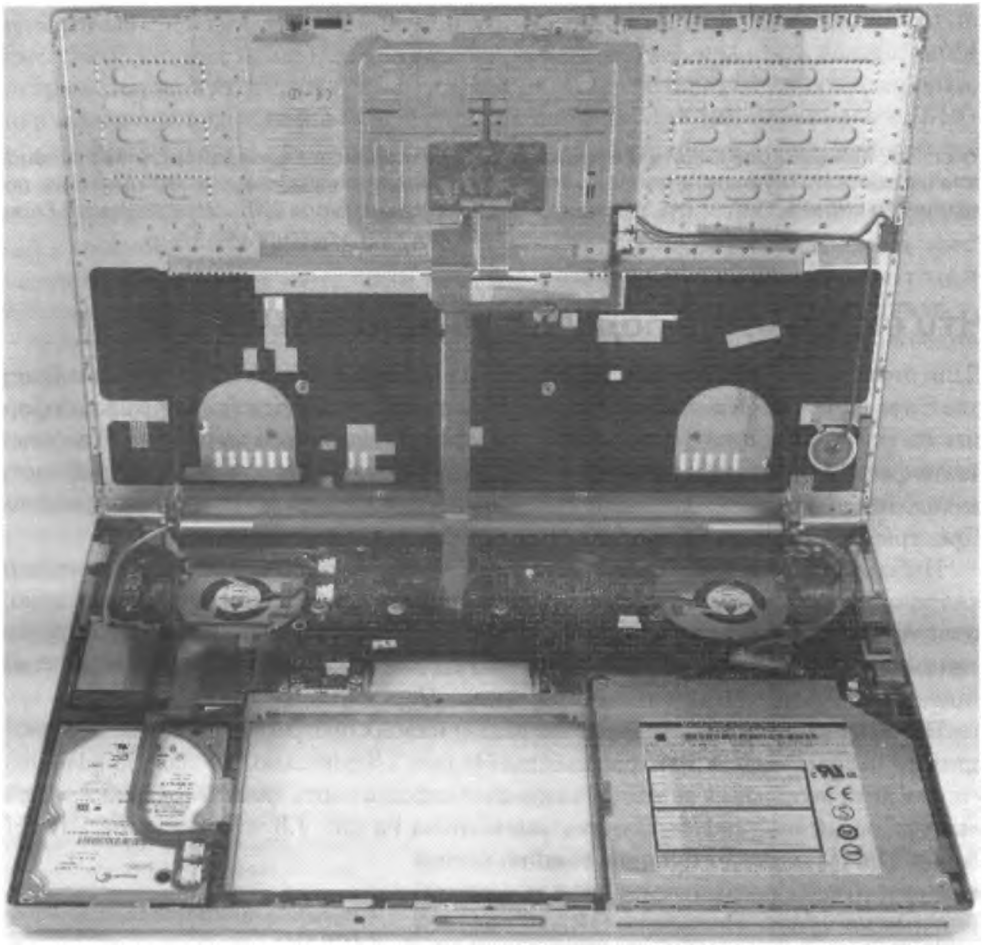


Рис. 1.7. Внутреннее устройство ноутбука с рис. 1.5. Коробка с белой наклейкой с левого края — это жесткий диск на 100 Гбайт с интерфейсом SATA, а коробка с правого края — DVD-привод. Пространство между ними — это место для аккумулятора ноутбука. Небольшое пространство чуть выше аккумуляторного отсека предназначено для DIMM-модулей памяти. На рис. 1.8 изображение DIMM-модулей, вставляемых снизу, дано крупным планом. Выше аккумуляторного отсека и DVD-привода находится печатная плата (плата ПК), называемая материнской которая содержит основную часть электронной начинки компьютера. Два круглых предмета в верхней части фотографии — это вентиляторы в корпусах. Процессор — это большой выпуклый прямоугольный объект, расположенный чуть ниже левого вентилятора.
Фотография любезно предоставлена администрацией сайта OtherWorldComputing.com

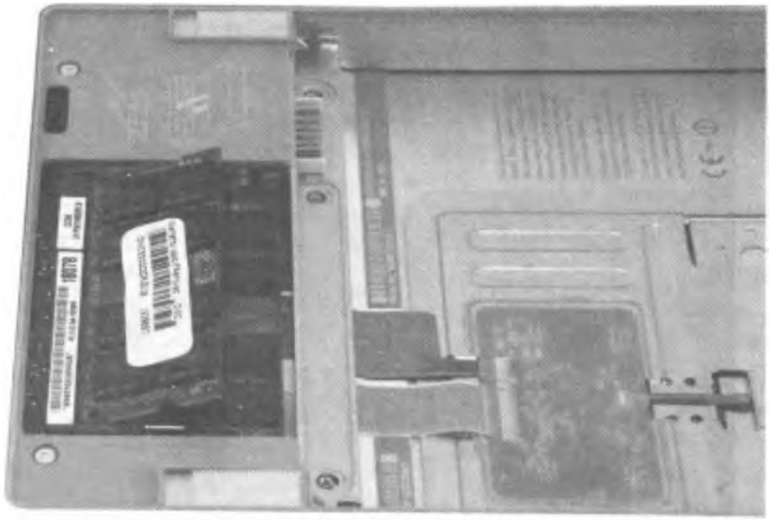


Рис. 1.8. увеличенный фрагмент ноутбука, на котором видны модули памяти. Основная память состоит из одного или нескольких небольших модулей, показанных в левой части фотографии. Пространство для аккумулятора расположено справа. DRAM-чипы установлены на платах (называемых **DIMM**-модулями — от dual inline memory modules — модули памяти с двухрядным расположением выводов), а затем вставлены в контактные колодки.
Фотография любезно предоставлена администрацией сайта OtherWorldComputing.com

Продолжим процесс углубления в компьютерное оборудование: на рис. 1.9 показаны детали микропроцессора. Логически процессор включает в себя два основных компонента: операционный блок и блок управления, то есть, соответственно, «исполнительную» и «мыслительную» части процессора. **Операционный блок** выполняет арифметические операции, а **блок управления** руководит действиями операционного блока, памяти и устройств ввода-вывода в соответствии с предписаниями инструкций программы. Операционные блоки и блоки управления высокопроизводительных конструкций описаны в главе 4.

Глубокое изучение любого компьютерного компонента позволяет лучше разобраться с тем, что происходит внутри компьютера. В процессоре имеется еще один тип памяти — **кэш-память**. Она состоит из быстродействующей памяти небольшого объема, работающей в качестве буфера DRAM-памяти. (Нетехническое определение слова «кэш» — тайник.) Кэш-память построена

Центральное процессорное устройство (ЦПУ)
Также называется процессором. Активная часть компьютера, содержащая операционный блок и блок управления и занимающаяся сложением чисел, их проверкой, подачей сигналов на активацию устройств ввода-вывода и т. д.

Операционный блок
Компонент процессора, выполняющий арифметические операции (также называемый арифметико-логическим устройством, АЛУ).

Блок управления
Компонент процессора, командующий операционным блоком, памятью и устройствами ввода-вывода в соответствии с инструкциями программы.

Кэш-память
Быстродействующая память небольшого объема, работающая в качестве буфера для менее быстрой и более объемной памяти.

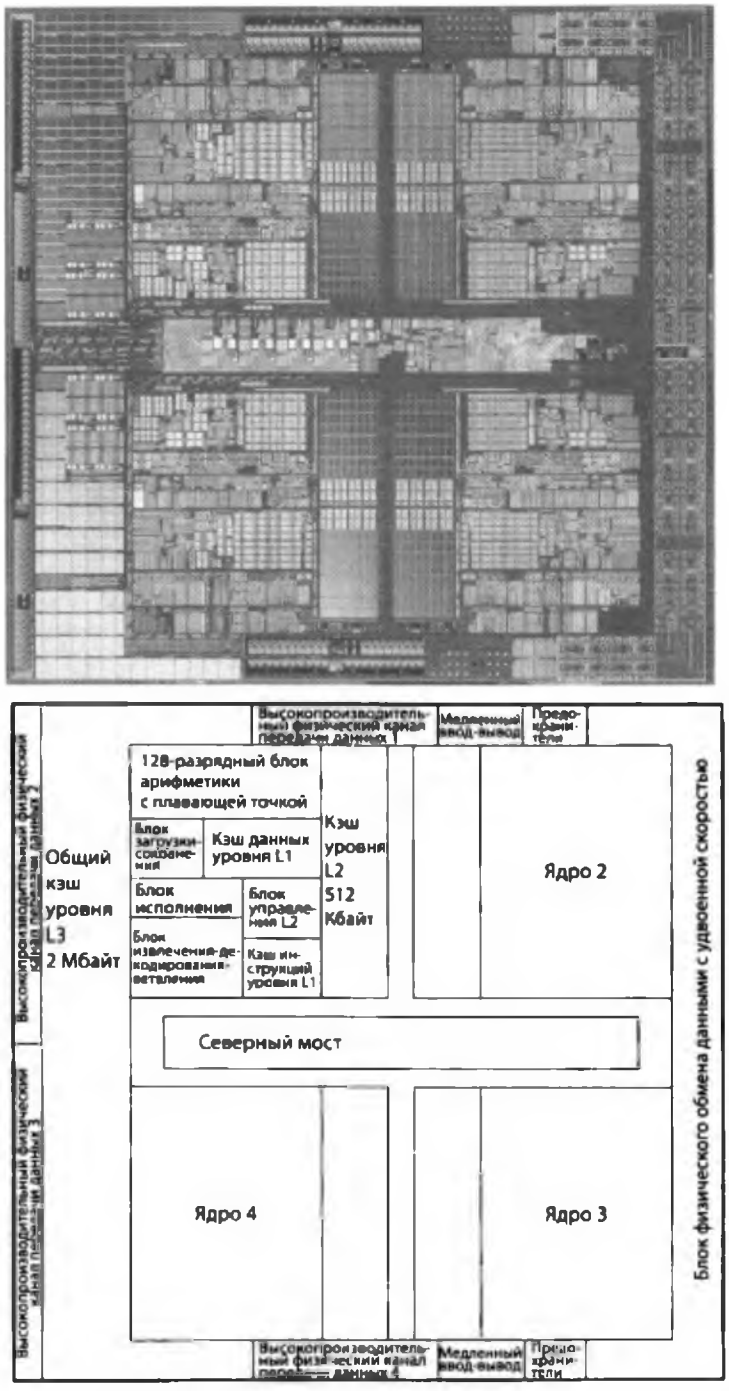


Рис. 1.9. Внутреннее устройство микропроцессора AMD Barcelona. Сверху расположена микрофотография процессора, а снизу показаны его основные блоки. Этот чип имеет четыре процессора, или «ядра». Микропроцессор ноутбука с рис. 1.7, имеющий на чипе два ядра, называется Intel Core 2 Duo

с использованием другой технологии памяти — **static random access memory (SRAM)**, то есть статической памяти с произвольным доступом. SRAM имеет более высокое быстродействие, но меньшую плотность элементов, и она дороже DRAM (см. главу 5).

В описании как программного, так и аппаратного обеспечения можно было заметить общий подход: при более глубоком изучении оборудования или программ открывается все больше информации, а подробности более низких уровней скрываются, чтобы представить более простую модель для более высоких уровней. Использование этих уровней, или **абстракций**, является основным технологическим приемом при разработке очень сложных компьютерных систем.

Одна из наиболее важных абстракций — это интерфейс между аппаратной частью и программным обеспечением самого низкого уровня. В силу ее важности она имеет специальное название: **архитектура набора команд**, или просто **архитектура** компьютера. Архитектура набора команд содержит все, что нужно знать программисту для создания работоспособных программ на двоичном машинном языке, включая инструкции, характеристики устройства ввода-вывода и т. д. Как правило, операционная система будет скрывать в своих модулях подробности осуществления ввода-вывода, распределения памяти и выполнения других низкоуровневых системных функций, поэтому прикладным программистам беспокоиться не стоит. Сочетание основного набора инструкций и интерфейса операционной системы называется **двоичным интерфейсом приложений** (application binary interface, **ABI**).

Архитектура набора инструкций позволяет разработчикам компьютеров вести речь о функциях независимо от выполняющей их аппаратной части. Например, можно говорить о функциях цифровых часов (хранении показаний времени, отображении времени, установке будильника) независимо от их аппаратной части (кварцевого резонатора, ЖК-дисплея, пластмассовых кнопок).

Разработчики компьютеров рассматривают архитектуру отдельно от ее **реализации**, придерживаясь тех же правил: реализация — это оборудование, удовлетворяющее требованиям архитектурной абстракции. Эти идеи подводят нас к еще одному подразделу «Самое важное».

Static random access memory (SRAM)
Память, также построенная на базе микросхемы, но имеющая более высокое быстродействие и меньшую плотность элементов по сравнению с DRAM.

Абстракция
Модель, дающая представление о временно невидимых низкоуровневых особенностях компьютерных систем и облегчающая проектирование сложных систем.

Архитектура набора инструкций
Также называется просто **архитектурой**. Абстрактный интерфейс между аппаратной частью и низкоуровневым программным обеспечением, удовлетворяющий все информационные потребности для написания работоспособных программ на машинном языке, включая инструкции, регистры, доступ к памяти ввод-вывод и т. д.

Двоичный интерфейс приложений (ABI)
Пользовательская часть набора инструкций плюс интерфейсы операционной системы, используемые прикладными программистами. Определяет стандарт для обеспечения двоичной переносимости между компьютерами.

Реализация
Оборудование, удовлетворяющее требованиям архитектурной абстракции.

Самое важное

Как аппаратное, так и программное обеспечение состоит из иерархических уровней, и каждый уровень скрывает подробности от вышестоящего уровня. Этот принцип *абстракции* является способом справиться со сложностью компьютерных систем, как для разработчиков аппаратной части, так и для разработчиков программного обеспечения. Ключевым интерфейсом между уровнями абстракции является *архитектура набора инструкций* — интерфейс между аппаратурой и низкоуровневым программным обеспечением. Этот абстрактный интерфейс позволяет многим *реализациям*, разным по стоимости и производительности, работать с одинаковым программным обеспечением.

Надежное место для хранения данных

Мы уже поняли, как данные вводятся, используются для вычислений и отображаются. Но если пропадет питание компьютера, все они будут потеряны, поскольку память внутри компьютера **энергозависима**, то есть при отсутствии питания она все «забывает». В отличие от нее, DVD-диск не «забывает» записанный на него фильм, когда вы выключаете DVD-плеер, благодаря используемой в нем технологии **энергонезависимой** памяти.

Энергозависимая память
Запоминающее устройство, например DRAM, которое сохраняет данные только при получении питания.

Энергонезависимая память
Разновидность памяти, сохраняющая данные даже при отсутствии источника питания. Используется для хранения программ между сеансами работы. К энергонезависимой памяти относятся, например, магнитные диски.

Вторичная память
Энергонезависимая память, используемая для хранения программ и данных между сеансами работы. На современных компьютерах она обычно представлена магнитными дисками.

Магнитный диск
Также называется жестким диском. Разновидность энергонезависимой вторичной памяти. Жесткий диск состоит из вращающихся дисков, покрытых материалом для магнитной записи.

Флэш-память
Энергонезависимая полупроводниковая память. Более дешевая и менее быстродействующая по сравнению с DRAM, но более дорогая и более быстродействующая по сравнению с магнитными дисками.

Чтобы различить энергозависимую память, которая используется для хранения данных и программ во время работы, и энергонезависимую память, которая используется для хранения данных и программ между сеансами работы, для первой из них используется термин **основная**, или **первичная, память**, а для второй — **вторичная память**. DRAM-память (основная память), занимает доминирующее положение начиная с 1975 года, а **магнитные диски** (вторичная память), с 1965 года. Главным энергонезависимым запоминающим устройством, используемым на всех серверных компьютерах и рабочих станциях, является **жесткий диск**. **Флэш-память**, энергонезависимая полупроводниковая память, используется вместо дисков в мобильных устройствах, таких как сотовые телефоны, и она все чаще заменяет диски в музыкальных плеерах и даже в ноутбуках.

На рис. 1.10 показано, что магнитный жесткий диск состоит из набора дисков, вращающихся на общей оси со скоростью от 5400 до 15 000 оборотов в минуту. Металлические диски

с обеих сторон покрыты материалом для магнитной записи, похожим на тот материал, который используется в магнитофонных кассетах или видеолентах. Для чтения и записи информации на жестком диске имеется *подвижный элемент*, содержащий небольшую электромагнитную катушку, называемую *головкой чтения-записи*, которая находится непосредственно над поверхностью. Весь привод постоянно находится в закрытом состоянии, что позволяет контролировать состояние его внутренней среды и что, в свою очередь, позволяет головкам диска быть ближе к поверхности диска.

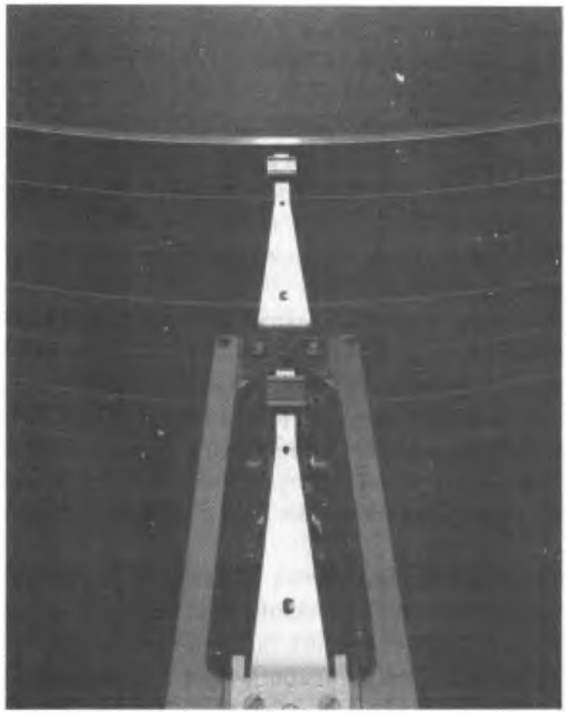


Рис. 1.10. Жесткий диск, у которого видны 10 дисков и головки чтения-записи

Диаметр жестких дисков сегодня варьируется более чем в три раза, от одного до трех с половиной дюймов. С годами диаметр уменьшался. Новые форм-факторы дисков понадобились всем устройствам: рабочим станциям, серверам, персональным компьютерам, ноутбукам, КПК и цифровым видеокамерам. Обычно самые большие диски обладают самой высокой производительностью, а самые маленькие имеют самую низкую стоимость. При этом стоимость хранения одного **гигабайта** данных имеет разное значение. Хотя большинство жестких дисков, как и тот диск, что показан на рис. 1.7, находятся внутри компьютеров, могут быть также жесткие

Гигабайт
Обычно в одном гигабайте насчитывается 1 073 741 824 (2^{30}) байт, хотя эта величина для некоторых систем связи и вторичных систем хранения данных переопределена и означает 1 000 000 000 (10^9) байт. Аналогично этому, в зависимости от контекста мегабайт может означать либо 2^{20}, либо 10^6 байт.

диски, подключенные с использованием таких внешних интерфейсов, как универсальная последовательная шина (USB).

Использование механических компонентов приводит к тому, что время доступа к данным для магнитных дисков существенно превышает этот же показатель для DRAM-памяти: у дисков оно составляет 5–20 миллисекунд, а для DRAM-памяти оно имеет значение 50–70 наносекунд, что повышает быстродействие DRAM-памяти в 100 000 раз. Но диски при одинаковой емкости пока еще стоят значительно дешевле DRAM-памяти, потому что себестоимость для заданного объема дискового устройства ниже себестоимости памяти такого же объема, основанной на использовании интегральных микросхем. В 2008 году стоимость одного гигабайта дисковой памяти была в 30–100 раз ниже стоимости такого же объема DRAM-памяти.

Таким образом, магнитные диски отличаются от оперативной памяти по трем главным показателям: диски являются энергонезависимыми, потому что они магнитные; у них медленнее осуществляется доступ к данным, потому что они являются механическими устройствами; и хранение гигабайта данных на них обходится дешевле, поскольку у них очень высокий объем хранящихся данных при относительно низкой стоимости.

Чтобы заполнить образовавшийся разрыв в показателях, многие пытались изобрести такую технологию памяти, которая была бы дешевле DRAM-памяти, но работала быстрее жесткого диска, но все эти попытки терпели неудачу. Претендентам никогда еще не удавалось поставить подобный продукт на рынок в подходящий момент. До времени предполагаемого появления нового продукта на рынке DRAM-память и диски будут продолжать быстро совершенствоваться, соответственно будут снижаться затраты на их изготовление, и потребности в претендующем на их место продукте тут же отпадут.

Но серьезный претендент в виде флэш-памяти все же появился. Эта полупроводниковая память является энергонезависимой, как и диски, имеет примерно такую же, как у них, пропускную способность, но время ожидания доступа к данным у нее в 100–1000 раз меньше, чем у дисков. Флэш-память получила распространение в видеокамерах и переносных музыкальных плеерах благодаря существенно меньшим размерам, более высокой прочности и значительно меньшему энергопотреблению, чем у дисков, несмотря на то что стоимость хранения гигабайта данных по состоянию на 2008 год у нее была в 6–10 раз выше, чем у диска. В отличие от дисков и DRAM-памяти биты флэш-памяти выдерживают всего лишь от 100 000 до 1 000 000 циклов перезаписи. Поэтому файловая система должна отслеживать количество операций записи и использовать приемы, позволяющие избегать истощения среды хранения данных, перемещая время от времени наиболее востребованные данные. Более подробно флэш-память рассматривается в главе 6.

Хотя жесткие диски не относятся к съемным устройствам, в качестве последних может послужить ряд других устройств:

- Оптические диски, включая компакт-диски (CD) и цифровые видеодиски (DVD), которые составляют наиболее распространенный тип съемных устройств

хранения данных. Бесспорным последователем DVD-дисков становятся оптические диски стандарта Blu-Ray (BD).

- Съемные карты флэш-памяти, обычно подключаемые к разъему USB и используемые для переноса файлов с устройства на устройство.
- Магнитные ленты, предоставляющие медленный последовательный доступ, и используемые для создания резервных копий дисков; эту же роль на себя сейчас часто берут дублирующие жесткие диски.

Оптические диски функционируют иначе, чем магнитные жесткие диски. На компакт-дисках данные записываются по спирали, а отдельные биты записываются путем прожига на поверхности диска небольших углублений диаметром примерно в один микрон (10^{-6} метра). Чтение осуществляется путем освещения поверхности компакт-диска лазерным лучом, а наличие углубления или ровной (отражающей) поверхности определяется путем регистрации отраженного света. В DVD-дисках используется такая же регистрация степени отражения лазерного луча от серий углублений и участков ровной поверхности. Вдобавок к этому у DVD-дисков бывает несколько слоев, на которых может фокусироваться лазерный луч, а размер поверхности, используемой под запись каждого бита, значительно меньше, что в совокупности существенно повышает емкость носителя. В Blu-Ray-приводах используются лазеры с меньшей длиной волны, что еще больше уменьшает поверхность, необходимую для записи одного бита, повышая тем самым емкость носителя.

В устройствах записи оптических дисков, используемых в компьютерах, для создания углублений на предназначенном для записи слое, покрывающем поверхность компакт-диска или диска DVD, применяется лазер. Этот процесс записи происходит относительно медленно, занимая от нескольких минут (для записи полного компакт-диска) до нескольких десятков минут (для записи полного диска DVD). Поэтому для производства больших тиражей используется другая технология, называемая *штамповкой*, позволяющая снизить себестоимость одного оптического диска буквально до нескольких пенсов.

Перезаписываемые компакт-диски и диски DVD используют несколько иную поверхность для записи, имеющую отражающий материал с кристаллической структурой. На этой поверхности формируются участки, не отражающие лазерный луч, примерно так же, как это делается в компакт-дисках и дисках DVD, предназначенных для однократной записи. Для стирания компакт-диска или диска DVD поверхность нагревают и медленно охлаждают, позволяя процессу нормализации восстановить слой поверхности записи и вернуть ему первоначальную кристаллическую структуру. Перезаписываемые диски относятся к самым дорогим, однократно записываемые диски обходятся дешевле, а самыми дешевыми и менее затратными при записи являются диски, предназначенные только для чтения, которые используются для распространения программного обеспечения, музыки или фильмов.

Обмен данными с другими компьютерами

Мы уже объяснили, как происходит ввод, обработка, отображение и хранение данных, но осталась еще одна неохваченная тема, определяющая облик современных компьютеров: компьютерные сети. Точно так же как процессор, показанный на рис. 1.4, связывается с памятью и с устройствами ввода-вывода, сети связывают целые компьютеры, позволяя пользователям компьютеров расширять их возможности, в том числе за счет обмена информацией. Сети получили настолько широкое распространение, что стали основой современных компьютерных систем, а новый компьютер без необязательного сетевого интерфейса всерьез вообще не воспринимается. Включенные в сеть компьютеры имеют ряд серьезных преимуществ:

- *Обмен данными*. Компьютеры обмениваются информацией на высоких скоростях.
- *Использование общих ресурсов*. Вместо того чтобы обеспечить каждый компьютер устройствами ввода-вывода, можно совместно использовать всеми компьютерами в сети.
- *Удаленный доступ*. Благодаря связи компьютеров, находящихся на значительном удалении друг от друга, пользователям не нужно находиться в непосредственной близости от конкретного компьютера.

Сети различаются по протяженности и производительности, а стоимость обмена данными увеличивается соответственно скорости обмена и расстояния, на которую передается информация. Наверное, наиболее популярным типом сети является *Ethernet*. Длина сети может достигать километра, и по ней может передаваться до 10 гигабит в секунду. Благодаря этому Ethernet используется для объединения компьютеров, расположенных, например на одном этаже здания, и обычно называется **локальной вычислительной сетью**. Такие сети объединяются с помощью коммутаторов, которые, кроме этого, могут обеспечивать маршрутизацию и безопасность. **Глобальные сети** пересекают континенты и являются основой Интернета, который поддерживает так называемую Всемирную паутину (World Wide Web). Обычно такие сети основаны на использовании оптоволоконных каналов и арендуются у телекоммуникационных компаний.

За последние 25 лет сети буквально изменили облик вычислительных систем, став еще более распространенными и намного более производительными. В 70-х годах прошлого века к электронной почте имел доступ только весьма ограниченный круг пользователей, Интернета еще не было, и основным способом передачи большого объема данных между двумя географическими точками была пересылка магнитных лент по почте. Локальных сетей практически не было, а несколько существующих глобальных сетей имели скромные возможности и ограниченный доступ.

По мере совершенствования сетевых технологий реализация сетей становилась дешевле

Локальная вычислительная сеть (local area network, LAN)
Сеть, разработанная для обмена данными внутри территориально ограниченной области, обычно внутри отдельного здания.

Глобальная сеть (wide area network, WAN)
Сеть, распространяющаяся на сотни километров, способная охватить целый континент.

и они приобретали все больше возможностей. Например, первая приведенная к стандарту локальная вычислительная сеть, разработанная 25 лет назад, была одной из версий Ethernet, обладавшей максимальной пропускной способностью (также называемой полосой пропускания) в 10 миллионов бит в секунду, и совместно использовалась несколькими десятками, но не более чем сотней, компьютеров. Сегодня технологии локальных вычислительных сетей предлагают пропускную способность от 100 миллионов бит в секунду до 10 гигабит в секунду, в большинстве случаев совместно используемую всего лишь несколькими компьютерами. Технологии оптической связи позволили добиться примерно такого же роста пропускной способности и для глобальных сетей — от нескольких сотен килобит в секунду до нескольких гигабит в секунду, повысив при этом число подключенных к всемирной сети компьютеров от нескольких сотен до многих миллионов. Это сочетание невероятного распространения сетей с повышением их пропускной способности сделало сетевые технологии основой информационной революции, происшедшей за последние 25 лет.

За последние десять лет благодаря еще одной сетевой инновации компьютерные коммуникации изменились. Широкое распространение получили беспроводные технологии, и теперь соответствующее оборудование стало неотъемлемой принадлежностью ноутбуков. Возможность реализации радиосвязи с использованием все той же низкозатратной полупроводниковой технологии (CMOS), которая используется для памяти и микропроцессоров, позволила существенно снизить стоимость и привела к бурному развертыванию беспроводных сетей. Доступные на сегодня беспроводные технологии, которые по стандарту IEEE называются 802.11, позволяют достичь скорости передачи данных от одного до почти 100 миллионов бит в секунду. Беспроводные сети существенно отличаются от проводных сетей, поскольку все пользователи оказываются рядом в непосредственной близости за счет радиоволн.

Самопроверка

- Полупроводниковая DRAM-память и дисковый накопитель существенно отличаются друг от друга. Опишите основные отличия по каждой из следующих характеристик: энергозависимости, времени доступа к данным и стоимости.

Технологии создания процессоров и памяти

Совершенствование процессоров и памяти происходило быстрыми темпами, потому что разработчики компьютеров в гонке по созданию самого лучшего компьютера использовали самые последние достижения в области электроники. В табл. 1.1 показаны технологии, которые использовались в разное время, с оценкой каждой из них по показателю отношения сравнительной производительности к себестоимости единицы продукции. В разделе 1.7 рассматривается технология, которая стимулировала компьютерную промышленность начиная с 1975 года и будет продолжать ее стимулировать в обозримом будущем. Поскольку эта технология

определяет возможности компьютеров и скорость их развития, мы уверены в том, что все профессионалы компьютерного дела должны быть знакомы с основами интегральных микросхем.

Таблица 1.1. Отношение сравнительной производительности к себестоимости единицы продукции для технологий, использовавшихся в компьютерах в разные времена.

Год	Технология, которая использовалась в компьютерах	Отношение сравнительной производительности к себестоимости единицы продукции
1951	Электронная лампа	1
1965	Транзистор	35
1975	Интегральная микросхема	900
1995	Сверхбольшая интегральная схема	2 400 000
2005	Ультрабольшая интегральная схема	6 200 000 000

Источник: Компьютерный музей, г. Бостон. Данные по 2005 году экстраполированы авторами.

Транзистор — это простой переключатель, управляемый электрическим сигналом и работающий по принципу включено-выключено. *Интегральная микросхема* (*integrated circuit, IC*) объединяет внутри одного чипа от десятков до сотен транзисторов. Чтобы описать гигантское увеличение количества транзисторов с сотен до миллионов штук, к термину было добавлено определение «сверхбольшая» — *very large scale*, из которого сложилась аббревиатура *VLSI*, означающая **very large-scale integrated circuit**, то есть **сверхбольшая интегральная схема**.

Темпы возрастания интеграции сохраняли удивительную стабильность. На рис. 1.11 показано возрастание плотности элементов DRAM-памяти начиная с 1977 года. За последние 20 лет промышленность каждые три года четырехкратно увеличивала плотность элементов, в результате чего она выросла более чем в 16 тысяч раз! Это увеличение количества транзисторов в интегральных микросхемах известно как закон Мура, который утверждает, что плотность транзисторов удваивается каждые 18–24 месяца. Закон Мура был выведен из предсказания подобного роста емкости интегральных микросхем, сделанного в 60-х годах Гордоном Муром (Gordon Moore), одним из основателей компании Intel.

Такие темпы роста на протяжении почти 40 лет потребовали глобальных изменений в технологиях производства. Производство интегральных микросхем рассматривается в разделе 1.7.

Электронная лампа
Электронный компонент, предшественник транзистора, представляющий собой полый стеклянный баллон высотой 5–10 см, из которого выкачано максимально возможное количество воздуха и в котором для передачи данных используется пучок электронов.

Транзистор
Переключатель «включено-выключено», управляемый электрическим сигналом.

СВЕРХБольшая интегральная схема (СБИС, или VLSI)
Устройство, содержащее от сотен тысяч до миллионов транзисторов.

1.4. Производительность

Оценить производительность компьютера бывает довольно непросто. Разнообразие и сложность современных программных систем в сочетании с широким спектром технологий улучшения производительности, применяемых разработчиками оборудования, сделали оценку производительности еще сложнее.

При выборе компьютера из нескольких разных моделей производительность считается весьма важным свойством. Покупателям, а следовательно и разработчикам, важна точность оценки и сравнения различных компьютеров. Знают об этом и продавцы компьютеров. Зачастую продавцам хочется представить компьютер в наилучшем свете, независимо от того, отвечает ли он запросам потребителя. Поэтому при выборе компьютера нужно знать, как наилучшим образом определить уровень производительности и какие ограничения существуют при измерении производительности.

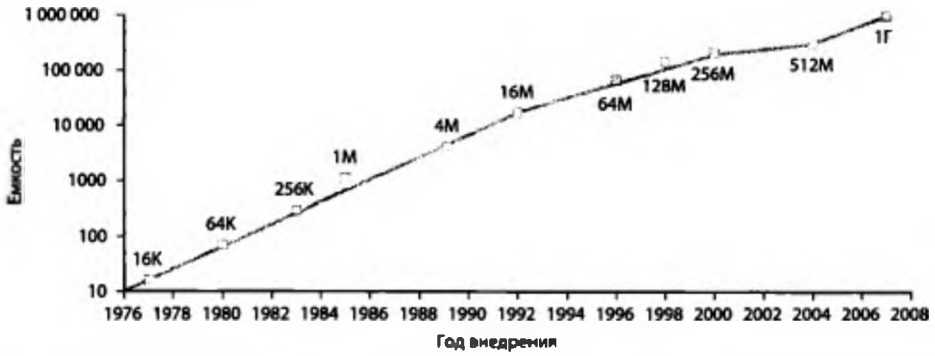


Рис. 1.11. Рост емкости DRAM-чипов. Значения по оси Y выражены в килобитах. Производители DRAM вчетверо увеличивали емкость каждые три года — 60% в год на протяжении 20 лет. В последние годы темпы роста снизились и стали близки к удвоению емкости каждые два-три года

Далее речь пойдет о различных способах определения производительности систем показателей для оценки производительности с точки зрения как пользователя компьютера, так и разработчика компьютерных систем. Также рассматривается соотношение этих показателей и дается классическое уравнение производительности процессора, которое будет использоваться по всему тексту.

Определение производительности

Что мы имеем в виду, когда говорим, что один компьютер по сравнению с другим обладает более высокой производительностью? Хотя этот вопрос может показаться очень простым, приводимая здесь аналогия с пассажирами самолетов показывает, насколько коварен может быть вопрос о производительности. В табл. 1.2 показаны обычные пассажирские самолеты, а также их крейсерская скорость, дальность и вместимость. Если нужно узнать, какой из самолетов, перечисленных в таблице, обладает лучшей производительностью, сначала следует определить само понятие

производительности. Например, рассматривая различные оценки производительности, мы выяснили, что самая высокая крейсерская скорость у Конкорда, самолет с наилучшим показателем дальности полета — DC-8, а наилучшая вместимость у Боинга 747.

Предположим, что производительность определяется в понятиях скорости. Тогда остаются еще два возможных определения. Самым быстрым самолетом можно считать тот, у которого самая высокая крейсерская скорость и который перевозит одного пассажира из одной точки в другую за наименьшее время. Если вы заинтересованы в перевозке из одной точки в другую 450 пассажиров, то, очевидно, как показано в последней графе, самым быстрым будет Боинг 747. Аналогичным образом, производительность компьютера также можно определять несколькими разными способами.

Таблица 1.2. Вместимость, дальность и скорость некоторых коммерческих самолетов

Самолет	Вместимость	Дальность, миль	Крейсерская скорость, миль в час	Пропускная способность, пассажиров × миль в час
Боинг 777	375	4630	610	228 750
Боинг 747	470	4150	610	286 700
Конкорд	132	4000	1350	178 200
Дуглас DC-8-50	146	8720	544	79 424

Последняя графа показывает оценку возможности самолета по перевозке пассажиров, представляющую собой произведение вместимости и крейсерской скорости (при этом игнорируется дальность и время взлета-посадки).

Если запустить программу на двух разных настольных компьютерах, то вы скажете, что быстрее из них работает тот, который справится с заданием первым. Если работать с центром обработки данных, имеющим несколько серверов, выполняющих задания, отправленные многими пользователями, вы скажете, что самым быстрым был тот компьютер, который выполнил за день наибольшее количество заданий. Как индивидуальный компьютерный пользователь, вы заинтересованы в сокращении **времени отклика** — то есть времени между запуском и завершением задачи, — которое также называется **временем выполнения**. Руководители центров обработки данных часто заинтересованы в повышении **пропускной способности** — общего объема работы, выполненного за заданное время. Следовательно, в большинстве случаев нам нужны разные оценки производительности,

Время отклика (Response time)
Также называется временем выполнения (execution time). Общее время, требующееся компьютеру для завершения задачи, включая время доступа к диску, памяти, активации устройств ввода-вывода, издержек операционной системы, времени выполнения задачи центральным процессором, и т. д.

Пропускная способность (troughput, bandwidth)
Еще один показатель производительности, представляющий собой количество задач, выполненных в единицу времени.

а также разные наборы приложений для сравнения встроенных и настольных компьютеров, для которых более значимо время выполнения, в отличие от серверов, для которых более значима пропускная способность.

Упражнение

Пропускная способность и время отклика

К чему приведут следующие изменения в компьютерной системе: к увеличению пропускной способности, уменьшению времени отклика или и к тому и к другому?

- Замена процессора более быстродействующим.
- Добавление процессоров к системе, которая использует несколько процессоров для разделения задач, например для проведения поиска во Всемирной паутине.

Ответ

Уменьшение времени отклика почти всегда увеличивает пропускную способность. Следовательно, в первом случае улучшатся как показатели времени отклика, так и показатели пропускной способности. Во втором случае ни одна из задач не будет выполняться быстрее, следовательно, повысится только пропускная способность.

Но если бы во втором случае время, требуемое на обработку, было сопоставимо с пропускной способностью, система могла бы заставить запросы выстроиться в очередь. В таком случае повышение пропускной способности могло бы также улучшить время отклика, поскольку оно уменьшило бы время ожидания в очереди. Таким образом, на многих реально существующих компьютерных системах изменение одного показателя — либо времени выполнения, либо пропускной способности — зачастую влечет за собой изменение и другого показателя.

При обсуждении производительности компьютеров на протяжении первых нескольких глав нас в основном будет интересовать время отклика. Для увеличения производительности нам требуется минимизировать время отклика или время выполнения какой-нибудь задачи. Таким образом, мы можем связать производительность и время выполнения для компьютера X следующей формулой:

$$\text{Производительность}_X = \frac{1}{\text{Время выполнения}_X}.$$

Это означает, что для двух компьютеров X и Y, если производительность X выше, чем производительность Y, мы получаем следующее:

$$\text{Произодительность}_X > \text{Производительность}_Y$$

$$\frac{1}{\text{Время выполнения}_X} > \frac{1}{\text{Время выполнения}_Y}.$$

$$\text{Время выполнения}_X > \text{Время выполнения}_Y$$

То есть время выполнения на компьютере Y больше, чем на компьютере X, если X быстрее, чем Y.

При обсуждении конструкции компьютера нам часто хочется связать производительность двух разных компьютеров количественным показателем. Для этого

мы будем использовать фразу «X в *n* раз быстрее, чем Y» или использовать равноценную фразу «Скорость X в *n* раз больше скорости Y», что означает:

$$\frac{\text{Производительность}_X}{\text{Производительность}_Y} = n.$$

Если X в *n* раз быстрее, чем Y, тогда время выполнения задачи на Y в *n* раз больше, чем на X:

$$\frac{\text{Производительность}_X}{\text{Производительность}_Y} = \frac{\text{Время выполнения}_Y}{\text{Время выполнения}_X} = n.$$

Упражнение

Относительная производительность
Если компьютер A выполняет программу за 10 секунд, а компьютер B выполняет ту же программу за 15 секунд, то насколько A быстрее B?

Ответ

Мы знаем, что A в *n* раз быстрее, чем B, если:

$$\frac{\text{Производительность}_A}{\text{Производительность}_B} = \frac{\text{Время выполнения}_B}{\text{Время выполнения}_A} = n.$$

Таким образом, соотношение производительности равно:

$$\frac{15}{10} = 1{,}5$$

и A поэтому в 1,5 раза быстрее, чем B.

В предыдущем упражнении мы можем также сказать, что компьютер B в 1,5 раза медленнее, чем компьютер A, поскольку

$$\frac{\text{Производительность}_A}{\text{Производительность}_B} = 1{,}5$$

означает, что

$$\frac{\text{Производительность}_A}{1{,}5} = \text{Производительность}_B$$

Чтобы упростить терминологию, при количественном сравнении компьютеров мы обычно будет употреблять термин *быстрее чем*. Поскольку производительность и время выполнения являются обратными величинами, *увеличение* производительности требует уменьшения времени выполнения. Чтобы избежать возможной путаницы между терминами *увеличение* и *уменьшение*, мы обычно будем говорить

«улучшение производительности» или «улучшение времени выполнения», подразумевая при этом «увеличение производительности» и «уменьшение времени выполнения».

Оценка производительности

Оценочным критерием компьютерной производительности является время: компьютер, выполняющий тот же объем работы за меньшее время, является более быстрым. *Время выполнения* программы оценивается в секундах, затраченных на ее выполнение. Но время может быть определено и другими способами, в зависимости от того, что берется в расчет. Наиболее простые — *время выполнения процесса*, *время отклика* или *общее затраченное время*. Эти понятия означают полное время, затраченное на выполнение задачи, включая обращения к диску, обращения к памяти, работу устройств ввода-вывода, издержки, связанные с работой операционной системы, то есть все, на что потрачено время.

Но компьютеры часто работают в режиме разделения времени, и процессор может работать над выполнением сразу нескольких программ. В таких случаях система может стремиться не к минимизации общего времени, затраченного на выполнение одной программы, а к оптимизации пропускной способности. Поэтому часто возникает потребность в проведении черты между общим затраченным временем и временем работы процессора в наших интересах. **Время выполнения задачи центральным процессором** (CPU execution time), или просто **процессорное время** (CPU time), которое учитывает это различие, — это время, которое центральный процессор затрачивает на работу над этой задачей, оно включает время ожидания операций ввода-вывода или время, затраченное на выполнение других программ. (Но при этом следует помнить, что время отклика, ожидаемое пользователем, будет общим затраченным временем, а не процессорным временем.) Процессорное время может и дальше делиться на процессорное время, затраченное на программу и называемое **процессорным временем пользователя** (user CPU time), и на процессорное время, затраченное на выполнение операционной системой задач в интересах программы, называемое **процессорным временем системы** (system CPU time). Провести четкую черту между процессорным временем системы и пользователя довольно трудно, поскольку зачастую трудно отнести активность операционной системы к той или иной пользовательской программе, трудности вызываются также и функциональными особенностями разных операционных систем.

Время выполнения задачи центральным процессором (CPU EXECUTION TIME)
Также называется процессорным временем (CPU time). Фактическое время, затраченное центральным процессором на выполнение конкретной задачи.

Процессорное время пользователя (user CPU time)
Время центрального процессора, затраченное на саму программу.

Процессорное время системы (system CPU time)
Время центрального процессора, затраченное в операционной системе на выполнение задач, связанных с выполнением программы.

Чтобы быть последовательными, мы сохраним различие между производительностью, основанной на общем затраченном времени, и производительностью, основанной на времени

выполнения задачи центральным процессором. Для ссылки на общее затраченное время на разгруженной системе будет использоваться термин *производительность системы*, а для ссылки на процессорное время пользователя будет использоваться термин *производительность центрального процессора*. В этой главе основное внимание будет уделено производительности центрального процессора, хотя наши рассуждения о том, из чего складывается производительность, могут относиться либо к оценкам общего затраченного времени, либо к оценкам процессорного времени.

Представление о производительности программ

Для разных приложений особо важную роль играют разные аспекты производительности компьютерной системы. Многие приложения, особенно те, которые работают на серверах, сильно зависят от производительности устройств ввода-вывода, которая, в свою очередь, зависит как от аппаратного, так и от программного обеспечения. Здесь представляет интерес общее затраченное время, определенное с помощью обычных часов. В некоторых условиях пользователя может интересовать пропускная способность, время отклика или сложная комбинация из этих двух показателей (например, сочетание максимальной пропускной способности с наихудшим временем отклика). Для повышения производительности программы нужно четко определить значение показателя производительности, а затем приступить к поиску возможных узких мест. В следующих главах будет описано, как вести поиск узких мест и повышать производительность в различных частях системы.

Хотя нас как пользователей компьютера интересует время, при исследовании компонентов компьютера удобно думать о производительности, пользуясь другими оценочными критериями. В частности, разработчики компьютеров могут анализировать компьютер, используя показатель, описывающий, насколько быстро аппаратура может выполнять основные функции. Практически все компьютеры используют тактовый генератор, определяющий момент наступления событий. Эти дискретные интервалы времени называются **тактовыми циклами** (или **тактами**, **тактами системных часов**, **тактовыми интервалами**, **тактовыми сигналами**, **циклами**). Конструкторы ссылаются на продолжительность **тактового интервала** и как на время завершения *тактового цикла* (например, 250 пикосекунд, или 250 пс), и как на *тактовую частоту* (например, 4 гигагерц, или 4 ГГц), которая является для тактового цикла обратной величиной. В следующем подразделе мы поговорим о связи между тактовым циклом, который использует разработчик аппаратной части компьютера, и секундами, которые использует пользователь.

Тактовый цикл (clock cycle)
Также называется тактом (tick), тактом системных часов (clock tick), тактовым интервалом (clock period), тактовым сигналом (clock), циклом (cycle). Это время одного тактового интервала, относящееся, как правило, к тактовому генератору процессора, работающему с постоянной частотой.

Тактовый интервал
Продолжительность каждого тактового цикла.

Самопроверка

1. Предположим, что нам известно, что работа приложения, использующего и клиентский настольный компьютер, и удаленный сервер, ограничивается производительностью сети. Скажите, что происходит при улучшении одной лишь пропускной способности? А при улучшении как времени отклика, так и пропускной способности? Или вообще не происходит никаких изменений?
   1. Между клиентом и сервером добавлен еще один сетевой канал, повышающий общую пропускную способность сети и сокращающий время ожидания доступа к сети (поскольку теперь уже имеется два канала).
   2. Улучшено качество сетевого программного обеспечения, благодаря чему сократилась задержка в сетевых линиях связи, но не повысилась пропускная способность сети.
   3. Увеличен объем оперативной памяти компьютера.
2. Компьютер С работает в четыре раза быстрее компьютера В, который выполняет заданное приложение за 28 секунд. Сколько времени займет выполнение этого же приложения на компьютере С?

Производительность центрального процессора и ее факторы

Пользователи и разработчики часто оценивают производительность исходя из разных критериев. Если мы сможем соотнести эти разные критерии, мы сможем определить воздействие конструктивных изменений на производительность в понятиях пользователя. Поскольку на данный момент мы рассматриваем только производительность центрального процессора, основным критерием производительности является время выполнения задачи центральным процессором. Основные показатели процессорного времени (количество тактовых циклов и время цикла тактового генератора) связаны простой формулой:

Время выполнения программы центральным процессором	=	Количество тактовых циклов процессора, затраченных на программу	×	Время цикла тактового генератора

И в другом варианте, поскольку тактовая частота и время цикла тактового генератора являются обратными величинами:

$$\text{Время выполнения программы центральным процессором} = \frac{\text{Количество тактовых циклов процессора, затраченных на программу}}{\text{Тактовая частота}}$$

Эта формула дает понять, что разработчик аппаратуры может повысить производительность, сократив количество тактовых циклов, затрачиваемых на программу, или сократив продолжительность тактового цикла. В следующих главах будет показано, что разработчик часто сталкивается с компромиссом между количеством тактовых циклов, необходимым для программы, и продолжительностью каждого

цикла. Многие технологии, уменьшающие количество тактовых циклов, могут также увеличивать продолжительность тактового цикла.

Упражнение

Повышение производительности

Интересующая нас программа выполняется на компьютере А, имеющем тактовую частоту 2 ГГц, за 10 секунд. Мы стараемся помочь разработчику компьютеров создать компьютер В, который выполнял бы эту программу за 6 секунд. Этот разработчик определил, что соответствующее повышение тактовой частоты вполне допустимо, но это повышение повлияет на всю остальную конструкцию центрального процессора и компьютеру В на выполнение программы потребуется в 1,2 раза больше тактовых циклов, чем компьютеру А. Какую тактовую частоту следует заказать разработчику компьютеров?

Ответ

Сначала давайте определим количество циклов, необходимых программе на компьютере А:

$$\text{Процессорное время}_A = \frac{\text{Количество тактовых циклов процессора}_A}{\text{Тактовая частота}_A}$$

$$10 \text{ секунд} = \frac{\text{Количество тактовых циклов процессора}_A}{2 \times 10^9 \text{ циклов в секунду}}$$

$$\text{Количество тактовых циклов процессора}_A = 10 \text{ секунд} \times 2 \times 10^9 \text{ циклов в секунду} = 20 \times 10^9 \text{ циклов}$$

Процессорное время для компьютера В может быть найдено с использованием следующего уравнения:

$$\text{Процессорное время}_B = \frac{1,2 \times 20 \times \text{Количество тактовых циклов процессора}_A}{\text{Тактовая частота}_B}$$

$$6 \text{ секунд} = \frac{1,2 \times 20 \times 10^9 \text{ циклов}}{\text{Тактовая частота}_B}$$

$$\text{Тактовая частота}_B = \frac{1,2 \times 20 \times 10^9 \text{ циклов}}{6 \text{ секунд}} = 1,2 \times 20 \times 10^9 \text{ циклов в секунду} = 4 \times 10^9 \text{ циклов в секунду} = 4\text{ГГц}$$

Чтобы выполнить программу за 6 секунд, компьютер В должен иметь вдвое большую тактовую частоту, чем компьютер А.

Показанные выше уравнения производительности не учитывают количество инструкций, необходимое для выполнения программ. (Инструкции, из которых составляется программа, рассматриваются в следующей главе.) Но поскольку компилятор сгенерировал вполне определенное количество инструкций, подлежащих выполнению, и компьютер должен выполнить инструкции, чтобы следовать программе, время выполнения должно зависеть от количества инструкций в программе. Один из способов представления времени выполнения состоит в том, что оно равно количеству выполненных инструкций, умноженному на среднее время

выполнения одной инструкции. Поэтому количество тактовых циклов, требуемое для программы, может быть записано в виде следующей формулы:

Количество тактовых циклов процессора	=	Количество инструкций для программы	×	Среднее количество тактовых циклов на инструкцию

Для термина **количество тактовых циклов на инструкцию**, означающего среднее количество тактовых циклов, затрачиваемых на выполнение каждой инструкции, часто используется сокращение **CPI** (clock cycles per instruction). Поскольку на разные инструкции, в зависимости от того, что именно они делают, затрачивается разное количество времени, CPI является усредненным показателем для всех инструкций, выполняемых в программе. CPI предоставляет единственный способ сравнения двух различных реализаций одной и той же архитектуры набора инструкций, поскольку количество инструкций, выполняемых в программе, будет, конечно же, одинаковым.

Количество тактовых циклов на инструкцию (CPI)
Среднее количество тактовых циклов на инструкцию для программы или ее фрагмента.

Упражнение

Использование уравнения производительности
Предположим, что есть две реализации одной и той же архитектуры набора команд. У компьютера A продолжительность тактового цикла равна 250 пс, значение CPI для некой программы равно 2,0, а у компьютера B продолжительность тактового цикла равна 500 пс, значение CPI для той же программы равно 1,2. Какой из компьютеров будет быстрее для данной программы и насколько?

Ответ

Мы знаем, что любой компьютер при каждом запуске программы выполняет одно и то же количество инструкций; обозначим это количество буквой *I*. Сначала определим количество тактовых циклов процессора для каждого компьютера:

$$\text{Количество тактовых циклов процессора}_A = I \times 2{,}0$$

$$\text{Количество тактовых циклов процессора}_B = I \times 1{,}2.$$

Теперь мы можем вычислить процессорное время для каждого компьютера:

$$\text{Процессорное время}_A = \text{Количество тактовых циклов процессора}_A \times \text{продолжительность тактового цикла} = I \times 2{,}0 \times 250 \text{ пс} = 500 \times I \text{пс}.$$

И точно так же для B:

$$\text{Процессорное время}_B \times I \times 1{,}2 \times 500 \text{ пс} = 600\ I \text{пс}.$$

Совершенно очевидно, что компьютер A работает быстрее. Насколько быстрее, позволяет определить соотношение показателей времени выполнения:

$$\frac{\text{Производительность процессора}_A}{\text{Производительность процессора}_B} = \frac{\text{Время выполнения}_B}{\text{Время выполнения}_B} = \frac{600 \times I \text{пс}}{500 \times I \text{пс}} = 1{,}2.$$

Можно сделать вывод, что для данной программы компьютер A в 1,2 раза быстрее компьютера B.

Классическое уравнение производительности центрального процессора

Теперь мы можем описать основное уравнение производительности в понятиях **количества инструкций** (количества инструкций, выполненных по программе), CPI и продолжительности тактового цикла:

$$\text{Процессорное время} = \text{Количество инструкций} \times \text{CPI} \times \text{Продолжительность тактового цикла}$$

или, поскольку тактовая частота является величиной, обратной продолжительности тактового цикла:

$$\text{Процессорное время} = \frac{\text{Количество инструкций} \times \mathit{CPI}}{\text{Тактовая частота}}.$$

Эти формулы особенно удобны тем, что в них использованы три ключевых фактора, влияющих на производительность. Мы можем использовать эти формулы для сравнения двух различных реализаций или для оценки альтернативной конструкции, если знаем, как она влияет на эти три параметра.

Количество инструкций
Количество инструкций, выполненных по программе.

Упражнение

Сравнение кодовых сегментов
Разработчик компилятора пытается принять решение относительно двух кодовых последовательностей для конкретного компьютера. Разработчики аппаратуры предоставили следующие факты:

	CPI для каждого класса инструкций		
	A	B	C
CPI	1	2	3

Для отдельной инструкции на языке высокого уровня создатель компилятора рассматривает две кодовые последовательности, для выполнения которых требуется следующее количество инструкций:

Кодовая последовательность	Количество инструкций для каждого класса инструкций		
	A	B	C
1	2	1	2
2	4	1	1

Для какой кодовой последовательности будет выполнено большее количество инструкций?
Какая из них будет выполняться быстрее?
Каков будет показатель CPI для каждой последовательности?

Для последовательности 1 выполняется 2 + 1 + 2 = 5 инструкций. Для последовательности 2 выполняется 4 + 1 + 1 = 6 инструкций. Стало быть, для последовательности 1 выполняется меньше инструкций.

Для определения общего количества тактовых циклов для каждой последовательности можно применить формулу зависимости количества тактовых циклов процессора от количества инструкций и CPI:

$$\text{Количество тактовых циклов процессоров} = \sum_{i=1}^{n} (CPI_i \times C_i).$$

Она дает нам следующий результат:

$$\text{Количество тактовых циклов процессора}_1 = (2 \times 1) + (1 \times 2) + (2 \times 3) = 2 + 2 + 6 = 10 \text{ циклов}.$$

$$\text{Количество тактовых циклов процессора}_2 = (4 \times 1) + (1 \times 2) + (1 \times 3) = 4 + 2 + 3 = 9 \text{ циклов}.$$

Таким образом, последовательность 2 выполняется быстрее, несмотря даже на то, что для нее выполняется на одну инструкцию больше. Поскольку выполнение кодовой последовательности 2 требует меньше общего количества тактовых циклов, но использует больше инструкций, она должна иметь меньший показатель CPI. Значения CPI могут быть вычислены следующим образом:

$$CPI = \frac{\text{Количество тактовых циклов процессора}}{\text{Количество инструкций}}.$$

$$CPI_1 = \frac{\text{Количество тактовых циклов процессора}_1}{\text{Количество инструкций}_1} = \frac{10}{5} = 2{,}0.$$

$$CPI_2 = \frac{\text{Количество тактовых циклов процессора}_2}{\text{Количество инструкций}_2} = \frac{9}{6} = 1{,}5.$$

Самое важное

В табл. 1.3 приведены основные оценочные показатели, используемые в компьютере на различных уровнях, и единицы их измерения в каждом конкретном случае. Как эти факторы объединяются для получения времени выполнения, измеряемого в секундах на программу, можно увидеть в следующей формуле:

Время = Количество секунд на программу =
= Количество инструкций на программу ×
× Количество тактовых циклов на инструкцию ×
× Количество секунд на тактовый цикл.

Всегда нужно помнить, что единственным полноценным и достоверным оценочным показателем производительности компьютера является время. Например, изменение набора команд, направленное на снижение количества инструкций,

может привести к созданию конструкции с более медленным тактовым циклом или с более высоким показателем CPI, что нивелирует все преимущества от уменьшения количества инструкций. Аналогично этому, поскольку показатель CPI зависит от типа выполняемых инструкций, код, для которого выполняется меньшее количество инструкций, может быть далеко не самым быстрым.

Таблица 1.3. Основные составляющие производительности и способы их оценки

Составляющая производительности	В чем измеряется
Время выполнения программы центральным процессором	В секундах на программу
Количество инструкций	В инструкциях, использованных для выполнения программы
Количество тактовых циклов на инструкцию (CPI)	В среднем количестве тактовых циклов на одну инструкцию
Продолжительность тактового цикла	В секундах на тактовый цикл

Как можно определить значение этих факторов в уравнении производительности? Время выполнения программы центральным процессором можно оценить, запустив программу, а продолжительность тактового цикла обычно приводится в документации компьютера. Труднее, наверное, будет определить количество инструкций и CPI. Разумеется, если известна тактовая частота и время выполнения программы центральным процессором, то для определения всего остального нужно получить лишь количество инструкций или CPI.

Количество инструкций можно определить, используя программные средства, предназначенные для исследования процесса выполнения, или эмулятор архитектуры. Кроме этого, для записи различных оценочных показателей, включая количество выполняемых инструкций, средний показатель CPI, и для определения источников потери производительности можно воспользоваться аппаратными счетчиками, имеющимися у большинства процессоров. Поскольку количество инструкций зависит от архитектуры, но не от ее конкретной реализации, его можно определить и без знания всех подробностей реализации. А вот CPI зависит от разнообразных конструктивных особенностей компьютера, включая и систему памяти, и структуру процессора (в чем мы сможем убедиться, изучая главы 4 и 5), а также и от сочетания типов инструкций, выполняемых по программе (набор инструкций). Таким образом, показатель CPI варьируется от приложения к приложению, и от реализации к реализации одного и того же набора инструкций.

Предыдущий пример показывает всю рискованность оценки производительности с использованием только одного фактора (количества инструкций). При сравнении двух компьютеров нужно принимать во внимание все три компонента, которые обусловливают время выполнения. Если некоторые факторы имеют одинаковое значение, как тактовая частота в предыдущем примере, производительность может быть определена сравнением всех неодинаковых по значению факторов. Поскольку CPI варьируется в зависимости от **набора инструкций**,

Набор инструкций
Оценка динамической частоты появления инструкций в одной или нескольких программах.

должно сравниваться как количество инструкций, так и CPI, даже если используется одна и та же тактовая частота. В конце этой главы приведены упражнения по вычислению последовательности улучшений компьютера и компилятора, влияющих на тактовую частоту, CPI и количество инструкций. В разделе 1.8 будет рассмотрена общая оценка производительности, не учитывающая все условия и поэтому способная ввести вас в заблуждение.

Представление о производительности программ

Производительность программы зависит от алгоритма, языка, компилятора, архитектуры и используемого аппаратного обеспечения. Оценка влияния этих компонентов на факторы, фигурирующие в уравнении производительности центрального процессора, сведена в следующую таблицу:

Аппаратный или программный компонент	На что влияет?	Как влияет?
Алгоритм	На количество инструкций и, возможно, на CPI	Алгоритм определяет количество инструкций, необходимых для выполнения исходной программы, а следовательно, и количество выполняемых процессором инструкций. Алгоритм может также влиять на CPI за счет предпочтения более медленных или более быстрых инструкций. Например, если алгоритм использует больше операций с плавающей точкой, это может привести к более высокому CPI.
Язык программирования	На количество инструкций и на CPI	Язык программирования, естественно, влияет на количество инструкций, поскольку инструкции языка транслируются в инструкции процессора, определяющие общее количество выполняемых инструкций. Язык, в силу своих свойств, может также влиять на CPI: например, язык с мощной поддержкой абстракций данных (к примеру, Java) может потребовать применения опосредованных вызовов, использующих более затратные, с точки зрения CPI, инструкции.
Компилятор	На количество инструкций и на CPI	Эффективность работы компилятора влияет и на количество инструкций, и на среднее количество тактов, приходящееся на одну инструкцию, поскольку компилятор определяет ход трансляции инструкций языка в инструкции процессора. Компилятор может играть весьма сложную роль и осуществлять комплексное воздействие на CPI.
Архитектура набора команд	На количество инструкций, на тактовую частоту и на CPI	Архитектура набора команд воздействует на все три аспекта производительности центрального процессора, поскольку она влияет на инструкции, необходимые для выполняемой функции, на количество циклов, необходимое для выполнения каждой инструкции, и, в общем, на тактовую частоту процессора.

Уточнение. Вопреки возможным ожиданиям того, что минимальный показатель CPI будет равен 1,0, в главе 4 будет показано, что некоторые процессоры извлекают и выполняют за один такт сразу несколько инструкций. Чтобы отразить такой подход, некоторые разработчики перевернули аббревиатуру CPI, чтобы можно было говорить о *IPC*, или о *instruction per clock cycle*, то есть о количестве инструкций, выполняемых за один тактовый цикл. Если процессор выполняет в среднем две инструкции за один тактовый цикл, то его показатель IPC равен двум, а следовательно, его CPI равен 0,5.

Самопроверка

Некое приложение, написанное на языке Java, выполняется на настольном компьютере за 15 секунд. Новый выпуск компилятора Java требует только 0,6 инструкции, генерировавшейся старым компилятором. Но, к сожалению, он повышает показатель CPI в 1,1 раза. Каково ожидаемое ускорение выполнения программы при использовании нового компилятора? Выберите правильный ответ из трех приведенных ниже:

а) $\frac{15 \times 0,6}{1,1} = 8,2$ с;

б) $15 \times 0,6 \times 1,1 = 9,9$ с;

в) $\frac{15 \times 1,1}{0,6} = 27,5$ с.

1.5. Барьер потребляемой мощности

На рис. 1.12 показан рост тактовой частоты и потребляемой мощности восьми поколений микропроцессоров Intel за 25 лет. За два с лишним десятилетия наблюдалось быстрый рост как тактовой частоты, так и потребляемой мощности, который прекратился совсем недавно. Причиной их одновременного роста является взаимозависимость, а причиной замедления этого процесса стало достижение практического предела величины тепловой энергии, которую можно отвести от микропроцессоров.

Доминирующая технология изготовления интегральных микросхем носит название КМОП (Комплементарная логика на транзисторах металл-оксид-полупроводник). Для КМОП основным источником рассеиваемой мощности является так называемая динамическая мощность, то есть мощность, потребляемая во время переключения. Динамическая мощность рассеивания зависит от емкостной нагрузки каждого транзистора, прикладываемого напряжения и частоты переключения транзистора:

$$\text{Мощность} = \text{Емкостная нагрузка} \times \text{Напряжение}^2 \times \text{Частота переключения}$$

Частота переключения зависит от тактовой частоты. Емкостная нагрузка на транзистор зависит от количества транзисторов, соединенных с выходом (так называемым коэффициентом разветвления по выходу), и от технологии, определяющей емкостное сопротивление проводников и транзисторов.

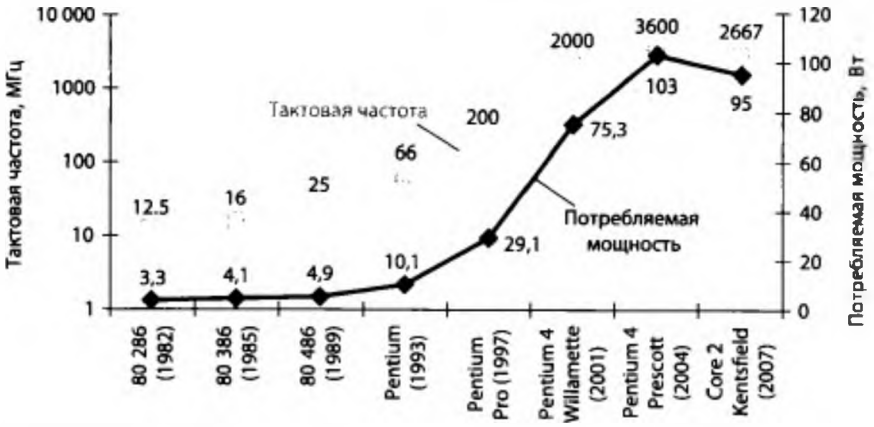


Рис. 1.12. Тактовая частота и потребляемая мощность микропроцессоров семейства Intel x86 за восемь поколений и 25 лет. Значительный скачок в тактовой частоте и потребляемой мощности, не приведший к пропорциональному росту производительности, связан с появлением Pentium 4. Проблемы с температурным режимом работы процессора Prescott привели к отказу от дальнейшего развития линейки Pentium 4. В линейке Core 2 произошел возврат к простому конвейеру с меньшими тактовыми частотами с несколькими процессорами на одном чипе

Как можно было увеличить тактовую частоту в 1000 раз при увеличении потребляемой мощности в 30 раз? Потребляемая мощность может быть снижена за счет снижения напряжения питания, что и происходило с появлением каждого нового поколения технологии, а потребляемая мощность зависит от квадрата напряжения. Обычно напряжение с появлением каждого нового поколения снижалось на 15%. За 20 лет напряжение питания снизилось с 5 до 1 В, поэтому потребляемая мощность возросла лишь в 30 раз.

Упражнение

Относительная потребляемая мощность

Предположим, что идет разработка нового, более простого процессора, имеющего емкостную нагрузку, составляющую 85% от емкостной нагрузки более сложного старого процессора. Далее предположим, что у него имеется регулируемое напряжение питания, позволяющее уменьшить напряжение по сравнению с прежним процессором на 15%, что приводит к 15% снижению частоты. Как это повлияет на динамическую мощность?

Ответ

$$\frac{\text{Мощность}_{\text{н}}}{\text{Мощность}_{\text{ст}}} =$$

$$= \frac{(\text{Емкостная нагрузка} \times 0,85) \times (\text{Напряжение} \times 0,85) \times (\text{Частота переключения} \times 0,85)}{\text{Емкостная нагрузка} \times \text{Напряжение} \times \text{Частота переключения}}$$

Таким образом, соотношение мощностей равно

$$0,85^4 = 0,52$$

Следовательно, новый процессор потребляет почти в два раза меньшую мощность, чем старый.

В настоящее время проблема состоит в том, что дальнейшее понижение напряжения питания настолько повышает утечку тока, что транзисторы становятся похожими на водозапорные краны, которые невозможно закрутить до конца. Даже сегодня около 40% рассеиваемой мощности приходится на утечку. Если ток утечки транзисторов начнет повышаться, то с этим процессом станет трудно справиться.

Пытаясь решить проблему потребляемой мощности, конструкторы уже сейчас подключают громоздкие устройства для усиления отвода тепла и отключают те части микросхемы, которые не задействуются в данном тактовом цикле. Хотя существуют и более затратные способы охлаждения чипов, позволяющие повысить их рассеиваемую мощность, скажем, до 300 Вт, подобные технологии слишком дороги для настольных компьютеров.

Поскольку разработчики компьютеров уперлись в барьер потребляемой мощности, им понадобился новый способ достижения прогресса. И они выбрали другой путь, отличающийся от того, по которому они шли в конструировании микропроцессоров первые 30 лет.

Уточнение. Хотя динамическая мощность и является основным источником рассеиваемой мощности в КМОП, имеет место и статическая рассеиваемая мощность, создаваемая утечкой тока даже у запертого транзистора. Как уже упоминалось, утечка тока в 2008 году составляла 40% рассеиваемой мощности. Таким образом, увеличение количества транзисторов повышает рассеиваемую мощность, даже если транзистор заперт. Для устранения утечки была разработана масса технологий и технологических новшеств, но дальнейшее снижение напряжения питания дается с большим трудом.

1.6. Коренное изменение: переход от одного к нескольким процессорам

> До сих пор большинство программ можно было сравнить с музыкой, написанной для одного исполнителя, но с новым поколением чипов мы приобрели некоторый опыт написания программ для дуэтов, квартетов и других небольших ансамблей, но создание аранжировок для больших оркестров и хоров является совершенно другой разновидностью проблемы.
>
> *Брайан Хьюз*

Ограничение по потребляемой мощности заставило внести в конструирование микропроцессоров существенные изменения. На рис. 1.13 показано улучшение показателя времени отклика программ для микропроцессоров с течением времени. Начиная с 2002 года рост снизился с 1,5 раза в год до менее чем 1,2 раза в год.

Вместо того чтобы продолжать уменьшение времени отклика одной программы, запущенной на одном процессоре, в 2006 году во все компании по производству

настольных компьютеров и серверов были поставлены микропроцессоры содержащие на одном чипе несколько процессоров, преимущество которых чаще всего выражалось в увеличении пропускной способности, чем в уменьшении времени отклика. Чтобы не путаться в словах «процессор» и «микропроцессор», компании стали называть процессоры ядрами, а такие микропроцессоры стали чаще всего называться многоядерными. Следовательно, «четырехъядерный» микропроцессор — это чип, содержащий четыре процессора, или четыре ядра.

В табл. 1.4 показано количество процессоров (ядер), потребляемая мощность и тактовые частоты самых последних микропроцессоров. Согласно официальным документально подтвержденным планам многих компаний, удвоение количества ядер на один микропроцессор для полупроводниковой технологии должно происходить примерно раз в два года (см. главу 7).

В былые годы программисты, не меняя ни одной строки своего кода, могли рассчитывать на то, что новинки аппаратуры, архитектуры и компиляторов будут удваивать производительность их программ каждые 18 месяцев. Сегодня, для того чтобы получить существенное сокращение времени отклика, программистам нужно переписывать свои программы, чтобы воспользоваться преимуществами нескольких процессоров. Кроме того, чтобы ускорить работу программы на новых микропроцессорах, программистам потребуется и дальше улучшать производительность кода.

Чтобы уделить особое внимание тесному взаимодействию программных и аппаратных систем, мы воспользуемся специальным подразделом *Интерфейс аппаратного и программного обеспечения*, и ниже следует первый из таких подразделов.

Интерфейс аппаратного и программного обеспечения

Распараллеливание всегда было важным фактором производительности вычислений, но зачастую его наличие не бросалось в глаза. В главе 4 будет рассмотрена конвейеризация — весьма изящная технология, позволяющая ускорить выполнение программ за счет совмещения выполнения инструкций. Это один из примеров *распараллеливания на уровне инструкций*, где параллельная работа оборудования рассматривается весьма условно, позволяя программисту и компилятору считать, что оборудование осуществляет последовательное выполнение инструкций.

Почему программистам так трудно создавать по-настоящему параллельные программы? Первая причина заключается в том, что параллельное программирование направлено на повышение производительности, что усложняет сам процесс программирования. Нужно, чтобы программа была не только правильной, решала важную задачу и предоставляла удобный интерфейс для человека или для других программ, которые ее вызывают, но еще и чтобы она была быстро выполняемой. В противном случае, если производительность не играет важной роли, будет создаваться последовательно выполняемая программа.

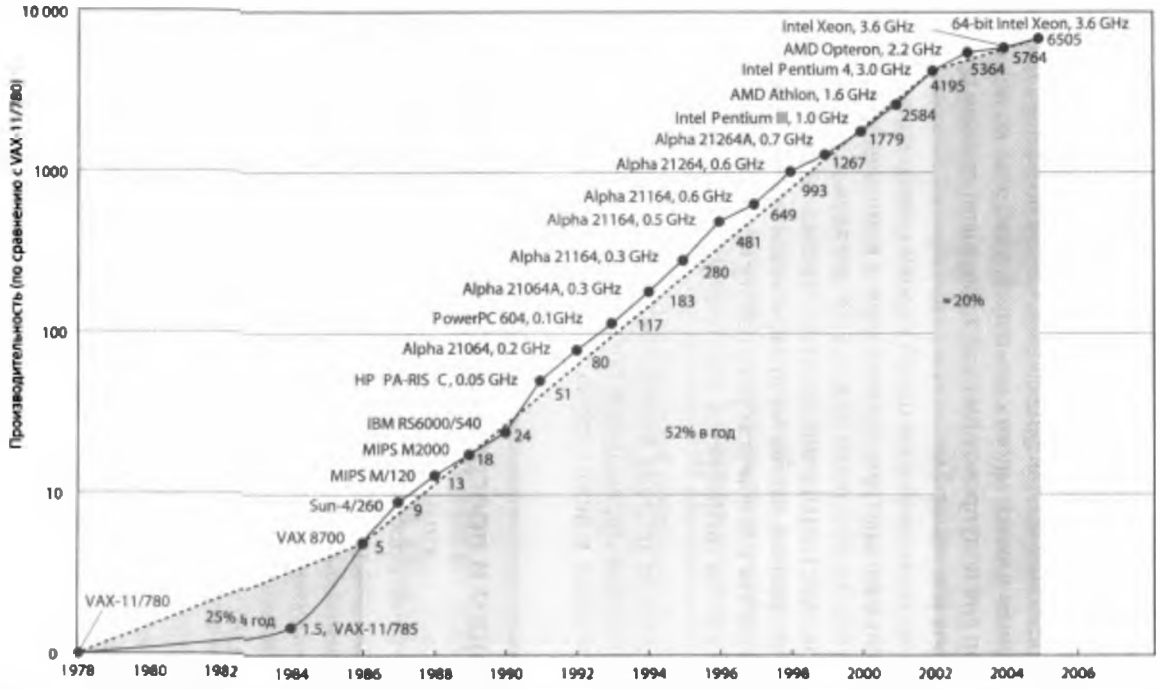


Рис. 1.13. Прирост производительности процессоров с середины 1980-х годов. В этом графике за точку отсчета взята производительность компьютера VAX 11/780, оцененная с помощью контрольных задач SPECint (см. раздел 1.8). До середины 1980-х годов прирост производительности процессоров зависел в основном от совершенствования технологий и составлял в среднем 25% в год. Увеличение с тех времен прироста производительности до почти 52% можно отнести на счет более совершенных архитектурных и организационных находок. До 2002 года этот годовой прирост производительности примерно был равен семи условным единицам. Производительность вычислений, ориентированных на работу с числами с плавающей точкой, увеличивалась еще быстрее. Начиная с 2002 года ограничения, связанные с потребляемой мощностью, доступностью распараллеливания на уровне инструкций и большого времени ожидания при обращении к памяти, привели к замедлению прироста производительности монопроцессора до примерно 20% в год

Таблица 1.4. Количество ядер, тактовая частота и потребляемая мощность многоядерных микропроцессоров выпуска 2008 года

Показатель	AMD Opteron X4 (Barcelona)	Intel Nehalem	IBM Power 6	Sun Ultra SPARC T2 (Niagara 2)
Ядер на чип	4	4	2	8
Тактовая частота	2,5 ГГц	~ 2,5 ГГц (?)	4,7 ГГц	1,4 ГГц
Мощность, потребляемая микропроцессором	120 Вт	~ 100 Вт (?)	~ 100 Вт ?	94 Вт

Вторая причина заключается в том, что быстродействие для параллельного оборудования означает, что программист должен разбить приложение на части таким образом, чтобы каждый процессор имел примерно одинаковый объем работы за один и тот же период времени и чтобы издержки на диспетчеризацию и координацию не скрадывали рост производительности, достигаемый за счет распараллеливания.

В качестве аналогии предположим, что задача заключалась в написании газетной статьи. Восемь репортеров, работающих над одной и той же статьей, потенциально могут написать статью в восемь раз быстрее. Для достижения такого увеличения скорости кому-то нужно разбить задачу таким образом, чтобы у каждого репортера было чем заняться в один и тот же период времени. Таким образом, возникает необходимость в диспетчеризации подзадач. Если что-нибудь пойдет не так и работа только у одного репортера займет времени больше, чем у семи остальных, преимущества от привлечения восьми репортеров будут менее выраженными. Поэтому для достижения желаемого ускорения необходима сбалансированность нагрузки. Другая опасность появится в том случае, если репортеры для написания своих разделов будут тратить слишком много времени на разговоры друг с другом. Неудача может постигнуть вас и в том случае, когда одна часть статьи, например заключение, не может быть написана, пока не будут завершены все остальные части. Поэтому следует обратить внимание на сокращение издержек на общение и синхронизацию. Как для этой аналогии, так и для параллельного программирования трудности заключаются в диспетчеризации, сбалансированности нагрузки, времени, затрачиваемом на синхронизацию, и в издержках на обмен данными. Как вы уже, наверное, догадались, эти трудности возрастают с ростом числа репортеров, привлеченных для написания газетной статьи, и с ростом числа процессоров, используемых в параллельном программировании.

Для отображения этих коренных изменений, происшедших в компьютерной отрасли, в следующих пяти главах данного издания книги будет выделено по одному разделу, посвященному влиянию параллельной революции на рассматриваемые в главе вопросы:

- *Глава 2, раздел 2.11. Распараллеливание и инструкции: синхронизация.* Обычно независимые параллельные задачи нуждаются в координации по времени, например в сообщении о том, что они завершили свою работу. В этом разделе рассматриваются инструкции, используемые многоядерными процессорами для синхронизации задач.

- *Глава 3, раздел 3.6. Распараллеливание и компьютерная арифметика: ассоциативность.* Зачастую программисты, занимающиеся параллельным программированием, берут за основу последовательную программу. Вполне естественным будет следующий вопрос о работоспособности их параллельной версии: «А выдает ли она точно такой же результат?». Если нет, то вполне логично сделать вывод, что в новую версию вкрались ошибки. Эта логика основана на том, что компьютерная арифметика обладает свойством ассоциативности: вы получаете одинаковую сумму при сложении миллиона чисел, независимо от того, в каком порядке это делается. В этом разделе объясняется, что, несмотря на справедливость данной логики для целых чисел, она не соблюдается для чисел с плавающей точкой.
- *Глава 4, раздел 4.10. Параллельность и современное распараллеливание на уровне инструкций.* С учетом сложностей, связанных с параллельным программированием, в 1990-х годах были приложены огромные усилия для создания оборудования и компиляторов, проявляющих свойства неявного распараллеливания. В этом разделе рассматриваются некоторые из этих смелых технологий, включая одновременное извлечение и выполнение сразу нескольких инструкций и выстраивание предположений относительно результатов решений и выполнение инструкций на основе выдвинутых гипотез.
- *Глава 5, раздел 5.8. Распараллеливание и иерархии памяти: согласованность в использовании кэш-памяти.* Один из путей снижения издержек на обмен данными заключается в использовании всеми процессорами одного и того же пространства памяти, чтобы любой процессор мог читать или записывать любые данные. Учитывая, что все процессоры сегодня используют для временного хранения копий данных кэш-память на быстродействующих элементах, нетрудно представить, что параллельное программирование станет еще сложнее, если в блоках кэш-памяти, связанных с каждым процессором, будут находится противоречивые значения общих данных. В этом разделе рассматриваются механизмы, поддерживающие данные во всех блоках кэш-памяти в согласованном состоянии.
- *Глава 6, раздел 6.9. Распараллеливание и ввод-вывод данных: массивы независимых жестких дисков с избыточностью информации (RAID).* Если в этой параллельной революции проигнорировать ввод и вывод данных, могут проявиться неучтенные последствия параллельного программирования, заставляющие ваши параллельные программы тратить основную часть своего времени на ожидание ввода или вывода данных. В этой главе рассматриваются RAID-системы, технология повышения производительности доступа к средствам хранения информации. RAID-системы показывают еще одно из преимуществ параллельных вычислений: при наличии многочисленных копий информационных ресурсов система способна продолжать обслуживание, несмотря на отказ одного из них. Следовательно, RAID-система может повысить как производительность, так и доступность ввода-вывода.

Кроме этих разделов параллельной обработке данных посвящена вся седьмая глава, в которой более подробно рассматриваются сложности параллельного программирования, представляются два отличающихся друг от друга подхода

обмену данными: общее адресное пространство и явный обмен сообщениями, рассматриваются ограниченные модели параллельных вычислений, которые проще программировать, рассматриваются трудности оценочного тестирования параллельных процессоров, представляются новые простые модели производительности для многоядерных процессоров, и, наконец, рассматриваются и оцениваются четыре примера многоядерных микропроцессоров, использующих эту модель.

1.7. Реальное оборудование: производство и оценочное тестирование AMD Opteron X4

> Я думал, что [компьютеры] найдут такое же универсальное применение, как и книги. Но я не думал, что это произойдет настолько быстро, поскольку не предполагал, что на одном чипе будет такое огромное количество элементов, сколько в итоге на нем уместилось. Неожиданно появился транзистор, и все произошло намного быстрее, чем мы ожидали.
>
> *Джон Преспер Эккерт, один из изобретателей компьютера ENIAC, высказывание 1991 года*

В каждой главе есть раздел с названием «Реальное оборудование», который связывает понятия, рассматриваемые в данной книге, с компьютером, который может использоваться в повседневной работе. В этих разделах рассматриваются технологии, положенные в основу современных компьютеров. В этом первом разделе «Реальное оборудование» мы рассмотрим производство интегральных микросхем и оценку производительности и потребляемой мощности на примере процессора AMD Opteron X4.

Начнем с исходных материалов. Производство чипов начинается с **кремния**. Поскольку кремний не является хорошим проводником электрического тока, его называют **полупроводником**. С помощью специального химического процесса кремний можно трансформировать:

- в отличные проводники электрического тока (при помощи использования микроскопических медных или алюминиевых проводков);
- в отличные изоляторы, не пропускающие электрический ток (подобные пластмассовым или стеклянным изоляторам);
- в блоки, которые при определенных условиях являются либо проводниками, либо изоляторами (как переключатели)

Кремний
Природный элемент, являющийся полупроводником.

Полупроводник
Вещество, не являющееся хорошим проводником электрического тока.

Транзисторы относятся к последней категории. Сверхбольшие интегральные схемы (СБИС, или VLSI), состоят из миллиардов комбинаций проводников, изоляторов и переключателей, изготовляемых в одном небольшом корпусе.

Процесс производства интегральных микросхем существенным образом сказывается на стоимости чипов, следовательно, он важен и для разработчиков компьютеров. Этот процесс показан на рис. 1.14. Он начинается с **выращенного кристалла кремния**, похожего на огромную сосиску. Современные выращенные кристаллы имеют размеры 8–12 дюймов (20–30 см) в диаметре и 12–24 дюйма (30–60 см) в длину. Кристалл в конечном итоге разрезается на тонкие пластины, так называемые **вафли**, толщиной не более 0,1 дюйма (2,54 мм). Затем эти вафли проходят несколько этапов обработки, во время которых на каждую вафлю накладываются шаблоны, создающие транзисторы, проводники и изоляторы. Сегодняшние интегральные микросхемы содержат только один слой транзисторов, но могут иметь от одного до восьми слоев металлических проводников, разделенных слоями изоляторов.

Единственный микроскопический изъян в самой вафле или сбой в проведении одного из нескольких десятков этапов обработки может привести к выходу из строя участка вафли. Эти так называемые **дефекты** практически исключают возможность производства идеальной вафли. Чтобы справиться с дефектом, использовалось несколько приемов, но самый простой из них заключался в помещении на одну вафлю множества независимых компонентов. Обработанная вафля затем нарезалась на эти компоненты, так называемые заготовки под интегральную схему, или **пластины**, у которых было еще менее формальное название **чипы**. На рис. 1.15 показана фотография вафли, содержащей микропроцессоры до их нарезки, а ранее, на рис. 1.9, была показана отдельная микропроцессорная пластина и ее основные компоненты.

Выращенный кристалл кремния
Прут, состоящий из кремниевого кристалла 20–30 см в диаметре и 30–60 см в длину.

Вафля
Пластина из кристалла кремния, не превышающая по толщине 2,54 мм, используемая для изготовления чипов.

Дефект
Микроскопический изъян на вафле или сбой при проведении этапов обработки, который может привести к отказу пластины, содержащей этот дефект.

Пластина
Под пластинами понимаются отдельные прямоугольные секции, нарезанные из вафли, которые менее формально известны как чипы.

Выход годных изделий
Процентное отношение годных пластин к общему количеству пластин на вафле.

Нарезка позволяла отбраковать не всю вафлю, а только пластины с изъянами. Этот подход количественно определялся **выходом годных изделий**, который измерялся в процентном отношении годных пластин к общему числу пластин на вафле.

С увеличением размеров пластин быстро росла и стоимость интегральных микросхем, что обуславливалось как меньшим выходом годных изделий, так и меньшим количеством пластин, помещавшихся на вафле. Чтобы снизить стоимость, крупные пластины «сужались» за счет более миниатюрных размеров как транзисторов, так и проводников. Это улучшало показатель выхода готовых изделий и увеличивало количество пластин на одной вафле.

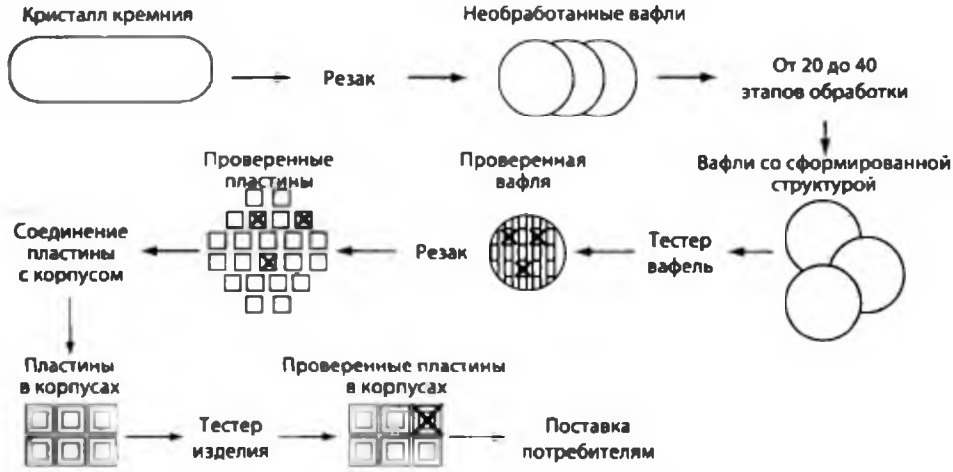


Рис. 1.14. Процесс производства чипов. После нарезки из кремниевого кристалла необработанные вафли проходят от 20 до 40 этапов для получения вафель со сформированной структурой (см. рис. 1.15). Вафли с нанесенной структурой проверяются тестером и на них помечаются годные области. Затем вафля разрезается на пластины (см. рис. 1.15). На этом рисунке из одной вафли получается 20 пластин, 17 из которых проходят проверку. (Крестик означает негодную пластину.) Выход годных пластин в данном случае равен 17/20, или 85%. Затем годные пластины соединяются с корпусами и проверяются еще раз перед поставкой изделий в корпусах потребителям. Эта финальная проверка выявила наличие одного плохого изделия

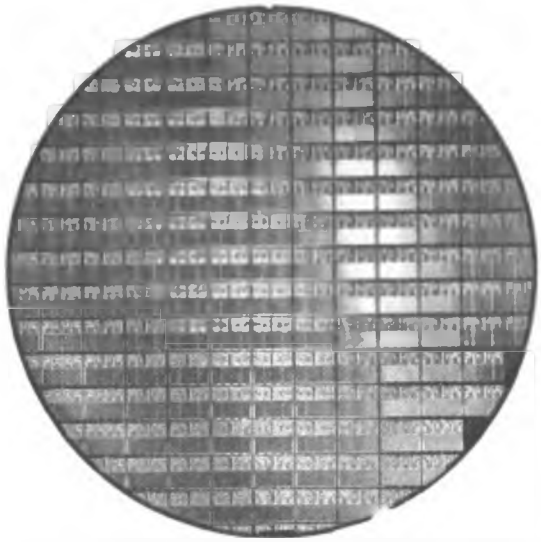


Рис. 1.15. 12-дюймовая (300 мм) вафля, состоящая из чипов AMD Opteron X2, предшественников чипов Opteron X4 (Courtesy AMD). Количество пластин на вафлю при 100% выходе годной продукции равно 117. Несколько десятков частично закругленных чипов на границе вафли являются негодными; их появление обусловлено тем, что проще создавать прямоугольные маски для обработки кремния. Для этой пластины используется 90-нанометровая технология, то есть самые маленькие транзисторы имеют размер примерно 90 нм (настоящие транзисторы меньше, чем на этой фотографии)

В процессе *сборки* годные пластины соединяются с входными и выходными выводами корпуса. Готовые изделия проходят окончательную проверку, поскольку в процессе заключения в корпус могут допускаться ошибки, а затем поставляются потребителям.

Как ранее упоминалось, все более важным конструктивным ограничителем является потребляемая мощность. Проблема мощности имеет два аспекта. Во-первых, мощность должна быть подведена к чипу и распределена по нему; современные микропроцессоры имеют сотни контактов только для питания и заземления! Аналогично этому, многие уровни внутренних соединений используются исключительно для распределения питания и заземления по блокам чипа. Во-вторых, мощность рассеивается в виде тепла и должна быть отведена. В 2008 году микропроцессор AMD Opteron X4 модель 2356 2,0 ГГц выделял 120 ватт, которые должны были быть отведены от чипа, площадью немногим больше 1 см^2!

Уточнение. Стоимость интегральной микросхемы может быть выражена тремя простыми уравнениями:

$$\text{Стоимость одной пластины} = \frac{\text{Стоимость вафли}}{\text{Количество пластин на вафле} \times \text{Коэффициент выхода}}$$

$$\text{Количество пластин на вафле} = \frac{\text{Площадь вафли}}{\text{Площадь пластины}}$$

$$\text{Выход} = \frac{1}{(1 + (\text{Дефектов на площадь} \times \text{Площадь пластины}/2))^2}$$

Первое уравнение вывести несложно. Второе является приблизительным, поскольку в нем не учтена площадь рядом с границей круглой вафли, на которой не помещаются прямоугольные пластины (см. рис. 1.15). Последнее уравнение основано на эмпирических предположениях, касающихся коэффициента выхода готовой продукции на предприятиях по изготовлению интегральных микросхем, использует коэффициент, связанный с числом важных этапов обработки.

Следовательно, в зависимости от уровня дефектности и размера пластины и вафли стоимость обычно не имеет линейной зависимости от площади пластины.

SPEC — контрольные задачи для оценки производительности центральных процессоров

Пользователи компьютеров, изо дня в день запускающие одни и те же программы, могут стать идеальными кандидатурами для оценки нового компьютера. Набор запускаемых программ сформирует **рабочую нагрузку**. Для оценки двух компьютерных систем пользователю нужно будет просто сравнить время выполнения программ, входящих в рабочую нагрузку на этих двух компьютерах. Но для большинства пользователей подобная ситуация нетипична. Они вынуждены полагаться на другие методы оценки производительности того или иного компьютера, надеясь на то, что эти методы дадут реальную картину того, насколько хорошо этот компьютер справится с типичной для пользователя рабочей нагрузкой. Этот альтернативный

метод обычно связан с тем, что при оценке компьютера используется набор **контрольных задач** — программ, специально подобранных для оценки производительности. Контрольные задачи формируют рабочую нагрузку, которая, в соответствии с надеждами пользователя, сможет предугадать реальную рабочую нагрузку.

Для создания стандартного набора контрольных задач для современных компьютерных систем многими продавцами компьютеров финансируется и поддерживается объединенная группа по оценке производительности вычислительных систем — SPEC (System Performance Evaluation Cooperative). В 1989 году SPEC начала свою деятельность с того, что создала набор контрольных задач, предназначенных для оценки производительности процессоров (который сейчас называется SPEC89), развитие которого прошло через пять поколений. Набор, относящийся к самому последнему поколению — SPEC CPU2006, состоит из 12 контрольных задач по работе с целыми числами (CINT2006) и 17 контрольных задач по работе с числами с плавающей точкой (CFP2006). Контрольные задачи по работе с целыми числами варьируются от частей компилятора языка C, шахматных программ и до имитатора квантового компьютера. Контрольные задачи по работе с числами с плавающей точкой включают коды регулярных сеток для моделирования методом конечных элементов, коды метода частиц для задач молекулярной динамики, и редкие коды линейной алгебры для решения задач гидрогазодинамики.

В табл. 1.4 дано описание объединенных контрольных задач SPEC, и времени их выполнения на процессоре Opteron X4, а также показаны множители, определяющие это время выполнения: количество инструкций, CPI и время тактового цикла. Обратите внимание на то, что показатели CPI различаются в 13 раз.

Чтобы упростить продажу компьютеров, группа SPEC решила выдавать отчет в виде одного числа, являющегося результатом выполнения всех 12 контрольных задач по работе с целыми числами. Оценки времени выполнения сначала нормализуются путем деления времени выполнения на эталонном процессоре на время выполнения оцениваемого компьютера; эта нормализация дает оценку, называемую *SPECratio*, преимущество которой в том, что более высокие числовые результаты свидетельствуют о более высокой производительности (то есть, SPECratio является величиной, обратной времени выполнения). Показатель CINT2006 или CFP2006 является суммарной оценкой, полученной путем извлечения среднего геометрического показателей SPECratio.

Уточнение. Когда два компьютера сравниваются при помощи показателей SPECratio, используется среднее геометрическое, поскольку оно дает одни и те же относительные ответы, независимо от того, какой компьютер используется для нормализации результатов. Если бы усреднение значений нормализованных показателей времени выполнения проводилось с использованием среднего арифметического, результаты различались бы в зависимости от того компьютера, который выбирался бы в качестве эталона.

Рабочая нагрузка
Набор программ, запускаемых на компьютере, который представлен либо существующей коллекцией приложений, используемых пользователем, либо составлен из реальных программ, примерно соответствующих такой коллекции. В типичной рабочей нагрузке определяются как сами программы, так и частота их использования.

Контрольная задача (BENCHMARK)
Программа, отобранная для использования при сравнении производительности компьютеров.

Таблица 1.5. Контрольные задачи SPECINTC2006, запущенные на AMD Opteron X4 модель 2356 (Barcelona). В соответствии с классическим уравнением производительности центрального процессора, представленным в разделе 1.4, время выполнение является произведением трех факторов: количества инструкций, количества тактовых циклов на инструкцию (CPI) и времени тактового цикла в наносекундах. SPECratio является всего лишь эталонным временем, поделенным на измеренное время выполнения. Единое число, указанное в качестве SPECINTC2006, является геометрическим средним показателей SPECratio. В таблице 5.13 показывается, что mcf, libquantum, omnetpp и xalancbmk имеют относительно высокие показатели CPI, потому что у них высокие коэффициенты отсутствия нужных данных в кэше

Описание	Название	Количество инструкций $\times 10^9$	CPI	Время тактового цикла ($c \times 10^{-9}$)	Время выполнения (c)	Время обращения (c)	SPECratio
Обработка интерпретируемых строк	perl	2,118	0,75	0,4	637	9,770	15,3
Сжатие с сортировкой блоков	bzip2	2,389	0,85	0,4	817	9,650	11,8
Работа компилятора GNU C	gcc	1,050	1,72	0,4	724	8,050	11,1
Комбинаторная оптимизация	mcf	336	10,0	0,4	1,345	9,120	6,8
Игра GO (искусственный интеллект)	go	1,658	1,09	0,4	721	10,490	14,6
Поиск цепочки генов	hmmer	2,783	0,96	0,4	890	9,330	10,5
Игра шахматы (искусственный интеллект)	sjeng	2,176	0,80	0,4	837	12,100	14,5
Эмуляция квантового компьютера	libquantum	1,623	1,61	0,4	1,047	20,720	19,8
Сжатие видео	h264avc	3,102	0,80	0,4	993	22,130	22,3
Работа с библиотекой имитатора дискретных событий	omnetpp	587	2,94	0,4	690	6,250	9,1
Игры-лабиринты	astar	1,082	1,79	0,4	773	7,020	9,1
XML-парсинг	xalancbmk	1,058	2,70	0,4	1,143	6,900	6,0
Среднее геометрическое							11,7

Формула для вычисления среднего геометрического:

$$\sqrt[n]{\prod_{i=1}^{n} \text{Соотношение времени выполнения}_i},$$

где *Соотношение времени выполнения*$_i$ является временем выполнения, нормализованным по отношению к эталонному компьютеру, для i-той программы из общей рабочей нагрузки, состоящей из n программ, а означает результат $a_1 \times a_2 \times \ldots \times a_n$.

Контрольная задача SPEC Power

Сегодня группа SPEC предлагает более десятка различных наборов контрольных задач, разработанных для тестирования разнообразных компьютерных сред с использованием реальных приложений и четко определенных правил выполнения и требований к отчетам. Самой последней оценкой является SPECpower. Она сообщает о потребляемой мощности серверов за определенный период времени на различных уровнях рабочей нагрузки, разбитой на 10% приращения. В табл. 1.6 показаны результаты для сервера, использующего микропроцессор Barcelona.

Таблица 1.6. Контрольная задача SPECpower_ssj2008, запущенная на компьютере, имеющем два процессора 2,3 ГГц AMD Opteron X4 2356 (Barcelona) с 16 Гбайт DDR2-667 DRAM и одним жестким диском емкостью 500 Гбайт

Номинальная нагрузка (%)	Производительность (ssj_ops)	Средняя потребляемая мощность (Вт)
100%	231 867	295
90%	211 282	286
80%	185 803	275
70%	163 427	265
60%	140 160	256
50%	118 324	246
40%	92 035	233
30%	70 500	222
20%	47 126	206
10%	23 066	180
0%	0	141
Общая сумма	1 283 590	2605
Σ ssj_ops / Σ мощности =		493

Контрольная задача SPECpower начинается с контрольной задачи SPEC для бизнес-приложений, написанных на Java (SPECJBB2005), с помощью которой оцениваются процессоры, блоки кэш- и оперативной памяти, а также виртуальная

машина Java, компилятор, сборщик мусора и части операционной системы. Производительность оценивается в пропускной способности и в таких единицах, как количество бизнес-операций в секунду. И опять, чтобы упростить продажу компьютеров, SPEC свела все эти числовые показатели в один, называемый «Общее количество операций ssj_ops на ватт». Для расчета этого итогового показателя используется следующая формула:

$$\text{Общее количество } ssj_ops_i \text{ на ватт} = \left(\sum_{i=0}^{10} ssj_ops_i\right) / \left(\sum_{i=0}^{10} \text{мощность}_i\right),$$

где ssj_ops_i — это производительность при каждом 10% приращении, а $мощность_i$ — это мощность, рассеиваемая на каждом уровне производительности.

Самопроверка

Основным фактором в определении стоимости интегральной микросхемы является объем выпуска. Какими нижеперечисленными причинами объясняется, почему чип, выпущенный более крупной партией, должен стоить дешевле?

1. При больших объемах выпуска производственный процесс может быть настроен на конкретную конструкцию, увеличивая процент выхода годной продукции.
2. На крупную партию приходится меньший объем конструкторских работ, чем на мелкую.
3. Маски, используемые для производства чипов, стоят дорого, поэтому их стоимость, приходящаяся на один чип, при больших объемах выпуска снижается.
4. Стоимость конструкторских разработок высока и сильно зависит от объема производства, поэтому стоимость одной пластины с производством больших партий изделий снижается.
5. Пластины крупносерийных партий обычно меньше по размеру, чем пластины мелкосерийных партий, и поэтому у них более высокий выход годной продукции на вафлю.

1.8. Заблуждения и недоразумения

> Наука должна начинаться с вымыслов и с критики этих вымыслов.
>
> *Сэр Карл Поппер. «Философия науки»*

Цель раздела, посвященного заблуждениям и недоразумениям, которые будут возникать при прочтении каждой главы, — объяснить суть некоторых часто встречающихся ложных представлений, которые могут складываться и в вашем сознании. Такие ложные представления мы называем *заблуждениями*. При рассмотрении того или иного заблуждения мы будем стараться приводить контраргументы. Мы также рассматриваем *недоразумения*, или легко допускаемые ошибки. Часто недоразумения возникают вследствие обобщения принципов, которые пригодны только для вполне определенных ситуаций. Цель таких разделов — помочь вам избежать

подобных ошибок в отношении тех компьютеров, которые вы, возможно, будете разрабатывать или использовать. Заблуждений и недоразумений, касающихся стоимости и производительности, не удалось избежать многим разработчикам компьютеров, не исключая и нас самих. Поэтому в данном разделе не будет недостатка в соответствующих примерах. Начнем с заблуждения, которое было свойственно многим конструкторам и которое раскрыло довольно важную взаимосвязь в проектировании компьютеров.

Заблуждение. *Надежда на то, что улучшение одной из составных частей компьютера должно пропорционально степени этого улучшения привести к повышению общей производительности.*

Данное заблуждение было свойственно как разработчикам оборудования, так и разработчикам программного обеспечения. Это может быть достаточно хорошо проиллюстрировано с помощью простой конструкторской задачи. Предположим, что из 100 секунд работы программы 80 приходится на операции умножения. Насколько нужно увеличить скорость выполнения операций умножения, чтобы программа работала в пять раз быстрее?

Значение времени выполнения программы после внесения улучшения можно получить с помощью простого уравнения, известного как **закон Амдала**:

```
Время выполнения после улучшения =
Время выполнения, на которое влияет улучшение / Степень улучшения +
Время выполнения, на которое не влияет улучшение
```

Применительно к нашей задаче:

```
Время выполнения после улучшения = 80 с / n + (100 - 80 сек)
```

Поскольку нам нужно, чтобы производительность выросла в пять раз, новое время выполнения должно быть равно 20 с, то есть:

```
20 с = 80 с / n + 20 с
     = 80 с / n
```

Откуда следует, что *нет такой степени* улучшения операции умножения, которая позволила бы добиться пятикратного повышения производительности, если на операции умножения приходится в целом 80% рабочей нагрузки.

Повышение производительности, которого можно достичь с заданным улучшением, ограничено объемом той части задачи, для которой применяется улучшенное свойство. Это положение также приводит к тому, что в повседневной жизни называется законом убывающей отдачи.

Закон Амдала может применяться для оценки повышения производительности, когда известно время, затрачиваемое какой-нибудь функцией, и ее потенциальное ускорение. Наряду с уравнением производительности центрального процессора закон Амдала является удобным средством для оценки потенциальных улучшений. Более подробно закон Амдала будет исследован в упражнениях.

Закон Амдала
Правило, утверждающее, что повышение производительности, возможное при заданном улучшении, ограничено тем объемом задачи, для которого применяется улучшенное свойство. Это правило является количественным вариантом закона убывающей отдачи.

При разработке оборудования большое распространение получило следствие, выведенное из закона Амдала: нужно повышать скорость

выполнения часто встречающихся задач. Эта простая установка напоминает о том, что во многих случаях частота наступления одних событий намного выше частоты наступления других событий. Закон Амдала подсказывает, что возможность усовершенствования зависит от того, сколько времени занимает то или иное событие. Таким образом, ускорение выполнения весьма распространенной задачи будет влиять на повышение производительности больше, чем оптимизация выполнения какой-нибудь довольно редкой задачи. Как это ни парадоксально, но распространенная задача зачастую проще редкой задачи, и, следовательно, ее решение проще усовершенствовать.

Закон Амдала также используется в качестве аргумента для практических ограничений на число параллельных процессоров. Этот аргумент будет изучен в разделе «Заблуждения и недоразумения» главы 7.

Заблуждение. *Компьютеры при низкой загруженности потребляют незначительную мощность.*

Вопрос эффективности энергопотребления при небольшой загруженности вызывает интерес, поскольку рабочая нагрузка серверов не бывает постоянной. К примеру, загруженность центральных процессоров серверов компании Google основную часть времени составляет от 10 до 50 %, а на 100% загруженность приходится менее 1% общего времени. В табл. 1.7 показана потребляемая мощность для серверов, продемонстрировавших лучшие результаты выполнения контрольных задач SPECpower при 100% загруженности, 50% загруженности, 10% загруженности и при простое. Даже те серверы, которые были загружены всего на 10%, потребляли около двух третей своей пиковой мощности.

Поскольку рабочая нагрузка серверов является величиной непостоянной, но при этом они потребляют довольно большую долю от своей пиковой мощности, Луис Баррозо (Luiz Barroso) и Урс Хёлцле (Urs Hölzle) [2007] показывают, что мы должны перепроектировать оборудование на достижение «энергопропорционального вычисления». Если будущие серверы станут использовать, скажем, 10% пиковой мощности при 10% рабочей нагрузке, мы сможем уменьшить сумму счета за электричество центров обработки данных и посодействовать снижению выбросов углекислого газа в атмосферу.

Заблуждение. *Использование для оценки производительности подгруппы уравнений производительности.*

Мы уже рассматривали суть заблуждения относительно предсказания производительности на основе учета чего-либо одного — тактовой частоты, количества инструкций и CPI. Другой весьма распространенной ошибкой является использование для сравнения производительности только двух из трех факторов. Хотя при вполне определенных условиях использование только двух факторов может быть оправданным, все же в применении этого принципа оценки довольно легко ошибиться. Действительно, все предложенные варианты вместо использования времени в качестве оценочного показателя производительности в конечном счете привели к непонятным утверждениям, искажению результатов или неправильной их интерпретации.

Таблица 1.7. Результаты выполнения контрольной задачи SPECPower для трех серверов с наилучшим суммарным показателем ssj_ops на ватт по состоянию на четвертый квартал 2007 года. Суммарный показатель ssj_ops на ватт трех серверов составляет 698, 682 и 667 соответственно. Объем памяти двух верхних серверов составляет 16 Гбайт, а нижнего сервера — 8 Гбайт

Производитель сервера	Микропроцессор	Всего ядер/гнезд	Тактовая частота	Пиковая производительность (ssj_ops)	Потребление при 100% загрузке	Потребление при 50% загрузке	Потребление при 50% загрузке / 100%	Потребление при 10% загрузке	Потребление при 10% загрузке / 100%	Потребление при активном простое	Потребление при активном простое / 100%
HP	Xeon E5440	8/2	3,0 Гц	308,022	269 W	227 W	84%	174 W	65%	160 W	59%
Dell	Xeon E5440	8/2	2,8 Гц	305,413	276 W	230 W	83%	173 W	63%	157 W	57%
Fujitsu Seimens	Xeon E3220	4/1	2,4 Гц	143,742	132 W	110 W	83%	85 W	65%	80 W	60%

Альтернативой показателю времени служит показатель **MIPS** (million instructions per second — **миллион инструкций в секунду**). Для отдельно взятой программы MIPS вычисляется просто:

$$MIPS = \frac{\text{Количество инструкций}}{\text{Время выполнения} \times 10^6}.$$

Поскольку MIPS является показателем выполнения инструкций, он определяет производительность в величинах, обратных времени выполнения: чем быстрее компьютер, тем выше его показатель MIPS. Этот показатель хорош тем, что его несложно понять, и у более быстрых компьютеров предполагаются более высокие показатели, что сообразуется с нашими интуитивными представлениями.

При использовании MIPS в качестве оценочного показателя при сравнении компьютеров возникают три проблемы. Во-первых, MIPS определяется как показатель выполнения инструкций, но при этом характеристики самих инструкций в расчет не берутся. Используя MIPS, невозможно сравнивать компьютеры с различным набором инструкций, поскольку количество инструкций будет заведомо разным. Во-вторых, значение MIPS на одном и том же компьютере от программы к программе изменяется, поэтому компьютер не может постоянно иметь один и тот же показатель MIPS. Например, если подставить вместо времени выполнения формулу его вычисления, мы увидим, как MIPS зависит от тактовой частоты и CPI:

$$MIPS = \frac{\text{Количество инструкций}}{\frac{\text{Количество инструкций} \times \text{CPI}}{\text{Тактовая частота}} \times 10^6} = \frac{\text{Тактовая частота}}{CPI \times 10^6}.$$

Вспомним, что на Opteron X4 при выполнении контрольной задачи SPEC2006 значение CPI увеличивается в 13 раз. И наконец, что самое важное, если в новой программе выполняется больше инструкций, но выполнение каждой из них происходит быстрее, MIPS может измениться независимо от изменения производительности!

Самопроверка

Оценочный показатель	Компьютер A	Компьютер B
Количество инструкций	10 миллиардов	8 миллиардов
Тактовая частота	4 ГГц	4 ГГц
CPI	1,0	1,1

1. У какого компьютера показатель MIPS выше?
2. Какой из компьютеров быстрее?

Миллион инструкций в секунду (MIPS) Измерение скорости выполнения программы, основанное на количестве миллионов инструкций. MIPS вычисляется как количество инструкций, поделенное на время выполнения, умноженное на 10^6.

1.9. Заключительные комментарии

> ENIAC содержит 18 000 электронных ламп и весит 30 тонн, а в будущем компьютеры могут состоять из тысячи электронных ламп и весить всего полторы тонны.
>
> *«Популярная механика», март 1949 года*

Хотя сделать точное предсказание, на каком уровне отношения стоимости к производительности окажутся компьютеры будущего, довольно трудно, можно совершенно точно сказать, что они будут лучше сегодняшних. Чтобы добиться определенных успехов, разработчики компьютеров и программисты должны разбираться в широком спектре проблем.

Разработчики оборудования, так же как разработчики программного обеспечения, создают компьютерные системы в иерархических уровнях, где каждый более низкий уровень скрывает свои тайны от более высокого уровня. Этот принцип абстрагирования положен в основу современных компьютерных систем, но это не означает, что конструкторы могут ограничиться знанием одной-единственной абстракции. Возможно, наиболее важным примером абстракции может стать интерфейс между оборудованием и низкоуровневым программным обеспечением, который называется архитектурой набора команд. Поддержание постоянства архитектуры набора команд дает возможность многим реализациям этой архитектуры, возможно отличающимся друг от друга стоимостью и производительностью, выполнять одни и те же программы. Недостаток такого подхода заключается в том, что архитектура может не допускать нововведений, требующих изменения интерфейса.

Существует испытанный метод определения и предоставления информации о производительности, оценочным показателем которого является время выполнения реальных программ. Это время выполнения зависит от других важных показателей и может быть получено с помощью следующей формулы:

Количество секунд на программу = Количество инструкций на программу × Количество тактовых циклов на инструкцию × Количество секунд на тактовый цикл.

Эта формула и составляющие ее показатели будут использоваться много раз. Но нужно помнить, что по отдельности эти показатели не определяют производительность: надежным оценочным показателем производительности может служить только их произведение, которое равно времени выполнения.

Самое важное

Единственным верным и безупречным оценочным показателем производительности является время выполнения. Было предложено и востребовано множество других оценочных показателей. Безупречность некоторых из них подвергалась сомнению с самого начала, поскольку они не отражали время выполнения, применение других, пригодных только при вполне определенных условиях, расширялось и переносилось за пределы этих условий или осуществлялось без дополнительных уточнений, позволяющих допустить их применение.

Основой технологии создания современных процессоров является применение кремния. Не менее важным в изучении технологии создания интегральных микросхем является понимание ожидаемых темпов технологических изменений. Наряду с тем, что применение кремния обеспечивает быстрый прогресс оборудования, улучшение соотношения цены и производительности осуществляется за счет использования в компьютерах новых организационных принципов. Двумя основными принципами являются использование параллельных вычислений с помощью нескольких процессоров и локализация доступа к уровням иерархической структуры памяти с помощью кэш-памяти.

Кроме площади пластины наиболее важным показателем в разработке микропроцессоров стала потребляемая мощность. Стремление остановить рост потребляемой мощности и увеличить производительность вынудило производителей оборудования перейти на изготовление многоядерных микропроцессоров, а это, в свою очередь, заставило производителей программного обеспечения перейти на создание программ для параллельного оборудования.

Конструкция компьютеров всегда оценивалась с точки зрения соотношения стоимости и производительности, но учитывались и другие важные факторы, такие как потребляемая мощность, надежность, стоимость эксплуатации и возможность расширения. Хотя данная глава касалась в основном стоимости, производительности и потребляемой мощности, в лучших конструкциях среди всех факторов упор делается на приемлемом балансе для определенного рынка сбыта.

Путеводитель по данной книге

В основу предложенных абстракций положены пять компонентов компьютера: операционный блок, блок управления, блок памяти, устройства ввода и вывода (см. рис. 1.4). Эти пять компонентов служат также структурной основой для всех остальных глав данной книги:

- операционный блок: главы 3, 4 и 7;
- блок управления: главы 4, 7;
- память: глава 5;
- устройство ввода: глава 6;
- устройство вывода: глава 6.

Как уже ранее упоминалось, в главе 4 описывается, как в процессорах используются скрытые параллельные вычисления, а в главе 7 рассматривается настоящий параллельный многоядерный микропроцессор. В главе 5 рассматривается, как различные компоненты памяти используют иерархию вычислений. В главе 2 описывается набор инструкций — интерфейс между компиляторами и компьютером — и подчеркивается роль компиляторов и языков программирования в наборе инструкций. В главе 3 рассматривается порядок обработки компьютерами арифметических данных.

1.10. Упражнения

Предоставлены Хавьером Бругейра

Большинство упражнений в данном издании составлены так, что в них даются качественные описания, а альтернативные количественные параметры приведены во вспомогательной таблице. Эти параметры нужны для решения задач, содержащихся в упражнении. При решении отдельных задач может использоваться любое количество параметров, сколько параметров нужно использовать для решения конкретной задачи — решать вам. Например, может быть сказано: «Решите задачу 4.1.1, используя параметры строки А таблицы». Если нужно, преподаватели могут переделать эти упражнения для создания новых решений, заменив заданные параметры своими собственными оригинальными значениями.

В разных главах используется разное число количественных упражнений, которое во многом зависит от рассматриваемой темы. Там, где количественный подход применить невозможно, используются более простые упражнения.

Относительный временной рейтинг упражнений показан в квадратных скобках после номера каждого упражнения. В среднем упражнение с рейтингом [10] должно занять в два раза больше времени, чем упражнение с рейтингом [5]. Разделы текста, которые должны быть изучены перед попыткой выполнения упражнения, будут даны в угловых скобках: например, <1.3> означает, что решить это упражнение поможет изучение раздела 1.3 «Что скрывается под корпусом компьютера».

Упражнение 1.1

Найдите слово или фразу из представленного списка, наиболее соответствующую описанию в приведенных ниже задачах. Используйте при ответах номера, стоящие слева от слова. Каждый из ответов должен использоваться только один раз.

1. виртуальные миры	14. операционная система
2. настольные компьютеры	15. компилятор
3. серверы	16. бит
4. самые дешевые серверы	17. инструкции
5. суперкомпьютеры	18. язык ассемблера
6. терабайт	19. машинный язык
7. петабайт	20. C
8. центры обработки данных	21. ассемблер
9. встроенные компьютеры	22. язык высокого уровня
10. многоядерный процессор	23. системные программы
11. VHDL	24. прикладные программы
12. оперативная память	25. Кобол
13. центральный процессор	26. Фортран

1.1.1 [2] <1.1> Компьютеры, используемые для решения серьезных задач, доступ к которым осуществляется обычно через сеть.

1.1.2 [2] <1.1> 10^{15} или 2^{50} байт.

1.1.3 [2] <1.1> Компьютеры, состоящие из нескольких сотен или тысяч процессоров и терабайтов памяти.

1.1.4 [2] <1.1> Сегодня эта сфера применения относится к области научной фантастики.

1.1.5 [2] <1.1> Тип памяти, который называется памятью с произвольным доступом.

1.1.6 [2] <1.1> Часть компьютера, называемая центральным процессорным устройством.

1.1.7 [2] <1.1> Тысячи процессоров, формирующие большой кластер.

1.1.8 [2] <1.1> Микропроцессор, содержащий несколько процессоров на одном и том же чипе.

1.1.9 [2] <1.1> Настольный компьютер без экрана или клавиатуры, доступ к которому обычно осуществляется по сети.

1.1.10 [2] <1.1> Самый распространенный в настоящее время класс компьютеров, которые запускают одно приложение или один набор взаимосвязанных приложений.

1.1.11 [2] <1.1> Специальный язык, используемый для описания компонентов оборудования.

1.1.12 [2] <1.1> Персональный компьютер, предоставляющий неплохую производительность отдельно взятому пользователю при умеренной стоимости.

1.1.13 [2] <1.2> Программа, транслирующая инструкции на языке высокого уровня в инструкции языка ассемблера.

1.1.14 [2] <1.2> Программа, транслирующая символьные инструкции в двоичные инструкции.

1.1.15 [2] <1.2> Язык высокого уровня для обработки бизнес-информации.

1.1.16 [2] <1.2> Язык двоичных цифр, понятный процессору.

1.1.17 [2] <1.2> Команды, которые может понять процессор.

1.1.18 [2]<1.2> Язык высокого уровня для научных вычислений.

1.1.19 [2] <1.2> Символьное представление инструкций машины.

1.1.20 [2] <1.2> Интерфейс между пользовательской программой и оборудованием, предоставляющий множество различных служб и контрольных функций.

1.1.21 [2] <1.2> Программное обеспечение или программы, разработанные пользователями.

1.1.22 [2] <1.2> Двоичный разряд (со значением 0 или 1).

1.1.23 [2] <1.2> Уровень программного обеспечения (операционная система и компиляторы), который находится между прикладным программным обеспечением и оборудованием.

1.1.24 [2] <1.2> Язык высокого уровня, используемый для написания приложений и системного программного обеспечения.

1.1.25 [2] <1.2> Переносимый язык, состоящий из слов и алгебраических выражений, которые перед запуском на компьютере должны быть переведены на язык ассемблера.

1.1.26 [2] <1.2> 10^{12} или 2^{40} байт.

Упражнение 1.2

1.2.1 [10] <1.3> Каким должен быть размер (в байтах) буфера кадра цветного дисплея, использующего по 8 бит для каждого из основных цветов (красного, зеленого, синего) на пиксел и имеющего разрешение экрана 1280 × 800 пикселов?

1.2.2 [5] <1.3> Если у компьютера имеется 2 Гбайт оперативной памяти и если предположить, что в ней не будет содержаться никакая другая информация, то сколько кадров она может вместить?

1.2.3 [5] <1.3> Если компьютеру, подключенному к сети Ethernet с пропускной способностью в 1 гигабит в секунду, нужно передать файл объемом 256 Кбайт, то сколько это займет времени?

1.2.4 [5] <1.3> Если предположить, что кэш-память в десять раз быстрее DRAM-памяти, которая, в свою очередь, в 100 000 раз быстрее памяти на магнитном диске, а флэш-память в 1000 раз быстрее памяти на магнитном диске, то сколько времени займет чтение файла из DRAM-памяти, с диска и с флэш-памяти, если чтение этого файла с кэш-памяти занимает 2 мкс?

Упражнение 1.3

Предположим, что три разных процессора: P1, P2 и P3 – выполняют один и тот же набор инструкций, имея при этом тактовые частоты и показатели CPI, указанные в таблице:

Процессор	Тактовая частота	CPI
P1	2 ГГц	1,5
P2	1,5 ГГц	1,0
P3	3 ГГц	2,5

1.3.1 [5] <1.4> У какого процессора будет наивысшая производительность?

1.3.2 [5] <1.4> Каждый из процессоров выполняет программу за 10 секунд, определите необходимое им для этого количество циклов и количество инструкций.

1.3.3 [10] <1.4> Мы пытаемся сократить время на 30%, но это влечет за собой увеличение показателя CPI на 20%. Какая нужна тактовая частота для получения такого сокращения времени?

Для расположенных ниже заданий воспользуйтесь информацией из таблицы:

Процессор	Тактовая частота	Число инструкций	Время
P1	2 ГГц	20×10^9	7 с
P2	1,5 ГГц	30×10^9	10 с
P3	3 ГГц	90×10^9	9 с

1.3.5 [5] <1.4> Определите тактовую частоту процессора P2, которая сравняет показатели времени работы с показателями процессора P1.

1.3.6 [5] <1.4> Определите количество инструкций для процессора P2, которое его время до процессора P3.

Упражнение 1.4

Рассмотрим две различные реализации одной и той же архитектуры набора инструкций.

Существует четыре класса инструкций: A, B, C и D. Тактовые частоты и CPI каждой реализации показаны в таблице:

	Тактовая частота	CPI класс A	CPI класс B	CPI класс C	CPI класс D
P1	1,5 ГГц	1	2	3	4
P2	2 ГГц	2	2	2	2

1.4.1 [10] <1.4> Заданная программа, имеющая 10^6 инструкций, разбивается по классам в следующих пропорциях: 10% инструкций относятся к классу A, 20% — к классу B, 50% — к классу C и 20% — к классу D, какая из реализаций работает быстрее?

1.4.2 [5] <1.4> Каким будет общий показатель CPI для каждой реализации?

1.4.3 [5] <1.4> Определите количество тактовых циклов, необходимое для обоих случаев.

В следующей таблице показано количество инструкций, используемых в программе:

Арифметика	Сохранение	Загрузка	Ветвление	Всего
500	50	100	50	700

1.4.4 [5] <1.4> Если предположить, что арифметическая инструкция выполняется за один цикл, инструкция загрузки и сохранения выполняется за пять циклов, а инструкция ветвления выполняется за два цикла, то каким будет время выполнения программы на процессоре с тактовой частотой 2 ГГц?

1.4.5 [5] <1.4> Определите CPI для этой программы.

1.4.6 [10] <1.4> Какими будут ускорение выполнения программы и показатель CPI, если количество инструкций загрузки удастся сократить в два раза?

Упражнение 1.5

Рассмотрим две различные реализации P1 и P2 одного и того же набора инструкций.

Инструкции в наборе относятся к одному из пяти классов (A, B, C, D и E). Тактовая частота каждой реализации и ее показатель CPI для каждого класса показаны в таблице:

		Тактовая частота	CPI класс A	CPI класс B	CPI класс C	CPI класс D	CPI класс E
а	P1	1,0 ГГц	1	2	3	4	3
	P2	1,5 ГГц	2	2	2	4	4
б	P1	1,0 ГГц	1	1	2	3	2
	P2	1,5 ГГц	1	2	3	4	3

1.5.1 [5] <1.4> Предположим, что наивысшая производительность определяется как наивысшая скорость, с которой компьютер может выполнить какую-нибудь последовательность инструкций. Какова наивысшая производительность P1 и P2, выраженная в количестве инструкций, выполняемых в секунду?

1.5.2 [5] <1.4> Если количество инструкций, выполняемых при работе конкретной программы, равномерно распределено между классами инструкций, за исключением класса A, чьи инструкции встречаются в два раза чаще, то какой из компьютеров работает быстрее? Насколько быстрее?

1.5.3 [5] <1.4> Если количество инструкций, выполняемых при работе конкретной программы, равномерно распределено между классами инструкций, за исключением класса E, чьи инструкции встречаются в два раза чаще, то какой из компьютеров работает быстрее? Насколько быстрее?

В следующей таблице показано распределение инструкций для разных программ. Используя эти данные, вы будете исследовать различные компромиссные решения, связанные с изменениями, вносимыми в MIPS-процессор, которые оказывают влияние на производительность.

		Число инструкций				
		Вычисление	Загрузка	Сохранение	Ветвление	Всего
а	Программа 1	1000	400	100	50	1550
б	Программа 4	1500	300	100	100	1750

1.5.4 [5] <1.4> Предположим, что вычисления осуществляются за один тактовый цикл, инструкции загрузки и сохранения требуют по 10 тактовых циклов, а ветвление осуществляется за 3 тактовых цикла. Определите время выполнения каждой программы на MIPS-процессоре с тактовой частотой 3 ГГц.

1.5.5 [5] <1.4> Предположим, что вычисления осуществляются за один тактовый цикл, инструкции загрузки и сохранения требуют по 2 тактовых цикла, а ветвление осуществляется за 3 тактовых цикла. Определите время выполнения каждой программы на MIPS-процессоре с тактовой частотой 3 ГГц.

1.5.6 [5] <1.4> Предположим, что вычисления осуществляются за один тактовый цикл, инструкции загрузки и сохранения требуют по 2 тактовых цикла, а ветвление осуществляется за 3 тактовых цикла. Каким тогда будет ускорение выполнения программы, если количество инструкций вычисления удастся сократить в два раза?

Упражнение 1.6

Компиляторы могут оказывать существенное влияние на производительность приложений на отдельно взятом процессоре. В этом упражнении будет исследоваться влияние компиляторов на время выполнения.

	Компилятор A		Компилятор B	
	Число инструкций	Время выполнения	Число инструкций	Время выполнения
а	$1{,}00 \times 10^9$	1 с	$1{,}20 \times 10^9$	1,4 с
б	$1{,}00 \times 10^9$	0,8 с	$1{,}20 \times 10^9$	0,7 с

1.6.1 [5] <1.4> Для одной и той же программы использовались два разных компилятора. В расположенной выше таблице показано время выполнения двух программ, полученных на выходе разных компиляторов. Определите средний показатель CPI для каждой программы при условии, что продолжительность тактового цикла процессора составляет 1 нс.

1.6.2 [5] <1.4> При средних показателях CPI, определенных в упражнении 1.6.1, скомпилированные программы запускаются на двух разных процессорах. Если время выполнения на двух процессорах одно и то же, то на сколько выше тактовая частота процессора, на котором запущен код, выданный компилятором A, по сравнению с тактовой частотой процессора, на котором запущен код, выданный компилятором B?

1.6.3 [5] <1.4> Разработан новый компилятор, который выдает только 600 миллионов инструкций и имеет средний показатель CPI 1,1. Каким будет ускорение при использовании этого нового компилятора по сравнению с использованием компиляторов A или B на исходном процессоре из упражнения 1.6.1?

Рассмотрим две разные реализации одного и того же набора инструкций: P1 и P2. В этом наборе существует пять классов инструкций (A, B, C, D и E). Процессор P1 имеет тактовую частоту 4 ГГц, а процессор P2 имеет тактовую частоту 6 ГГц. Среднее количество циклов для каждого класса инструкций для P1 и P2 перечислено в таблице:

	Класс	CPI на P1	CPI на P2
а	A	1	2
	B	2	2
	C	3	2
	D	4	4
	E	5	4
б	A	1	2
	B	1	2
	C	1	2
	D	4	4
	E	5	4

1.6.4 [5] <1.4> Предположим, что наивысшая производительность определяется как наивысшая скорость, с которой компьютер может выполнить какую-нибудь последовательность инструкций. Какова наивысшая производительность P1 и P2, выраженная в количестве инструкций, выполняемых в секунду?

1.6.5 [5] <1.4> Если в упражнении 1.6.4 количество инструкций, выполняемых при работе конкретной программы, равномерно распределено между классами инструкций, за исключением класса A, чьи инструкции встречаются в два раза чаще, то насколько P2 быстрее P1?

1.6.6 [5] <1.4> На какой тактовой частоте P2 имеет такую же производительность, как и P1 для набора инструкций, указанной в упражнении 1.6.5?

Упражнение 1.7

В таблице показано увеличение тактовой частоты и потребляемой мощности восьми поколений процессоров Intel за 28 лет:

Процессор	Тактовая частота	Потребляемая мощность
80286 (1982)	12,5 МГц	3,3 Вт
80386 (1985)	16 МГц	4,1 Вт
80486 (1989)	25 МГц	4,9 Вт
Pentium (1993)	66 МГц	10,1 Вт
Pentium Pro (1997)	200 МГц	29,1 Вт
Pentium 4 Willamette (2001)	2 ГГц	75,3 Вт
Pentium 4 Prescott (2004)	2 ГГц	103 вт
Core 2 Ketsfield (2007)	2,667 ГГц	95 Вт

1.7.1 [5] <1.5> Каково среднее геометрическое соотношений последовательных поколений как для тактовой частоты, так и для потребляемой мощности? (Среднее геометрическое рассмотрено в разделе 1.7.)

1.7.2 [5] <1.5> Какое наиболее существенное относительное изменение в тактовой частоте и потребляемой мощности среди поколений?

1.7.3 [5] <1.5> Насколько выше тактовая частота и потребляемая мощность последнего поколения процессоров по сравнению с их первым поколением?

Рассмотрим следующие значения напряжения питания для каждого поколения:

Процессор	Напряжение питания
80286 (1982)	5
80386 (1985)	5
80486 (1989)	5
Pentium (1993)	5
Pentium Pro (1997)	3,3
Pentium 4 Willamette (2001)	1,75
Pentium 4 Prescott (2004)	1,25
Core 2 Ketsfield (2007)	1,1

1.7.4 [5] <1.5> Определите среднюю емкостную нагрузку, пренебрегая статическим энергопотреблением.

1.7.5 [5] <1.5> Определите наибольшее относительное изменение в напряжении питания между поколениями.

1.7.6 [5] <1.5> Определите среднее геометрическое соотношений напряжений питания в поколениях, начиная с процессора Pentium.

Упражнение 1.8

Предположим, что нами разработаны новые версии процессора со следующими характеристиками:

Версия	Напряжение питания	Тактовая частота
Версия 1	5 В	0,5 ГГц
Версия 2	3,3 В	1,0 ГГц

1.8.1 [5] <1.5> На сколько снизилась емкостная нагрузка от версии к версии, если динамическая потребляемая мощность была уменьшена на 10%?

1.8.2 [5] <1.5> На сколько была снижена динамическая потребляемая мощность, если емкостная нагрузка не изменилась?

1.8.3 [5] <1.5> Предположим, что емкостная нагрузка версии 2 составляет 80% от емкостной нагрузки версии 1, определите напряжение питания для версии 2, если динамическая потребляемая мощность версии 2 снижена по сравнению с таким же показателем версии 1 на 40%.

Предположим, что в промышленности существуют тенденции появления технологического процесса нового поколения со следующими соотношениями показателей:

Емкостное сопротивление	Напряжение питания	Тактовая частота	Площадь
1	1/2	$2^{1/2}$	$2^{-1/2}$

1.8.4 [5] <1.5> Во сколько раз изменится динамическая потребляемая мощность?

1.8.5 [5] <1.5> Определите масштаб изменения емкостного сопротивления на единицу площади.

1.8.6 [5] <1.5> Используя данные из упражнения 1.7, определите напряжение питания процессора Core 2 для следующего поколения технологического процесса.

Упражнение 1.9

Хотя основным источником рассеиваемой мощности в CMOS является динамическое энергопотребление, статическое энергопотребление формируется за счет тока утечки: $V \times I_{утечки}$. Чем меньше размеры элементов чипа, тем более существенным является статическое энергопотребление. Предположим, что имеются данные по

статическому и динамическом энергопотреблению для нескольких поколений процессоров, показанные в следующей таблице:

	Технология	Динамическая мощность (Вт)	Статическая мощность (Вт)	Напряжение питания (В)
а	250 нм	49	1	3,3
б	90 нм	75	45	1,1

1.9.1 [5] <1.5> Определите процентное отношение от всей рассеиваемой мощности, приходящееся на статическое энергопотребление.

1.9.2 [5] <1.5> Если статическое энергопотребление зависит от тока утечки, $P_{ст} = V \times I_{утечки}$, определите ток утечки для каждой технологии.

1.9.3 [5] <1.5> Определите соотношение статического и динамического энергопотребления для каждой технологии.

Теперь рассмотрим динамическую рассеиваемую мощность различных версий заданного процессора для трех различных напряжений питания согласно следующей таблице:

	1,2 В	1,0 В	0,8 В
а	80 Вт	70 Вт	40 Вт
б	65 Вт	55 Вт	30 Вт

1.9.4 [5] <1.5> Определите статическое энергопотребление для каждой версии при напряжении питания 0,8 В, приняв за коэффициент соотношения статического и динамического энергопотребления значение 0,6.

1.9.5 [5] <1.5> Определите ток утечки для каждой версии при напряжении питания 0,8 В.

1.9.6 [10] <1.5> Определите наибольшее значение двух токов утечки при напряжениях питания 1,0 В и 1,2 В, приняв за коэффициент соотношения статического и динамического энергопотребления значение 1,7.

Упражнение 1.10

В таблице показано распределение инструкций заданного приложения, выполняемого на 1, 2, 4 или 8 процессорах. Используя эти данные, вы будете исследовать ускорение работы приложений на параллельных процессорах:

	Процессоры	Число инструкций на процессор			CPI		
		Арифм.	Загр./сохран.	Ветвления	Арифм.	Загр./сохран.	Ветвления
а	1	2560	1280	256	1	4	2
	2	1280	640	128	1	4	2
	4	640	320	64	1	4	2
	8	320	160	32	1	4	2

	Процессоры	Число инструкций на процессор			CPI		
		Арифм.	Загр./ сохран.	Ветвления	Арифм.	Загр./ сохран.	Ветвления
б	1	2560	1280	256	1	4	2
	2	1350	800	128	1	6	2
	4	800	600	64	1	9	2
	8	600	500	32	1	13	2

1.10.1 [5] <1.4, 1.6> В расположенных выше таблицах показано количество инструкций, необходимое для каждого процессора для выполнения программы на многопроцессорном устройстве с 1, 2, 4 или 8 процессорами. Каково общее количество инструкций, выполняемых каждым процессором? Каково суммарное количество инструкций, выполняемых на всех процессорах?

1.10.2 [5] <1.4, 1.6> Используя значения CPI, показанные в правой части таблицы, определите общее время выполнения этой программы на 1, 2, 4 и 8 процессорах, предположив, что тактовая частота каждого процессора составляет 2 ГГц.

1.10.3 [10] <1.4, 1.6> Если показатель CPI для арифметических инструкций был бы удвоен, то как это повлияло бы на время выполнения программы на 1, 2, 4 или 8 процессорах?

В следующей таблице показано количество инструкций, приходящихся на одно процессорное ядро многоядерного процессора, а также средний показатель CPI для выполнения программы на 1, 2, 4 или 8 ядрах. Используя эти данные, вы будете исследовать ускорение работы приложений на многоядерных процессорах.

	Ядер на процессор	Инструкций на ядро	Средний показатель CPI
а	1	$1{,}00 \times 10^{10}$	1,2
	2	$5{,}00 \times 10^{9}$	1,3
	4	$2{,}50 \times 10^{9}$	1,5
	8	$1{,}25 \times 10^{9}$	1,8
б	1	$1{,}00 \times 10^{9}$	1,2
	2	$5{,}00 \times 10^{9}$	1,2
	4	$2{,}50 \times 10^{9}$	1,2
	8	$1{,}25 \times 10^{9}$	1,2

1.10.4 [10] <1.4, 1.6> Предположим, что тактовая частота равна 3 ГГц, тогда каким будет время выполнения программы с использованием 1, 2, 4 или 8 ядер?

1.10.5 [10] <1.5, 1.6> Предположим, что потребляемая мощность процессорного ядра может быть описана следующим уравнением:

$$\text{Мощность} = \frac{5{,}0\ \text{мА}}{\text{МГц}} \text{Напряжение питания},$$

где напряжение питания процессора описывается следующим уравнением:

$$\text{Напряжение питания} = \frac{1}{4}\text{Частота} + 0{,}4,$$

в котором частота измеряется в гигагерцах. Таким образом, при частоте 5 ГГц напряжение питания будет равно 1,4 В. Определите потребляемую мощность при выполнении программы на 1, 2, 4 и 8 ядрах, предполагая, что каждое ядро работает на тактовой частоте 3 ГГц. Таким же образом определите энергопотребление при выполнении программы на 1, 2, 4 или 8 ядрах, предполагая, что каждое ядро работает на частоте 500 МГц.

1.10.6 [10] <1.5, 1.6> Определите затраты энергии при выполнении программы на 1, 2, 4 и 8 ядрах, предполагая что каждое ядро имеет тактовую частоту 3 ГГц и 500 МГц. Воспользуйтесь уравнением потребляемой мощности из упражнения 1.10.5.

Упражнение 1.11

В следующей таблице даны сведения для разных процессоров:

	Диаметр вафли	Пластин на вафлю	Дефектов на единицу площади	Стоимость одной вафли
а	15 см	90	0,018 дефекта/см^2	10
б	25 см	140	0,024 дефекта/см^2	20

1.11.1 [10] <1.7> Определите показатель выхода годной продукции.

1.11.2 [5] <1.7> Определите стоимость одной пластины.

1.11.3 [10] <1.7> Определите площадь пластины и показатель выхода годной продукции, если количество пластин на вафлю увеличится на 10%, а количество дефектов на единицу площади — на 15%.

Предположим, что благодаря совершенствованию технологии производства электронных устройств показатель выхода годной продукции изменялся, как показано в следующей таблице:

	T1	T2	T3	T4
Показатель выхода годной продукции	0,85	0,89	0,92	0,95

1.11.4 [10] <1.7> Определите количество дефектов на единицу площади для каждой технологии при площади пластины в 200 мм^2.

1.11.5 [5] <1.7> Изобразите графически изменение коэффициента выхода годной продукции по мере изменения количества дефектов на единицу площади.

Упражнение 1.12

В следующей таблице показаны результаты выполнения контрольных задач SPEC2006 на процессоре AMD Barcelona:

	Название	Количество инструкций × 10^9	Время выполнения (с)	Эталонное время (с)
а	perl	2118	500	9770
б	mcf	336	1200	9120

1.12.1 [5] <1.7> Определите CPI при продолжительности тактового цикла 0,333 нс.

1.12.2 [5] <1.7> Определите SPECratio.

1.12.3 [5] <1.7> Найдите для этих двух контрольных задач среднее геометрическое.

В следующей таблице показаны данные для еще двух контрольных задач:

	Название	CPI	Тактовая частота	SPECratio
а	sjeng	0,96	4 ГГц	14,5
б	omnetpp	2,94	4 ГГц	9,1

1.12.4 [5] <1.7> Определите увеличение процессорного времени при увеличении количества инструкций контрольных задач на 10%, которое не оказало никакого влияния на CPI.

1.12.5 [5] <1.7> Определите увеличение процессорного времени при увеличении количества инструкций контрольных задач на 10% и увеличении показателя CPI на 5%.

1.12.6 [5] <1.7> Определите изменение SPECratio для изменений, описанных в упражнении 1.12.5.

Упражнение 1.13

Предположим, что мы разрабатываем новую версию процессора AMD Barcelona с тактовой частотой 4 ГГц. Мы добавили к набору новые инструкции, но при этом сократили их общее количество на 15% от тех значений, которые были показаны для каждой контрольной задачи в упражнении 1.12. Полученное время выполнения показано в таблице:

	Название	Время выполнения (с)	Эталонное время (с)	SPECratio
а	perl	450	9770	21,7
б	mcf	1150	9120	7,9

1.13.1 [10] <1.8> Определите новые значения CPI.

1.13.2 [10] <1.8> В общем, эти значения CPI стали больше, чем те, которые были получены в предыдущих упражнениях для этих же контрольных задач. Главным

образом это произошло потому что тактовые частоты, использованные в обоих случаях, были равны 3 ГГц и 4 ГГц. Определите, пропорционально ли увеличение CPI соответствующему увеличению тактовой частоты. Если нет, то почему?

1.13.3 [5] <1.8> Насколько сократилось процессорное время?

В таблице показаны данные для еще двух контрольных задач:

	Название	Время выполнения (с)	CPI	Тактовая частота
а	sjeng	820	0,96	3 ГГц
б	omnetpp	580	2,94	3 ГГц

1.13.4 [10] <1.8> Определите количество инструкций, если время выполнения сократилось еще на 10% без изменения CPI, а тактовая частота равна 4 ГГц.

1.13.5 [10] <1.8> Определите тактовую частоту, требуемую для дальнейшего сокращения процессорного времени на 10% при неизменном количестве инструкций и не изменившемся показателе CPI.

1.13.6 [10] <1.8> Определите тактовую частоту, если показатель CPI снизился на 15%, а процессорное время — на 20% при неизменном количестве инструкций.

Упражнение 1.14

В разделе 1.8 рассматривалось заблуждение, касающееся использования для оценки производительности подгруппы уравнений производительности. Чтобы проиллюстрировать это положение, рассмотрим следующие данные по выполнению на разных процессорах заданной последовательности из 10^6 инструкций:

Процессор	Тактовая частота	CPI
P1	4 ГГц	1,25
P2	3 ГГц	0,75

1.14.1 [5] <1.8> Одним из распространенных заблуждений является отнесение компьютера с самой высокой тактовой частотой к компьютеру с наивысшей производительностью. Проверьте, является ли это утверждение справедливым для P1 и P2.

1.14.2 [10] <1.8> Еще одно заблуждение заключается в мнении о том, что процессору, выполняющему самое большое количество инструкций, понадобится и самое продолжительное процессорное время. Полагая, что процессор P1 выполняет последовательность из 10^6 инструкций и что CPI процессоров P1 и P2 не изменяются, определите количество инструкций, которое может выполнить процессор P2 за то же самое время, которое понадобится процессору P1 для выполнения 10^6 инструкций.

1.14.3 [10] <1.8> Одним из распространенных заблуждений является использование показателя MIPS (миллион инструкций в секунду) для сравнения производительности двух разных процессоров и мнение о том, что процессор, имеющий более высокий показатель MIPS обладает более высокой производительностью. Проверьте, является ли это утверждение справедливым для процессоров P1 и P2.

Еще одним распространенным показателем производительности является MFLOPS (миллион операций с плавающей точкой в секунду), определяемый как
MFLOPS = Количество операций с плавающей точкой / Время выполнения × 10^6

Но у этого показателя те же самые недостатки, что и у MIPS. Рассмотрим программы, запущенные на процессоре с тактовой частотой, равной 3 ГГц:

	Количество инструкций	Инстр. загр./сохр.	Инстр. с плав. точкой	Инстр. ветвлений	CPI (загр./сохр.)	CPI (с плав. точкой)	CPI (ветвл.)
а	10^6	50%	40%	10%	0,75	1	1,5
б	3×10^6	40%	40%	20%	1,25	0,70	1,25

1.14.4 [10] <1.8> Определите показатели MFLOPS для этих программ.

1.14.5 [10] <1.8> Определите показатели MIPS для этих программ.

1.14.6 [10] <1.8> Определите производительность, показанную при выполнении программ, и сравните ее с MIPS и MFLOPS.

Упражнение 1.15

Еще одно заблуждение, упомянутое в разделе 1.8, заключается в ожидании увеличения общей производительности компьютера за счет улучшения только одной из его компонент. Такое может произойти, но не всегда. Рассмотрим компьютер, выполняющий программы с процессорными временами, приведенными таблице:

	Инстр. с плав. точкой	Целочисленные инструкции	Инструкции загр./сохр.	Инструкции ветвления	Общее время
а	35 с	85 с	50 с	30 с	200 с
б	50 с	80 с	50 с	30 с	210 с

1.15.1 [5] <1.8> На сколько сократится общее время, если время для операций с плавающей точкой будет сокращено на 20%?

1.15.2 [5] <1.8> На сколько сократится время выполнения целочисленных операций, если общее время сократится на 20%?

1.15.3 [5] <1.8> Может ли общее время сократиться на 20% за счет сокращения одного только времени на выполнение инструкций ветвления?

В следующей таблице показано распределение инструкций по типам для каждого процессора при выполнении заданного приложения на различном количестве процессоров:

	Кол-во процессоров	Инстр. с плав. точкой	Целочисл. инстр.	Инстр. загр./сохр.	Инстр. ветвл.	CPI (с плав. точкой)	CPI (целочисл.)	CPI (загр./сохр.)	CPI (ветвл.)
а	1	560×10^6	2000×10^6	1280×10^6	256×10^6	1	1	4	2
б	8	80×10^6	240×10^6	160×10^6	32×10^6	1	1	4	2

Предположим, что каждый процессор работает на тактовой частоте 2 ГГц.

1.15.4 [10] <1.8> На сколько нужно улучшить показатель CPI для инструкций с плавающей точкой, если требуется, чтобы программа работала в два раза быстрее?

1.15.5 [10] <1.8> На сколько нужно улучшить показатель CPI для инструкций загрузки-сохранения, если требуется, чтобы программа работала в два раза быстрее?

1.15.6 [5] <1.8> На сколько сократится время выполнения программы, если показатель CPI для целочисленных инструкций и инструкций с плавающей точкой сократится на 40%, а показатель CPI для инструкций загрузки-сохранения и инструкций ветвления сократится на 30%?

Упражнение 1.16

Еще одно заблуждение, касающееся выполнения программ на многопроцессорных системах, заключается в ожидании повышения производительности за счет сокращения времени выполнения только части подпрограмм. В таблице показано время выполнения пяти подпрограмм, запущенных на разном количестве процессоров.

	Количество процессоров	Подпрограмма A (мс)	Подпрограмма B (мс)	Подпрограмма C (мс)	Подпрограмма D (мс)	Подпрограмма E (мс)
а	2	20	80	10	70	5
б	16	4	14	2	12	2

1.16.1 [10] <1.8> Определите общее время выполнения и степень его сокращения, если время выполнения подпрограмм A, C и E будет сокращено на 15%.

1.16.2 [10] <1.8> На сколько сократится общее время, если время выполнения подпрограммы B сократится на 10%?

1.16.3 [10] <1.8> На сколько сократится общее время, если время выполнения подпрограммы D сократится на 10%?

Время выполнения в многопроцессорной системе может быть разбито на время вычисления подпрограмм плюс время, затрачиваемое на обмен данными между процессорами (время маршрутизации). Рассмотрим показатели времени выполнения и времени маршрутизации, представленные в таблице. В данном случае время маршрутизации является важным компонентом общего времени.

Количество процессоров	Подпрограмма A (мс)	Подпрограмма B (мс)	Подпрограмма C (мс)	Подпрограмма D (мс)	Подпрограмма E (мс)	Маршрутизация (мс)
2	20	78	9	65	4	11
4	12	44	4	34	2	13
8	1	23	3	19	3	17
16	4	13	1	10	2	22
32	2	5	1	5	1	23
64	1	3	0,5	1	1	26

1.16.4 [10] <1.8> Определите для каждого удвоения количества процессоров соотношения времен, расходуемых на вычисления и маршрутизацию.

1.16.5 [5] <1.8> Используя средние геометрические значения соотношений, проведите экстраполяцию для определения времени на вычисления и времени маршрутизации в 128-процессорной системе.

1.16.6 [10] <1.8> Определите время на вычисления и время маршрутизации для системы с одним процессором.

Ответы на вопросы для самопроверки

Раздел 1.1: Вопросы даны для обсуждения, поэтому приемлемо множество ответов.

Раздел 1.3: Дисковая память: энергонезависимая, имеющая большое время доступа (порядка нескольких миллисекунд), стоимость от 20 центов до 2 долларов за гигабайт. Полупроводниковая память: энергозависимая, имеющая небольшое время доступа (порядка нескольких наносекунд), стоимость от 20 до 75 долларов за гигабайт.

Раздел 1.4, самопроверка 1: 1. *а*: улучшаются оба параметра; *б*: сокращается время отклика, в: улучшений не происходит. 2. 7 секунд.

Раздел 1.4, самопроверка 2: *б*.

Раздел 1.7: К правильным относятся причины, указанные под номерами 1, 3 и 4. Ответ под номером 5 также подходит, поскольку большой объем выпуска может привлечь дополнительные инвестиции для сокращения размера пластины, скажем, на 10%, что является неплохим экономическим решением.

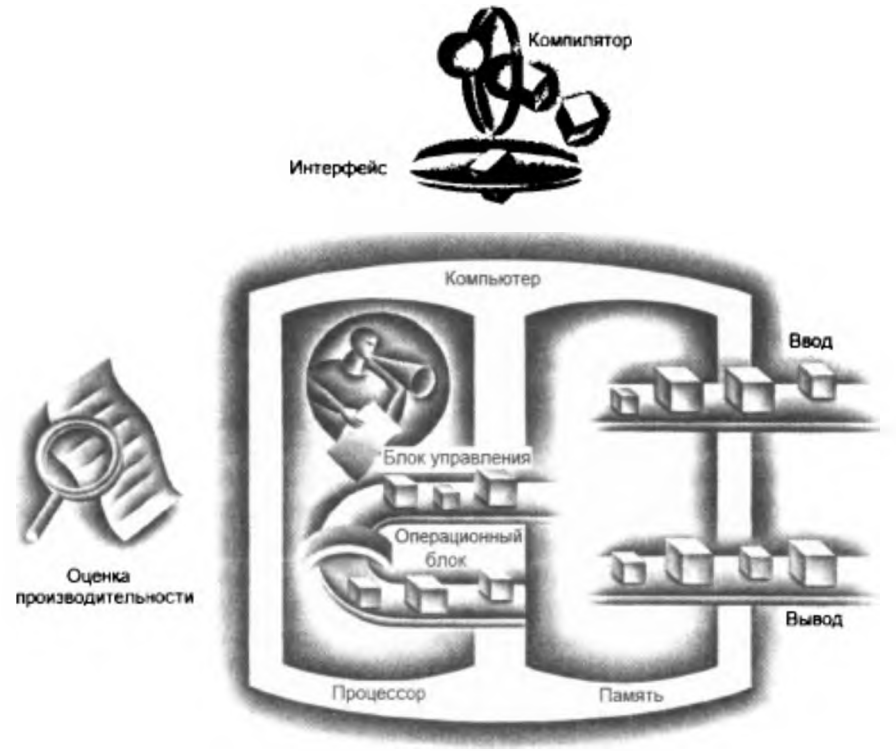
Раздел 1.8: *а*: компьютер А имеет более высокий показатель MIPS; *б*: компьютер В работает быстрее.

Глава 2

Инструкции: язык компьютера

С Богом я разговариваю на испанском, с женщинами на итальянском, с мужчинами на французском, а со своей лошадью на немецком.

Карл V, король Франции



2.1. Введение

Чтобы управлять компьютерным оборудованием, нужно говорить на его языке. Слова компьютерного языка называются *инструкциями*, а его словарь называется

набором инструкций. В данной главе будет рассмотрен набор инструкций настоящего компьютера, который будет представлен в двух формах — написанной людьми и прочитанной компьютером. Инструкции будут представлены по нисходящей. Мы начнем с формы записи, которая выглядит как некий весьма ограниченный язык программирования, и будем постепенно конкретизировать ее, пока не станет виден настоящий язык настоящего компьютера. Наше углубление в эту тему продолжится в главе 3, где будет рассмотрено оборудование для арифметических операций и представлены числа с плавающей точкой.

Может сложиться мнение, что компьютерные языки имеют такое же разнообразие, что и человеческие, но на самом деле настоящие компьютерные языки очень похожи друг на друга и напоминают больше местные диалекты, чем абсолютно разные языки. Следовательно, изучив один из таких языков, можно без особого труда освоить и все остальные. Сходство языков обусловлено тем, что все компьютеры построены на аппаратных технологиях, имеющих одни и те же основные принципы, и поэтому все компьютеры должны обеспечивать выполнение ряда базовых операций. Более того, у всех разработчиков компьютеров одна и та же цель: подобрать такой язык, который упростит создание аппаратуры и компилятора, обеспечив при этом максимальную производительность при минимальных стоимости и потребляемой мощности. И эта цель со временем не претерпела изменений. Следующее высказывание было сделано задолго до появления у вас возможности приобрести компьютер, в 1947 году, и не утратило своей актуальности и сегодня:

> Используя методы формальной логики, нетрудно заметить, что существует определенный [набор инструкций], вполне достаточный для управления и инициализации выполнения любой последовательности операций.... С современной точки зрения по-настоящему конструктивные предложения по выбору [набора инструкций] исходят большей частью из практических соображений: простота востребованного [набором команд] оборудования и четкое осознание области его применения для решения актуальных задач в сочетании со скоростью их решения.
>
> *Баркс, Голдстин и фон Нейман, 1947 год*

Для современных компьютеров «простота оборудования» является таким же ценным фактором, что и в далекие 50-е годы прошлого века. Цель данной главы — изучить набор инструкций, соответствующий приведенному в цитате определению конструктивности, с одновременным показом, как этот набор представлен на аппаратном уровне и какие взаимоотношения существуют между языками программирования высокого уровня и более примитивным машинным языком. Все примеры представлены на языке программирования C.

При изучении способов представления инструкций вам также будет раскрыт секрет вычислительного процесса: **концепция хранимой программы**. Кроме этого, вы получите практику применения «иностранного языка», составляя программы на языке компьютера. Вы также поймете, какое влияние оказывают языки программирования и оптимизация компиляторов

Набор инструкций
Словарь, состоящий из команд, понятных данной архитектуре.

Концепция хранимой программы
Идея о том, что в памяти в виде чисел могут храниться инструкции и данные многих типов, которая привела к появлению компьютеров, хранящих программу в памяти.

на производительность. В заключение будет дан обзор исторической эволюции наборов инструкций и других компьютерных диалектов.

Выбранный нами набор инструкций был разработан компанией MIPS Technologies и представляет собой самый удачный пример из всего, что разрабатывалось начиная с 1980-х годов. Затем немного внимания будет уделено двум другим популярным наборам инструкций. Набор ARM очень похож на MIPS, а в 2008 году в составе встраиваемых устройств было поставлено три миллиарда ARM-процессоров. Другой пример, набор инструкций Intel x86, применяется на 330 миллионах персональных компьютеров, произведенных в 2008 году.

Мы будем рассматривать набор инструкций MIPS поэтапно, рассматривая наряду с ними и структуры компьютера. Это учебное руководство, в котором материал излагается поэтапно, от более общих к более конкретным понятиям, включает в объяснения компьютерные компоненты, делая язык компьютера более понятным и привлекательным. Предварительный обзор набора инструкций, рассматриваемого в данной главе, дается в табл. 2.1.

2.2. Операции, осуществляемые компьютерным оборудованием

Каждый компьютер должен выполнять арифметические операции. Запись на языке ассемблера MIPS

```
add a, b, c
```

приказывает компьютеру сложить две переменные `b` и `c` и поместить их сумму в переменную `a`.

Эта система записи жестко задана для каждой арифметической инструкции MIPS, выполняющей только одну операцию, и всегда должна иметь именно три переменные. Предположим, к примеру, что нужно поместить сумму четырех переменных `b`, `c`, `d` и `e`, в переменную `a`. (В данном разделе мы намеренно не даем определения тому, что такое «переменная», оставив это для следующего раздела.)

Сумма четырех переменных получается в результате выполнения следующей последовательности инструкций:

```
add a, b, c       # Сумма b и c помещается в a
add a, a, d       # Теперь в a будет сумма b, c и d
add a, a, e       # А теперь в a будет сумма b, c, d и e
```

Таким образом, для получения суммы четырех переменных понадобились три инструкции.

Все слова, находящиеся в каждой строке справа от символа решетки (#), являются *комментариями*, предназначенными для человеческого восприятия. Компьютер их игнорирует. Следует заметить, что в отличие от других языков программирования каждая строка этого языка может содержать не более одной инструкции. Другим отличием от языка C является то, что комментарии всегда завершаются в конце строки.

Таблица 2.1. Язык ассемблера MIPS, рассматриваемый в данной главе

Операнды MIPS

Название	Пример	Комментарии
32 регистра	$s0–$s7, $t0–$t9, $zero, $a0–$a3, $v0–$v1, $gp, $fp, $sp, $ra, $at	Легкодоступное средство размещения данных. Для выполнения арифметических операций данные в MIPS всегда должны помещаться в регистры, регистр $zero всегда равен 0, регистр $at зарезервирован ассемблером для работы с большими константами.
2^{30} слов памяти	Memory[0], Memory[4], ..., Memory[4294967292]	Доступны только инструкциям по перемещению данных. MIPS использует байтовую адресацию, поэтому последовательные адреса слов отличаются друг от друга на 4. В памяти хранятся структуры данных, массивы и регистры с «утечкой» содержимого

Язык ассемблера MIPS

Категория инструкций	Инструкции	Пример	Значение	Комментарии
Арифметические	add (сложение)	add $s1,$s2,$s3	$s1 = $s2 + $s3	Трехрегистровые операнды
	subtract (вычитание)	sub $s1,$s2,$s3	$s1 = $s2 – $s3	Трехрегистровые операнды
	add immediate (непосредственное сложение)	addi $s1,$s2,20	$s1 = $s2 + 20	Используется для прибавления константы
Переноса данных	load word (загрузка слова)	lw $s1,20($s2)	$s1 = Memory[$s2 + 20]	Слово из памяти в регистр
	store word (сохранение слова)	sw $s1,20($s2)	Memory[$s2 + 20] = $s1	Слово из регистра в память
	load half (загрузка половины)	lh $s1,20($s2)	$s1 = Memory[$s2 + 20]	Половина слова из памяти в регистр
	load half unsigned (загрузка половины без знака)	lhu $s1,20($s2)	$s1 = Memory[$s2 + 20]	Половина слова из памяти в регистр
	store half (сохранение половины)	sh $s1,20($s2)	Memory[$s2 + 20] = $s1	Половина слова из регистра в память

Категория инструкций	Инструкции	Пример	Значение	Комментарии
	load byte (загрузка байта)	lb $s1,20($s2)	$s1 = Memory[$s2 + 20]	Байт из памяти в регистр
	load byte unsigned (загрузка байта без знака)	lbu $s1,20($s2)	$s1 = Memory[$s2 + 20]	Байт из памяти в регистр
	store byte (сохранение байта)	sb $s1,20($s2)	Memory[$s2 + 20] = $s1	Байт из регистра в память
	load linked word (загрузка связанного слова)	ll $s1,20($s2)	$s1 = Memory[$s2 + 20]	Загрузка слова в качестве первой половины атомарного обмена
	store condition. word (сохранение условного слова)	sc $s1,20($s2)	Memory[$s2+20]=$s1; $s1=0 или 1	Сохранение слова в качестве второй половины атомарного обмена
	load upper immed. (непосредственная загрузка в верхние разряды)	lui $s1,20	$s1 = 20 * 2^{16}	Загрузка константы в верхние 16 разрядов
Логические	and (И)	and $s1,$s2,$s3	$s1 = $s2 & $s3	Трехрегистровые операнды; поразрядное И
	or (ИЛИ)	or $s1,$s2,$s3	$s1 = $s2 | $s3	Трехрегистровые операнды; поразрядное ИЛИ
	nor (исключающее ИЛИ)	nor $s1,$s2,$s3	$s1 = ~ ($s2 | $s3)	Трехрегистровые операнды; поразрядное исключающее ИЛИ
	and immediate (непосредственное «И»)	andi $s1,$s2,20	$s1 = $s2 & 20	Поразрядное И регистра и константы
	or immediate (непосредственное «ИЛИ»)	ori $s1,$s2,20	$s1 = $s2 | 20	Поразрядное ИЛИ регистра и константы

Категория инструкций	Инструкции	Пример	Значение	Комментарии
	shift left logical (логический сдвиг влево)	sll $s1,$s2,10	$s1 = $s2 << 10	Сдвиг влево на константу
	shift right logical (логический сдвиг вправо)	srl $s1,$s2,10	$s1 = $s2 >> 10	Сдвиг вправо на константу
Условного ветвления	branch on equal (ветвление при равенстве)	beq $s1,$s2,25	if ($s1 == $s2) go to PC + 4 + 100	Тест на равенство; ветвление, связанное со счетчиком команд
	branch on not equal (ветвление при неравенстве)	bne $s1,$s2,25	if ($s1!= $s2) go to PC + 4 + 100	Тест на неравенство; ветвление, связанное со счетчиком команд
	set on less than (установить, если меньше чем)	slt $s1,$s2,$s3	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Сравнение «меньше чем»; для beq, bne
	set on less than unsigned (установить, если меньше, чем без знака)	sltu $s1,$s2,$s3	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Сравнение «меньше чем» без знака
	set less than immediate (установить, если непосредственно меньше чем)	slti $s1,$s2,20	if ($s2 < 20) $s1 = 1; else $s1 = 0	Сравнение «меньше чем» с константой
	set less than immediate unsigned (установить, если непосредственно меньше, чем без знака)	sltiu $s1,$s2,20	if ($s2 < 20) $s1 = 1; else $s1 = 0	Сравнение «меньше чем» с константой, без знака

Категория инструкций	Инструкции	Пример	Значение	Комментарии
Безусловного перехода	jump (перейти)	j 2500	go to 10000	Переход по указанному адресу
	jump register (перейти по регистру)	jr $ra	go to $ra	Для переключения; возвращение из процедуры
	jump and link (перейти и сделать ссылку)	jal 2500	$ra = PC + 4; go to 10000	Для вызова процедуры

Для операций, подобных сложению, вполне естественным является использование трех операндов: два складываемых числа и место, куда будет помещена сумма. Требование наличия у каждой операции трех операндов, не больше и не меньше, соответствует философии «простоты»: аппаратура для разного количества операндов сложнее аппаратуры для их фиксированного количества. Эта ситуация иллюстрирует первый из четырех основополагающих принципов конструирования оборудования:

Принцип конструирования 1: простота предпочитает постоянство.

Теперь в следующих двух упражнениях мы можем показать связь программ, написанных на языках программирования высокого уровня, с программами, использующими эту более простую систему записи.

Упражнение

Компиляция двух инструкций присваивания на языке C в MIPS-код

Данный фрагмент программы на языке C содержит пять переменных: a, b, c, d и e. Поскольку язык Java произошел от языка C, это и несколько других упражнений работают на любом из этих языков высокого уровня:

```
a = b + c;
d = a - e;
```

Трансляция кода с языка C на язык ассемблера MIPS выполняется *компилятором*. Покажите код MIPS, производимый компилятором.

Ответ

MIPS-инструкция работает с двумя исходными операндами и помещает результат в один целевой операнд. Следовательно, две простые инструкции, показанные выше, компилируются в следующие две инструкции языка ассемблера MIPS:

```
add a, b, c
sub d, a, e
```

Упражнение

Компиляция сложной инструкции на языке C в MIPS-код

В относительно сложной инструкции содержится пять переменных — f, g, h, i и j:

```
f = (g + h) - (i + j);
```

Каким будет итог работы C-компилятора?

Ответ

Компилятор должен разбить эти инструкции на несколько ассемблерных инструкций, поскольку на одну MIPS-инструкцию может приходиться только одна операция. Первая MIPS-инструкция вычислит сумму переменных `g` и `h`. Результат нужно куда-нибудь поместить, поэтому компилятор создаст временную переменную по имени `t0`:

```
add t0,g,h    # временная переменная t0 содержит g + h
```

Хотя следующей по очереди будет операция вычитания, перед ее осуществлением нужно вычислить сумму переменных `i` и `j`. Поэтому вторая инструкция помещает сумму `i` и `j` в другую временную переменную, создаваемую компилятором, которая называется `t1`:

```
add t1,i,j    # временная переменная t1 содержит i + j
```

И наконец, инструкция вычитания из первой суммы вычитает первую, а разницу помещает в переменную `f`, завершая тем самым откомпилированный код:

```
sub f,t0,t1    # f получает значение t0 - t1, то есть (g + h) - (i + j)
```

Самопроверка

В каком языке программирования потребуется больше строк кода для отдельно взятой функции? Выстройте указанные ниже языки в нужном порядке.

1. Java.
1. C.
2. Язык ассемблера MIPS.

Уточнение. Чтобы улучшить переносимость программ, Java изначально рассматривался как язык, зависимый от программного интерпретатора. Набор инструкций интерпретатора называется *байт-кодами Java*, которые сильно отличаются он набора инструкций MIPS. Чтобы приблизить производительность к производительности программ, написанных на языке C, сегодня Java-системы обычно компилируют в набор инструкций, присущий компьютерной системе, например в MIPS. Поскольку такая компиляция осуществляется намного позднее компиляции программ на языке C, такие Java-компиляторы часто называют JIT-компиляторами (*Just In Time* — к нужному моменту). В разделе 2.12 показано, что в процессе запуска JIT-компиляторы используются позднее C-компиляторов, а в разделе 2.13 показано, как отражается на производительности компиляция по сравнению с интерпретацией Java-программ.

2.3. Операнды компьютерного оборудования

В отличие от программ на языках высокого уровня операнды арифметических инструкций строго регламентированы; они относятся к ограниченному числу специальных ячеек памяти, встроенных непосредственно в оборудование и называемых *регистрами*. Регистры являются основными элементами, используемыми в конструкции оборудования, которые также видны программисту, когда компьютер находится в окончательно собранном виде, поэтому о регистрах можно думать

как о неких кирпичиках, присутствующих в конструкции компьютера. Размер регистра в MIPS-архитектуре составляет 32 бита (разряда); группы, состоящие из 32 разрядов, встречаются настолько часто, что в MIPS-архитектуре им присвоено название **слово**.

Одним из основных отличий между переменными в языке программирования и регистрами является ограниченное число регистров, которое обычно в современных компьютерах вроде MIPS равно 32. Таким образом, продолжая наше нисходящее пошаговое углубление в язык MIPS, в этом разделе мы добавили ограничение, состоящее в том, что каждый из трех операндов арифметических инструкций MIPS должен быть выбран из одного из тридцати двух 32-разрядных регистров.

Причина ограничения количества регистров числом 32 может быть найдена во втором из наших принципов конструирования, положенных в основу технологии компьютерного оборудования:

Принцип конструирования 2: чем меньше, тем быстрее.

Слишком большое количество регистров может увеличить продолжительность тактового цикла просто потому, что чем длиннее путь электронных сигналов, тем дольше одни идут.

Положения типа «чем меньше, тем быстрее» не носят абсолютного характера: 31 регистр может работать не быстрее 32 регистров. Все же справедливость подобных наблюдений заставляет разработчиков компьютеров относиться к ним со всей серьезностью. В данном случае конструктор должен выбрать разумный баланс между требованием для программ большего количества регистров с конструкторским желанием сохранить быстроту тактового цикла. Еще одна причина ограничиться использованием 32 регистров, которая показывается в разделе 2.5, — количество разрядов, которые будут использоваться в формате инструкции.

В главе 4 показывается ведущая роль, которую играют эти регистры в конструкции оборудования, и, как мы увидим в этой главе, эффективность использования регистров является критическим фактором производительности программы.

Хотя можно было бы просто написать инструкции, используя для регистров номера от 0 до 31 в соответствии с соглашением, в MIPS для представления регистров используются имена из двух символов, следующие за знаком доллара. В разделе 2.8 будет объяснена причина выбора таких имен. А сейчас мы будем использовать для регистров, соотносящихся с переменными в C- и Java-программах, имена `$s0`, `$s1`, ..., а для временных регистров, необходимых для компиляции программы в MIPS-инструкции, имена `$t0`, `$t1`, ...

Упражнение

Компиляция присваивания на языке C с использованием регистров

Задача связывания переменных программы с регистрами возлагается на компилятор. Возьмем, к примеру, инструкцию присваивания из нашего прежнего примера:

```
f = (g + h) - (i + j);
```

Переменные `f`, `g`, `h`, `i` и `j` назначаются соответственно регистрам `$s0`, `$s1`, `$s2`, `$s3` и `$s4`. Как будет выглядеть MIPS-код, полученный в результате компиляции?

Слово

Естественная единица размера памяти в компьютере, обычно это группа из 32 разрядов, соответствующая размеру регистра в MIPS-архитектуре.

Ответ

Откомпилированная программа очень похожа на предыдущий пример, за исключением того, что мы заменяем переменные именами упомянутых ранее регистров и дополнительно используем два временных регистра, $t0 и $t1, которые соответствуют показанным выше временным переменным:

```
add $t0,$s1,$s2  # регистр $t0 содержит g + h
add $t1,$s3,$s4  # регистр $t1 содержит i + j
sub $s0,$t0,$t1  # f получает значение $t0 - $t1, которое равно (g + h)-(i + j)
```

Операнды памяти

В языках программирования имеются, как в этих примерах, простые переменные, содержащие элементы данных, но есть также и более сложные структуры данных, например массивы. Эти сложные структуры данных могут содержать больше элементов данных, чем количество регистров в компьютерах. Как компьютер может представить такие крупные структуры и получить к ним доступ?

Вспомним пять компонентов компьютера, которые были представлены в главе 1 и повторно показаны в начале данной главы. Процессор может хранить в регистрах только небольшой объем данных, но компьютерная память содержит миллиарды элементов данных. Следовательно, структуры данных (массивы и структуры) хранятся в памяти.

Как ранее объяснялось, арифметические операции проводится в MIPS-инструкциях только с регистрами, следовательно, MIPS-ассемблер должен включать инструкции, переносящие данные между памятью и регистрами. Такие инструкции называются **инструкциями переноса данных**. Для доступа к слову в памяти инструкции нужно предоставить **адрес** памяти. Память — это просто большой одномерный массив, в котором адрес ведет себя как индекс этого массива, начинаясь с 0. Например, на рис. 2.1 третий элемент данных имеет адрес 2, а значение элемента `Memory[2]` равно 10.

Рис. 2.1. Адреса и содержимое памяти по этим адресам. Если бы эти элементы были словами, данные адреса были бы неправильными, поскольку в MIPS обычно используется байтовая адресация, где каждое слово представлено четырьмя байтами. На рис. 2.2 показана адресация памяти для последовательной пословной адресации

Инструкция переноса данных, которая копирует данные из памяти в регистр, традиционно называется *загрузкой* (*load*). Формат инструкции загрузки состоит из имени операции, за которым следует загружаемый регистр, а затем константа и регистр, используемые для доступа к памяти. Сумма константы и содержимого второго регистра формирует адрес памяти. Настоящее MIPS-имя для этой инструкции — `lw`, что означает *загрузить слово* (*load word*).

Упражнение

Компиляция присваивания, когда операнд находится в памяти

Предположим, что `A` — массив из 100 слов, и компилятор, как и раньше, связал переменные `g` и `h` с регистрами `$s1` и `$s2`. Также предположим, что стартовый, или базовый, адрес массива находится в регистре `$s3`. Откомпилируйте эту инструкцию присваивания на языке C:

```
g = h + A[8];
```

Ответ

Хотя в этой инструкции присваивания содержится всего одна операция, один из операндов находится в памяти, поэтому сначала нужно перенести `A[8]` в регистр. Адресом этого элемента массива является сумма базы массива `A`, которая находится в регистре `$s3`, и номера выбранного элемента `8`. Данные должны быть помещены во временный регистр, чтобы их можно было использовать в следующей инструкции. На основе схемы, изображенной на рис. 2.1, первой инструкцией, выданной компилятором, будет

```
lw $t0,8($s3)  # Временный регистр $t0 получает A[8]
```

(Далее мы слегка подправим эту инструкцию, но пока будем пользоваться этой упрощенной версией.) Следующая инструкция может работать со значением `$t0` (которое равно `A[8]`), поскольку это регистр. Инструкция должна прибавить значение `h` (содержащееся в `$s2`) к `A[8]` (`$t0`) и поместить сумму в регистр, соответствующий переменной `g` (которая связана с `$s1`):

```
add $s1,$s2,$t0 # g = h + A[8]
```

Константа в инструкции переноса данных (`8`) называется *смещением* (*offset*), а регистр, прибавляемый для формирования адреса (`$s3`), называется *базовым регистром* (*base register*).

Интерфейс аппаратного и программного обеспечения

Кроме связывания переменных с регистрами, компилятор распределяет для структур данных вроде массивов место в памяти. Затем компилятор может поместить соответствующий стартовый адрес в инструкции переноса данных.

Инструкция переноса данных
Команда, которая перемещает данные между памятью и регистрами.

Поскольку во многих программах применяются восьмибитные *байты*, большинство архитектур осуществляют адресацию отдельных байтов. Поэтому адрес слова соответствует адресу одного из четырех байтов этого слова, а адреса

Адрес
Значение, используемое для описания местоположения конкретного элемента данных в массиве памяти.

последовательно расположенных слов отличаются на 4. Например, на рис. 2.2 показаны реальные MIPS-адреса для слов с рис. 2.1: байтовый адрес третьего слова равен 8.

В MIPS слова должны начинаться с адресов, кратных четырем. Это требование называется **условием выравнивания** и имеется у многих архитектур. (В главе 4 показывается, почему выравнивание приводит к ускорению переноса данных.)

Компьютеры делятся на те, которые используют в качестве адреса слова адрес самого левого, или «старшего» («big-endian») байта, и на те, которые используют адрес самого правого, или «младшего» («little-endian») байта. MIPS относится к лагерю *«старшебайтников» (big-endian)*.[1]

Байтовая адресация также отражается и на индексе массива. Чтобы получить соответствующий байтовый адрес в представленном ранее коде, *смещение, прибавляемое к базовому регистру* `$s3`, *должно быть 4 × 8, или 32*, чтобы в качестве адреса загрузки выбирался `A[8]`, а не `A[8/4]`. (См. связанное с этим заблуждение, рассматриваемое в разделе 2.18.)

Рис. 2.2. Реальная адресация памяти в MIPS и содержимое памяти для этих слов. Измененные адреса выделены, чтобы отличаться от тех, что были показаны на рис. 2.1. Поскольку в MIPS адресуется каждый байт, адреса слов кратны четырем: в слове содержится 4 байта

Дополняющей для инструкции загрузки является инструкция, которая традиционно называется *сохранением* (*store*); она копирует данные из регистра в память. Формат у сохранения такой же, как и у загрузки: имя операции, затем сохраняемый регистр, после него смещение к выбранному элементу массива и, наконец, базовый регистр. И еще раз: MIPS-адрес указывается частично константой и частично содержимым регистра. Настоящее MIPS-имя для этой инструкции — `sw`, что означает *сохранить слово* (*store word*).

Условие выравнивания
Требование, согласно которому данные должны быть выровнены в памяти по границам, естественным для данной архитектуры. Как правило — по границам слов.

[1] Точнее, при использовании адресации big-endian сами байты слова в памяти располагаются в обратном порядке: старший байт имеет меньший адрес (который и является адресом слова), а младший — имеет наибольший адрес (адрес слова + 3 для 32-разрядного слова). — *Примеч. ред.*

Упражнение

Компиляция с использованием загрузки и сохранения

Предположим, что переменная `h` связана с регистром `$s2`, а базовый адрес массива `A` хранится в регистре `$s3`. Какой код MIPS-ассемблера будет создан для следующей инструкции присваивания на языке C?

```
A[12] = h + A[8];
```

Ответ

Хотя в C-инструкции прописана только одна операция, теперь два операнда находятся в памяти, поэтому нам нужны дополнительные MIPS-инструкции. Первые две инструкции такие же, как и в предыдущем примере, за исключением того, что на этот раз в инструкции загрузки слова для выбора `A[8]` используется правильное смещение для байтовой адресации и инструкция `add` помещает сумму в регистр `$t0`:

```
lw   $t0,32($s3)  # Временный регистр $t0 получает A[8]
add  $t0,$s2,$t0  # Временный регистр $t0 получает h + A[8]
```

Заключительная инструкция сохраняет сумму в `A[12]`, используя в качестве смещения 48 (4 × 12) и в качестве базового регистра `$s3`.

```
sw   $t0,48($s3)  # Сохранение h + A[8] в A[12]
```

Интерфейс аппаратного и программного обеспечения

В MIPS-архитектуре «загрузить слово» и «сохранить слово» являются инструкциями, копирующими слова между памятью и регистрами. Другие разновидности компьютеров для переноса данных наряду с загрузкой и сохранением используют и другие инструкции. Такие альтернативные инструкции имеются в архитектуре Intel x86, рассматриваемой в разделе 2.17. У многих программ переменных больше, чем регистров у компьютера. Поэтому компилятор сохраняет наиболее часто используемые переменные в регистрах и помещает все остальные в память, используя загрузку и сохранение для перемещения переменных между регистрами и памятью. Процесс размещения часто используемых переменных (или тех, которые понадобятся позже) в память называется *откачкой* (*spilling*) регистров.

Принцип создания оборудования, касающийся размера и скорости, подсказывает, что память должна работать медленнее регистров, поскольку регистров значительно меньше. Именно так все и обстоит: доступ к данным осуществляется быстрее, если они находятся не в памяти, а в регистрах.

Более того, данными, находящимися в регистрах, легче воспользоваться. Арифметическая инструкция MIPS может считывать данные из двух регистров, обрабатывать их и записывать результат. Инструкция переноса данных MIPS только лишь читает один операнд или записывает один операнд, не осуществляя его обработку.

Таким образом, доступ к регистрам занимает меньше времени, и работа с ними более производительна по сравнению с работой с памятью, поэтому данные в регистрах быстрее в получении и проще в использовании. Доступ к регистрам требует также меньше энергии, чем доступ к памяти. Для достижения наивысшей производительности и экономичности компиляторы должны эффективно использовать регистры.

Константы или непосредственные операнды

Использование констант в программе носит массовый характер, они, к примеру, применяются при увеличении индекса для указания на следующий элемент массива. Фактически при выполнении набора контрольных задач SPEC2006 более половины арифметических инструкций MIPS имеют в качестве операнда константу.

Задействуя только те инструкции, которые нам попадались до сих пор, для использования какой-нибудь константы нам следует загрузить ее из памяти. (Константа должна быть помещена в память при загрузке программы.) Например, чтобы прибавить константу 4 к регистру $s3, можно воспользоваться следующим кодом:

```
lw $t0, AddrConstant4($s1)  # $t0 = константе 4
add $s3,$s3,$t0             # $s3 = $s3 + $t0 ($t0 == 4)
```

предполагая, что $s1 + AddrConstant4 является адресом памяти для константы 4.

Альтернативный вариант, исключающий использование инструкции загрузки, заключается в использовании предлагаемых версий арифметических инструкций, в которых один операнд является константой. Эта быстродействующая инструкция сложения с одним операндом-константой называется *непосредственным сложением* (*add immediate*) или addi. Для прибавления 4 к регистру $s3 нужно просто написать:

```
addi $s3,$s3,4 # $s3 = $s3 + 4
```

Непосредственные инструкции иллюстрируют третий принцип конструирования, впервые упомянутый в разделе «Заблуждения и недоразумения» главы 1:

Принцип конструирования 3: часто встречающиеся задачи должны выполняться быстрее.

Операнды-константы встречаются довольно часто, и за счет включения констант в арифметические инструкции операции проводятся намного быстрее и используют меньше энергии, чем в том случае, когда константы загружаются из памяти.

Константа нуль предназначена для другой цели — упрощения набора инструкций за счет предоставления полезных изменений. Например, операция перемещения представляет собой всего лишь инструкцию сложения, где один из операндов равен нулю. Поэтому в MIPS выделен регистр $zero с нулевым значением, реализованным на аппаратном уровне. (Наверное, тот факт, что этот регистр имеет номер нуль, не станет для вас неожиданностью.)

Самопроверка

Говоря о важности регистров, как можно оценить темпы увеличения количества регистров на чипе?

1. Как очень высокие: их количество увеличивалось согласно закону Мура, который предсказал удвоение количества транзисторов на чипе каждые 18 месяцев.
2. Как очень низкие: поскольку программы обычно распространяются на языке компьютера, архитектура набора команд обладает высокой инерционностью, и поэтому количество регистров увеличивается только с вводом в действие нового набора инструкций.

Уточнение. Хотя регистры MIPS в этой книге представлены как 32-разрядные, существует и 64-разрядная версия набора инструкций MIPS, имеющая тридцать два 64-разрядных регистра. Чтобы не создавать путаницы, официально разные наборы называются MIPS-32 и MIPS-64. В данной главе будет использоваться поднабор MIPS-32.

MIPS-адресация, использующая базовый регистр плюс смещение, отлично подходит как для структур, так и для массивов, поскольку регистр может указать на начало структуры, а смещение может выбрать нужный элемент. Пример будет показан в разделе 2.13.

Изначально использование регистра в инструкциях переноса данных было придумано для хранения индекса массива со смещением, использующимся в качестве начального адреса массива. Поэтому базовый регистр также называется индексным регистром. В настоящее время объемы памяти существенно возросли, и программная модель размещения данных стала более сложной, поэтому базовый адрес массива обычно передается в регистре, поскольку, как мы увидим далее, в смещении он не поместится.

Так как MIPS поддерживает отрицательные константы, в MIPS-инструкции непосредственного вычитания нет необходимости.

2.4. Числа со знаком и без знака

Сначала давайте рассмотрим вкратце, как компьютер представляет числа. Люди учатся пользоваться десятичными числами, то есть числами по основанию 10, но числа могут быть представлены и с использованием любого другого основания. Например, 123 по основанию 10 = 1111011 по основанию 2.

Числа хранятся в компьютерном оборудовании в виде серии электронных сигналов высокого и низкого уровня, поэтому они рассматриваются как числа по основанию 2. (Точно так же, как числа по основанию 10 называются *десятичными*, числа по основанию 2 называются *двоичными*.)

Отдельная цифра двоичного числа является, таким образом, «атомом» вычислительной техники, поскольку вся информация составляется из **двоичных цифр** (binary digits) или *битов* (*bits*). Этот основополагающий строительный блок может иметь одно из двух значений, которые могут рассматриваться в качестве нескольких альтернативных вариантов: высокий или низкий, включено или выключено, правда или ложь, 1 или 0.

Обобщая вышеизложенное, при любом основании числа значение i-той цифры d равно

$$d \times Основание^{i}$$

где i начинается с нуля и увеличивается справа налево. Благодаря этому получается вполне очевидный способ нумерации битов в слове: нужно просто использовать для этого бита сте-

Двоичная цифра
Называется также двоичным битом. Одна из двух цифр (0 или 1) в представлении чисел по основанию 2. Является составляющей в представлении информации.

пень основания. Для десятичных чисел мы будем использовать нижний индекс 10, а для двоичных — 2. Например,

1011_2

представляет собой

$$(1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0)_{10}$$
$$= (1 \times 8) + (0 \times 4) + (1 \times 2) + (1 \times 1)_{10}$$
$$= (8 + 0 + 2 + 1)_{10}$$
$$= 11_{10}$$

Мы нумеруем биты 0, 1, 2, 3, ... в слове *справа налево*. Расположенный ниже рисунок показывает нумерацию битов в MIPS-слове и содержит число 1011_2:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1

(32 разряда)

Поскольку слова рисуются не только по горизонтали, но и по вертикали, термины «самый левый» и «самый правый» могут быть непонятными. Поэтому фраза **самый младший бит** (или разряд) используется для ссылки на самый правый разряд (бит под номером 0), а фраза **самый старший бит** (или разряд) используется для ссылки на самый левый разряд (бит под номером 31).

Слово в MIPS имеет длину 32 разряда, поэтому с его помощью можно представить 2^{32} различных 32-разрядных сочетаний. Вполне естественно позволить этим комбинациям представить числа от 0 до $2^{32} \times 1$ ($4\ 294\ 967\ 295_{10}$):

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_2 = 0_{10}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_2 = 1_{10}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_2 = 2_{10}$

. . . . . .

$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_2 = 4\ 294\ 967\ 293_{10}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_2 = 4\ 294\ 967\ 294_{10}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_2 = 4\ 294\ 967\ 295_{10}$

То есть 32-разрядные двоичные числа могут быть представлены в понятиях двоичного значения, умноженного на степень числа 2 (здесь *xi* означает *i*-тый бит числа *x*):

$$(x31 \times 2^{31}) + (x30 \times 2^{30}) + (x29 \times 2^{29}) + \ldots + (x1 \times 2^1) + (x0 \times 2^0)$$

Следует помнить, что комбинации двоичных разрядов, представленные выше, это всего лишь *представления* чисел. На самом деле числа имеют бесконечное количество цифр, почти все из которых представлены нулем, за исключением всего нескольких самых правых цифр. Просто обычно лидирующие нули не показываются.

Младший бит
Самый правый разряд MIPS-слова.

Старший бит
Самый левый разряд MIPS-слова.

Может быть разработано оборудование, предназначенное для сложения, вычитания, умножения и деления этих комбинаций двоичных разрядов. Если число, являющееся результатом таких операций, не может быть представлено

этими самыми правыми разрядами оборудования, то говорят, что возникло *переполнение*. Язык программирования, операционная система и программа должны определить, что нужно делать в случае переполнения.

Компьютерные программы ведут вычисления как с положительными, так и с отрицательными числами, поэтому необходимо представление, позволяющее отличить положительное число от отрицательного. Наиболее очевидное решение состоит в добавлении отдельного знака, который легко может быть представлен отдельным разрядом; такое представление называется *знаком и абсолютным значением*.

Увы, у знака и абсолютного значения есть несколько недостатков. Во-первых, непонятно, куда ставить знаковый разряд. Справа? Слева? На самых первых компьютерах пробовали и то и другое. Во-вторых, сумматоры для знака и абсолютного значения могли требовать *дополнительного шага для установки знака, поскольку* заранее не может быть известно, каким будет знак. И наконец, отдельный знаковый разряд означает, что знак и абсолютное значение имеют как положительный, так и отрицательный нуль, что вызывает проблемы у невнимательных программистов. В результате всех этих недостатков от знака и абсолютного значения вскоре отказались.

В поисках более подходящих вариантов возник вопрос, каким будет результат для беззнаковых чисел, если попытаться вычесть большое число из небольшого. Ответ состоит в том, что будет предпринята попытка занять разряд в строке с лидирующими нулями, и в результате получится строка с лидирующими единицами.

С учетом того, что лучших вариантов не предвиделось, окончательное решение свелось к выбору представления, которое упрощало бы конструкцию оборудования: лидирующие нули означали положительное число, а лидирующие единицы означали отрицательное число. Это соглашение для представления двоичных чисел со знаком называется представлением с *дополнением до двух*:

```
0000 0000 0000 0000 0000 0000 0000 0000₂ = 0₁₀
0000 0000 0000 0000 0000 0000 0000 0001₂ = 1₁₀
0000 0000 0000 0000 0000 0000 0000 0010₂ = 2₁₀

0111 1111 1111 1111 1111 1111 1111 1101₂ = 2 147 483 645₁₀
0111 1111 1111 1111 1111 1111 1111 1110₂ = 2 147 483 646₁₀
0111 1111 1111 1111 1111 1111 1111 1111₂ = 2 147 483 647₁₀
1000 0000 0000 0000 0000 0000 0000 0000₂ = -2 147 483 648₁₀
1000 0000 0000 0000 0000 0000 0000 0001₂ = -2 147 483 647₁₀
1000 0000 0000 0000 0000 0000 0000 0010₂ = -2 147 483 646₁₀

1111 1111 1111 1111 1111 1111 1111 1101₂ = -3₁₀
1111 1111 1111 1111 1111 1111 1111 1110₂ = -2₁₀
1111 1111 1111 1111 1111 1111 1111 1101₂ = -1₁₀
```

Положительная половина чисел, от 0 до 2 147 483 647$_{10}$ ($2^{31} \times 1$), использует такое же представление, что и раньше. Следующая комбинация разрядов (1000 ... 0000$_2$) представляет самое большое отрицательное число $-2\ 147\ 483\ 648_{10}$ (-2^{31}). За ним следует уменьшающийся набор отрицательных чисел: $-2\ 147\ 483\ 647_{10}$ ($1000 \ldots 0001_2$) до -1_{10} ($1111 \ldots 1111_2$).

У двоичного дополнения есть одно отрицательное число, $-2\,147\,483\,648_{10}$, у которого нет соответствующего положительного числа. Подобный дисбаланс также был бедой невнимательного программиста, но использование знака и абсолютного значения создавало проблемы как для программиста, так и для разработчика оборудования. В результате всего этого каждый современный компьютер использует для представления чисел со знаками дополнение до двух.

Представление с дополнением до двух имеет то преимущество, что все отрицательные числа содержат единицу в старшем разряде. Следовательно, оборудованию приходится проверять только этот бит, чтобы узнать, отрицательное данное число или положительное (число нуль считается положительным). Этот разряд часто называют *знаковым разрядом*. Понимая назначение знакового разряда, мы можем представить положительные и отрицательные 32-разрядные числа в понятиях значения разряда, умноженного на степень числа два:

$$(x^{31} \times -2^{31}) + (x^{30} \times 2^{30}) + (x^{29} \times 2^{29}) + \ldots + (x^{1} \times 2^{1}) + (x^{0} \times 2^{0})$$

Знаковый разряд умножается на -2^{31}, а затем все остальные разряды умножаются на положительные версии соответствующих им значений по основанию.

Упражнение

Перевод двоичных чисел в десятичные

Каково десятичное значение этого 32-разрядного числа с дополнением до двух?

$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_{2}$

Ответ

Подставим числовые значения разрядов в показанную выше формулу:

$(1 \times -2^{31}) + (1 \times 2^{30}) + (1 \times 2^{29}) + \ldots + (1 \times 2^{2}) + (0 \times 2^{1}) + (0 \times 2^{0})$
$= -2^{31} + 2^{30} + 2^{29} + \ldots + 2^{2} + 0 + 0 = -2\,147\,483\,648_{10} + 2\,147\,483\,644_{10}$
$= -4_{10}$

Скоро мы увидим более короткий прием, упрощающий перевод отрицательных чисел в положительные.

Точно так же как и операции с беззнаковыми числами могут при отображении результата переполнить емкость, предоставляемую оборудованием, операции с числами с дополнением до двух могут создать такую же проблему. Переполнение возникает в том случае, когда самый старший сохраняемый разряд в комбинации двоичных разрядов не совпадает с бесконечными числами слева (то есть знаковый разряд имеет неправильное значение): нуль в крайнем левом разряде комбинации, когда число отрицательное, или единица, когда число положительное.

Интерфейс аппаратного и программного обеспечения

В отличие от только что рассмотренных чисел адресация памяти начинается с нуля и продолжается до самого большого адреса. Иначе говоря, отрицательные адреса не имеют смысла. Поэтому программам нужно работать иногда с числами, которые

могут быть положительными или отрицательными, а иногда с числами, которые могут быть только положительными. В некоторых языках программирования это различие находит специальное отражение. Например, в C первые называют целыми числами — *integers* (объявляются в программе с помощью ключевого слова int), а последние беззнаковыми целыми числами — *unsigned integers* (unsigned int). Некоторые руководства по стилям программирования на C даже рекомендуют объявлять первые как signed int, чтобы отличие было более заметным.

Давайте рассмотрим два полезных более коротких приема, которые используются при работе с числами с дополнением до двух. Первый прием — инвертирование числа с дополнением до двух. Нужно просто заменить каждый нуль на единицу, а каждую единицу на нуль, а затем прибавить к результату единицу. Этот короткий прием основан на том, что сумма числа и его инвертированного представления должна быть равна 111 ... 111_2, что является представлением числа –1. Поскольку значит, или .

Упражнение

Короткий прием инвертирования

Измените знак числа 2_{10} на отрицательный, а затем проверьте результат, изменив знак числа -2_{10}.

Ответ

2_{10} = 0000 0000 0000 0000 0000 0000 0000 0010_2

Инвертируйте это число путем инвертирования битов и прибавления единицы:

```
    1111 1111 1111 1111 1111 1111 1111 1101₂
  +                                       1₂
  ---------------------------------------------
  = 1111 1111 1111 1111 1111 1111 1111 1110₂
  = -2₁₀
```

А теперь проделайте все в обратном направлении

1111 1111 1111 1111 1111 1111 1111 1110_2.

Сначала инвертировав разряды, а потом прибавив единицу:

```
    0000 0000 0000 0000 0000 0000 0000 0001₂
  +                                       1₂
  ---------------------------------------------
  = 0000 0000 0000 0000 0000 0000 0000 0010₂
    = 2₁₀
```

Наш следующий прием научит конвертировать двоичное число, представленное в n разрядах, в число, представленное более чем в n разрядах. Например, поле непосредственного значения в инструкциях загрузки, сохранения, ветвления, сложения и установки бита, если меньше чем» содержит 16-разрядное число с дополнением до двух, которое представляет числа от $-32\,768_{10}$ (-2^{15}) до $32\,767_{10}$ ($2^{15} \times 1$). Чтобы прибавить поле непосредственного значения к 32-разрядному регистру, компьютер должен преобразовать 16-разрядное число в его 32-разрядный эквивалент. Короткий прием заключается в извлечении старшего разряда из более мелкой

величины — знакового разряда — и в копировании его для заполнения новых разрядов более крупной величины. Старые разряды просто копируются в правую часть нового слова. Этот короткий прием часто называется *расширением знака*.

Упражнение

Прием расширения знака
Преобразуйте 16-разрядную версию чисел 2_{10} и -2_{10} в 32-разрядные двоичные числа.

Ответ

16-разрядная двоичная версия числа 2 имеет следующий вид:

$0000\ 0000\ 0000\ 0010_2 = 2_{10}$

Преобразование в 32-разрядное число осуществляется путем создания 16 копий значений старшего разряда (0) и помещения их в левую половину слова. Правая половина слова получает старое значение:

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_2 = 2_{10}$

Теперь инвертируем 16-разрядную версию числа 2, используя первый короткий прием. Итак,

$0000\ 0000\ 0000\ 0010_2$

превращается в

$$\begin{array}{r} 1111\ 1111\ 1111\ 1101_2 \\ +\qquad\qquad\quad 1_2 \\ \hline = 1111\ 1111\ 1111\ 1110_2 \end{array}$$

Создание 32-разрядной версии отрицательного числа означает копирование знакового разряда 16 раз и помещение результата в левую часть:

$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_2 = -2_{10}$

Этот прием работает благодаря тому, что положительные числа с дополнением до двух в левой части действительно содержат бесконечные нули, а отрицательные числа с дополнением до двух имеют в левой части бесконечную череду единиц. Комбинация двоичных разрядов, представляющая число, скрывает лидирующие разряды, вмещая число в количество разрядов, предоставляемых оборудованием; расширение знака просто восстанавливает часть этих разрядов.

Краткие выводы

Главное, на что обращалось внимание в данном разделе, заключается в том, что в компьютерном мире нам нужно представление как положительных, так и отрицательных целых чисел, и, несмотря на все «за» и «против» различных вариантов, с 1965 года стало преобладать дополнение до двух.

Самопроверка

Каким будет десятичное значение этого 64-разрядного числа с дополнением до двух?

$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1000_2$

1. -4_{10}
2. -8_{10}
3. -16_{10}
4. $18\,446\,744\,073\,709\,551\,609_{10}$

Уточнение. Дополнение до двух получило свое название от правила, что беззнаковая сумма n-разрядного числа и его отрицательного значения равна 2^n; поэтому дополнение или отрицательное значение числа x с дополнением до двух равно $2^n - x$.

Третий вариант представления, составляющий компанию дополнению до двух и знаку и абсолютному значению, называется **дополнением до единицы**. Отрицательное значение при дополнении до единицы определяется инвертированием каждого разряда из нуля в единицу и из единицы в нуль, что помогает объяснить его название, поскольку дополнение числа x равно $2^n - x - 1$. Этот вариант также претендовал на решение, более подходящее, чем знак и абсолютное значение, и на нескольких ранних компьютерах для научных расчетов использовалась именно эта система записи. Это представление похоже на представление с дополнением до двух, за исключением того, что у него также имеется два нуля: $00\ldots00_2$ в качестве положительного нуля и $11\ldots11_2$ в качестве отрицательного нуля. Наибольшее отрицательное число, $10\ldots000_2$, представляет собой $-2\,147\,483\,647_{10}$, и положительные и отрицательные числа сбалансированы. Сумматоры чисел с дополнением до единицы нуждаются в дополнительном шаге для вычитания числа, и поэтому сегодня доминирующим представлением является дополнение до двух.

Последнем представлением, которое мы рассмотрим при обсуждении чисел с плавающей точкой в главе 3, будет форма представления самого отрицательного значения в виде $00\ldots000_2$, а самого положительного в виде $11\ldots11_2$, с нулем, имеющим значение $10\ldots00_2$. Эта форма называется **смещенным представлением**, поскольку она смещает число таким образом, чтобы число плюс смещение имело неотрицательное представление.

Уточнение. Для десятичных чисел со знаком мы используем «–» для представления отрицательного значения, поскольку ограничений на размер десятичного числа не существует. При фиксированном размере слова, двоичные и шестнадцатеричные (см. табл. 2.2) битовые строки могут кодировать знак; поэтому мы обычно не используем «+» или «–» с двоичной или шестнадцатеричной формой записи.

Дополнение до единицы
Система записи, представляющая самое большое отрицательное значение как $10\ldots000_2$, а самое большое положительное значение как $01\ldots11_2$, сохраняет баланс положительных и отрицательных чисел, но при этом имеет два нуля, один положительный ($00\ldots00_2$) и один отрицательный ($11\ldots11_2$). Этот термин также используется для обозначения инвертирования каждого разряда в комбинации: 0 в 1 и 1 в 0.

Смещенное представление
Форма представления самого отрицательного значения в виде $00\ldots000_2$ и самого положительного значения в виде $11\ldots11_2$, при которой нуль обычно имеет значение $10\ldots00_2$, таким образом, смещение числа по принципу число плюс смещение имеет неотрицательное представление.

2.5. Представление инструкций в компьютере

Теперь мы готовы объяснить разницу между способом инструктирования компьютеров людьми и способом видения инструкций компьютерами.

Инструкции хранятся в компьютере в виде последовательности электронных сигналов высокого и низкого уровня и могут быть представлены в виде чисел. Фактически любая часть инструкции может рассматриваться как отдельное число, и такие числа, помещенные рядом, и формируют инструкцию.

Поскольку ссылка на регистры присутствует чуть ли не в каждой инструкции, должно быть соглашение для отображения имен регистров на числа. В языке ассемблера MIPS регистры от `$s0` до `$s7` отображаются на регистры от 16 до 23, а регистры от `$t0` до `$t7` отображаются на регистры от 8 до 15. Поэтому `$s0` означает регистр 16, `$s1` означает регистр 17, `$s2` означает регистр 18, ..., `$t0` означает регистр 8, `$t1` означает регистр 9 и т. д. Соглашение для остальных 32 регистров будет рассмотрено в следующих разделах.

Упражнение

Трансляция инструкций ассемблера MIPS в инструкции машины

Давайте в качестве примера сделаем следующий шаг в освоении языка MIPS. Мы покажем версию инструкции настоящего языка MIPS, выраженную в символьном виде

```
add $t0,$s1,$s2
```

сначала как комбинацию десятичных чисел, а затем как комбинацию двоичных чисел.

Ответ

Десятичное представление имеет следующий вид:

0	17	18	8	0	32

Каждая из этих частей инструкции называется *полем*. Первое и последнее поля (которые в данном случае содержат 0 и 32) в комбинации сообщают MIPS-компьютеру, что эта инструкция выполняет сложение. Второе поле предоставляет номер регистра, являющегося для операции сложения первым операндом-источником (17 = `$s1`), а третье поле предоставляет для сложения второй операнд-источник (18 = `$s2`). Четвертое поле содержит номер регистра, получающего сумму (8 = `$t0`). Пятое поле в данной инструкции не используется, поэтому оно установлено в нуль. Таким образом, данная инструкция складывает регистр `$s1` с регистром `$s2` и помещает сумму в регистр `$t0`.

Эта инструкция может быть также представлена в виде полей не десятичных, а двоичных чисел:

000000	10001	10010	01000	00000	100000
6 разрядов	5 разрядов	5 разрядов	5 разрядов	5 разрядов	6 разрядов

Эта компоновка инструкции называется **форматом инструкции**. Путем подсчета разрядов можно определить, что эта MIPS-инструкция занимает в точности 32 разряда — размер, соответствующий слову данных. Исходя из нашего конструкторского принципа, гласящего о том, что простота предпочитает постоянство, все MIPS-инструкции имеют длину 32 разряда.

Чтобы отличить двоичную форму записи от языка ассемблера числовую версию инструкций называют **машинным языком**, а последовательность таких инструкций *машинным кодом*.

Может сложиться впечатление, что теперь придется читать и записывать длинные и неудобные двоичные числа. Избежать этого поможет применение чисел, имеющих большее основание, которые легко конвертируются в двоичные числа. Поскольку почти все размеры компьютерных данных кратны четырем, популярность получили **шестнадцатеричные числа** (по основанию 16). Поскольку основание 16 является степенью числа 2, мы можем просто проводить преобразование, заменяя каждую группу из четырех двоичных цифр одной шестнадцатеричной цифрой, и наоборот. В табл. 2.2 показан порядок преобразования одних чисел в другие.

Таблица 2.2. Таблица преобразования шестнадцатеричных и двоичных чисел

16	2	16	2	16	2	16	2
0_{16}	0000_2	4_{16}	0100_2	8_{16}	1000_2	c_{16}	1100_2
1_{16}	0001_2	5_{16}	0101_2	9_{16}	1001_2	d_{16}	1101_2
2_{16}	0010_2	6_{16}	0110_2	a_{16}	1010_2	e_{16}	1110_2
3_{16}	0011_2	7_{16}	0111_2	b_{16}	1011_2	f_{16}	1111_2

Легко просто заменить одну шестнадцатеричную цифру соответствующими ей четырьмя двоичными цифрами и наоборот. Если длина двоичного числа не делится на четыре, осуществляйте преобразование справа налево.

Поскольку довольно часто приходится иметь дело с различными основаниями чисел, чтобы не запутаться, для десятичных чисел будет применяться нижний индекс *10*, для двоичных чисел — нижний индекс *2*, а для шестнадцатеричных чисел — нижний индекс *16*. (При отсутствии нижнего индекса по умолчанию будет считаться, что число имеет основание 10.) Кстати, для шестнадцатеричных чисел в C и Java используется форма записи 0x*nnnn*.

Формат инструкции
Форма представления инструкции, состоящая из полей двоичных чисел.

Машинный язык
Двоичное представление, используемое для передачи информации внутри компьютерной системы.

Шестнадцатеричные числа
Числа по основанию 16.

Упражнение

Перевод двоичных чисел в шестнадцатиричные и обратно

Переведем следующие шестнадцатеричные и двоичные числа в числа с другими основаниями:

$eca8\ 64201_6$
$0001\ 0011\ 0101\ 0111\ 1001\ 1011\ 1101\ 1111_2$

Ответ

Для ответа достаточно посмотреть в табл. 2.2 в одном направлении:

eca8 6420 $_{16}$

1110 1100 1010 1000 0110 0100 0010 0000 $_2$

А затем в другом направлении:

0001 0011 0101 0111 1001 1011 1101 1111 $_2$

1357 9bdf $_{16}$

Поля MIPS

Чтобы их проще было рассматривать, полям MIPS присвоили имена:

op	rs	rt	rd	shamt	funct
6 разрядов	5 разрядов	5 разрядов	5 разрядов	5 разрядов	6 разрядов

Имя каждого поля MIPS-инструкций имеет следующее значение:

- *op*: основная операция инструкции, традиционно именуемая **opcode** (код операции).
- *rs*: первый регистр операнда-источника.
- *rt*: второй регистр операнда-источника.
- *rd*: регистр операнда-приемника. Он получает результат операции.
- *shamt*: величина сдвига (shift amount). (Этот термин и инструкции сдвига рассматриваются в разделе 2.6; до сих пор это поле не использовалось, поэтому в данном разделе оно имеет нулевое значение.)
- *funct*: функция. Это поле часто называют *кодом функции*, оно выбирает специфический вариант операции, указанной в поле op.

Когда инструкции нужно более длинное поле, чем то, которое было показано выше, возникает проблема. Например, инструкция загрузки слова должна указать два регистра и константу. Если адрес в показанном выше формате должен был использовать одно из 5-разрядных полей, то константа, входящая в инструкцию загрузки слова, должна быть ограничена только 2^5 или числом 32. Эта константа используется для выбора

Opcode
Поле, обозначающее операцию и формат инструкции.

элементов из массива или структуры данных, и зачастую она должна быть больше числа 32. Это 5-разрядное поле слишком мало для этих целей.

Получается, что у нас возник конфликт между желанием сохранить для всех инструкций одну и ту же длину и желанием иметь единый формат инструкции. Это подводит нас к заключительному принципу конструирования оборудования:

Принцип конструирования 4: хорошая конструкция требует удачных компромиссов.

Компромисс, выбранный конструкторами MIPS-систем, заключается в сохранении единой длины для всех инструкций, что потребовало различных видов форматов инструкций для разных типов инструкций. Например, формат, показанный выше, называется *R-типом* (для регистра), или *R-форматом*. Второй тип формата инструкции называется *I-типом* (для непосредственных данных – immediate), или *I-форматом*, и он используется для инструкций с непосредственными данными и инструкций переноса данных. Поля I-формата имеют следующий вид:

op	rs	rt	Константа или адрес
6 разрядов	5 разрядов	5 разрядов	16 разрядов

16-разрядный адрес означает, что инструкция загрузки слова может загрузить любое слово из области $\pm 2^{15}$ или 32 768 байт ($\pm 2^{13}$ или 8192 слов) с адресом, указанным в базовом регистре rs. Точно так же непосредственное сложение ограничено константой, имеющей значение в диапазоне $\pm 2^{15}$. Понятно, что в данном формате использование более 32 регистров потребовало бы еще одного бита для полей rs и rt, затрудняя размещение всего необходимого в одном слове.

Посмотрим на инструкцию загрузки слова из предыдущего упражнения по компиляции присваивания, когда операнд находится в памяти:

```
lw $t0,8($s3)  # Временный регистр $t0 получает A[8]
```

Здесь число 19 (для регистра $s3) помещается в поле rs, число 8 (для регистра $t0) помещается в поле rt, а число 32 помещается в поле адреса. Заметьте, что значение поля rt для этой инструкции изменилось: в инструкции загрузки слова поле rt указывает на регистр назначения, который получает результат загрузки.

Хотя использование нескольких форматов усложняет конструкцию оборудования, сложность можно снизить за счет схожести форматов. Например, первые три поля форматов R-типа и I-типа имеют одинаковые размеры и имена; длина четвертого поля в I-типе равна сумме длин последних трех полей в R-типе.

На тот случай, если у вас возник такой вопрос, форматы различаются за счет значения, находящегося в первом поле: каждому формату определяется особый набор значений для первого поля (op), поэтому оборудование знает, как нужно рассматривать последнюю половину инструкции: как три поля (R-тип) или как одно поле (I-тип). В табл. 2.3 показаны числа, используемые в каждом поле для рассматриваемых здесь MIPS-инструкций.

Таблица 2.3. Кодировка инструкций MIPS. В этой таблице «reg» означает номера регистров в диапазоне от 0 до 31, «address» означает 16-разрядный адрес, а «n.a.» (not applicable) означает, что это поле не появляется в данном формате. Заметьте, что инструкции add и sub имеют одинаковое значение в поле op; для определения варианта операции оборудование использует поле funct: сложение (32) или вычитание (34).

Инструкция	Format	op	rs	rt	rd	shamt	funct	address
add (сложение)	R	0	reg	reg	reg	0	32	n.a.
sub (вычитание)	R	0	reg	reg	reg	0	34_{10}	n.a.
add immediate (непосредственное сложение)	I	8_{10}	reg	reg	n.a.	n.a.	n.a.	константа
lw (загрузка слова)	I	35_{10}	reg	reg	n.a.	n.a.	n.a.	адрес
sw (сохранение слова)	I	43_{10}	reg	reg	n.a.	n.a.	n.a.	адрес

Упражнение

Трансляция языка асемблера MIPS в язык машины

Теперь мы можем взять полноценный пример того, что пишет программист, и того, что выполняет компьютер. Если в регистре `$t1` находится базовый адрес массива `A`, а регистр `$s2` соотносится с переменной `h`, инструкция присваивания

```
A[300] = h + A[300];
```

компилируется в

```
lw   $t0,1200($t1)  # Временный регистр $t0 получает A[300]
add  $t0,$s2,$t0    # Временный регистр $t0 получает h + A[300]
sw   $t0,1200($t1)  # Сохранение h + A[300] в A[300]
```

Каким будет код на машинном языке MIPS для этих трех инструкций?

Ответ

Чтобы упростить задачу, сначала представим инструкции на машинном языке с использованием десятичных чисел. Из табл. 2.3 можно определить три инструкции на машинном языке:

op	rs	rt	rd	address/shamt	funct
35	9	8	1200		
0	18	8	8	0	32
43	9	8	1200		

Инструкция `lw` индентифицируется числом 35 (см. табл. 2.3) в первом поле (`op`). Базовый регистр 9 (`$t1`) указан во втором поле (`rs`), а регистр назначения 8 (`$t0`) указан в третьем поле (`rt`). Смещение для выбора `A[300]` (1200 = 300 × 4) находится в последнем поле (`address`).

Следующая затем инструкция `add` указывается нулевым значением в первом поле (`op`) и числом 32 в последнем поле (`funct`). Три регистровых операнда (18, 8 и 8) находятся во втором, третьем и четвертом полях и соответствуют регистрам `$s2`, `$t0` и `$t0`.

Инструкция sw идентифицируется числом 43 в первом поле. Вся остальная часть этой последней инструкции идентична заключительной части инструкции lw.

Поскольку 1200_{10} = 0000 0100 1011 0000_2, двоичный эквивалент десятичной формы имеет следующий вид:

100011	01001	01000	0000 0100 1011 0000		
000000	10010	01000	01000	00000	100000
101011	01001	01000	0000 0100 1011 0000		

Обратите внимание на схожесть двоичного представления первой и последней инструкции. Единственное отличие заключается в выделенном в таблице третьем разряде слева.

Таблица 2.4 содержит свод той части машинного языка MIPS, которая рассматривалась в этом разделе. Как будет показано в главе 4, сходство двоичного представления родственных инструкций упрощает оборудование. Такие сходства служат еще одним примером постоянства в MIPS-архитектуре.

Машинный язык MIPS

Таблица 2.4. MIPS-архитектура, фигурировавшая в разделе 2.5. До сих пор рассматривались два формата MIPS-инструкций: R и I. Первые 16 разрядов у них имеют одинаковое предназначение: оба они содержат поле *op*, задающее базовую операцию; поле *rs*, задающее один из источников; и поле *rt*, задающее другой операнд-источник, за исключением инструкции загрузки слова, где в нем задается регистр назначения. R-формат делит последние 16 разрядов на поле *rd*, задающее регистр назначения; поле *shamt*, роль которого рассматривается в разделе 2.6; и поле *funct*, задающее конкретную операцию для инструкций R-формата. I-формат объединяет последние 16 разрядов в одно поле *address*.

Имя	Формат	Пример						Комментарии
add	R	0	18	19	17	0	32	add $s1,$s2,$s3
sub	R	0	18	19	17	0	34	sub $s1,$s2,$s3
addi	I	8	18	17	100			addi $s1,$s2,100
lw	I	35	18	17	100			lw $s1,100($s2)
sw	I	43	18	17	100			sw $s1,100($s2)
Размер поля		6 разрядов	5 разрядов	6 разрядов	5 разрядов	5 разрядов	6 разрядов	Длина всех MIPS-инструкций 32 разряда
R-формат	R	op	rs	rt	rd	shamt	funct	Формат арифметической инструкции
I-формат	I	op	rs	rt	адрес			Формат инструкций переноса данных

Самое важное

Современные компьютеры построены на двух ключевых принципах:

1. Инструкции представляются числами.
2. Программы хранятся в памяти, чтобы их можно было считывать или записывать как числа.

Эти принципы привели к концепции *программы, хранящейся в памяти*, изобретение которой выпустило компьютерного джинна из бутылки. На рис. 2.3 показаны возможности этой концепции. В частности, в памяти могут содержаться исходный код программы редактирования, соответствующий ему скомпилированный машинный код для программы редактирования, текст, используемый скомпилированной программой, и даже компилятор, генерирующий машинный код.

Последствием того, что инструкции представлены в виде чисел, является тот факт, что программы часто поставляются в виде файлов двоичных чисел. Коммерческое значение этого факта заключается в том, что компьютеры могут унаследовать готовое программное обеспечение, если они совместимы с существующим набором команд. Такая «совместимость на уровне двоичных кодов» часто вынуждает промышленность ограничиваться небольшим количеством архитектур набора инструкций.

Рис. 2.3. Концепция программы, хранящейся в памяти. Хранение программ в памяти позволяет компьютеру, выполняющему бухгалтерскую задачу, в мгновение ока стать компьютером, помогающим автору писать эту книгу. Это переключение происходит путем простой загрузки в память программ и данных с последующей командой компьютеру приступить к выполнению кода с заданного места. Обработка инструкций одинаковыми способами существенно упрощает как устройство памяти, так и программное обеспечение компьютерных систем. В частности, технологии памяти, необходимой для данных, могут быть также использованы для программ, а программы, например компиляторы, могут транслировать код, записанный в том виде, который наиболее удобен для человека, в код, понятный компьютеру

Самопроверка

Какая из MIPS-инструкций здесь представлена? Выберите один из четырех вариантов, показанных ниже.

op	rs	rt	rd	shamt	funct
0	8	9	10	0	34

1. add $s0, $s1, $s2
2. add $s2, $s0, $s1
3. add $s2, $s1, $s0
4. sub $t2, $t0, $t1

2.6. Логические операции

> Напротив, — продолжил Труляля. — Если бы это было так, это бы ещё ничего. Если бы, конечно, оно так и было. Но так как это не так, так оно и не этак. Такова логика вещей!
>
> *Льюис Кэрролл.*
> *Приключения Алисы в Стране чудес*

Хотя первые компьютеры работали с целыми словами, вскоре стало понятно, что было бы неплохо работать с полями разрядов внутри слова или даже с отдельными разрядами. Одним из примеров такой операции является анализ символов внутри слова, каждый из которых хранится в формате восьми разрядов (см. раздел 2.9). Это привело к тому, что такие операции были добавлены к языкам программирования и архитектурам наборов инструкций для того, чтобы кроме всего прочего упростить запаковку и распаковку битов внутри слов. Эти инструкции назвали логическими операциями. В табл. 2.5 показаны логические операции в C, Java и MIPS.

Таблица 2.5. Логические операции в языках C и Java и соответствующие им инструкции MIPS. MIPS реализует операцию НЕ, используя операцию НЕ ИЛИ с одним нулевым операндом

Логические операции	Операторы языка C	Операторы языка Java	Инструкции MIPS
Сдвиг влево	<<	<<	sll
Сдвиг вправо	>>	>>>	srl
Поразрядное И	&	&	and, andi
Поразрядное ИЛИ	|	|	or, ori
Поразрядное НЕ	~	~	nor

Первый класс таких операторов называется *сдвигами* (*shifts*). Они перемещают все разряды в слове влево или вправо, заполняя пустые разряды нулями. Например, если регистр $s0 содержит

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1001_2 = 9_{10}$

и выполняется инструкция сдвига влево на 4 разряда, новым значением будет следующее число:

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1001\ 0000_2 = 144_{10}$

Парной для сдвига влево является операция сдвига вправо. Настоящие названия этих двух MIPS-инструкций сдвига *shift left logical* (`sll`), то есть логический сдвиг влево, и *shift right logical* (`srl`), логический сдвиг вправо. Следующая инструкция выполняет операцию, показанную выше, при том условии, что исходное значение было помещено в регистр `$s0`, а результат будет помещен в регистр `$t2`:

```
sll $t2,$s0,4        # reg $t2 = reg $s0 << 4 разряда
```

Мы отложили рассмотрение поля `shamt` в R-формате. Оно предназначено для значения сдвига, используемого в соответствующих инструкциях. Таким образом, версия показанной ранее инструкции на машинном языке имеет следующий вид:

op	rs	rt	rd	shamt	funct
0	0	16	10	4	0

Кодировка инструкции `sll` представлена нулями как в поле `op`, так и в поле `funct`, поле `rd` содержит число 10 (регистр `$t2`), `rt` содержит число 16 (регистр `$s0`), а поле `shamt` содержит число 4. Поле `rs` не используется и поэтому имеет нулевое значение.

Инструкция логического сдвига влево имеет дополнительное преимущество. Сдвиг влево на i разрядов даст такой же результат, как и умножение на 2^i, точно также как сдвиг разрядов десятичного числа на i позиций эквивалентен умножению на 10^i. Например, показанная ранее инструкция `sll` осуществила сдвиг на 4 разряда, что дает такой же результат, что и умножение на 2^4 или на 16. Первая комбинация разрядов, показанная выше, представляла число 9, а $9 \times 16 = 144$, что и является значением второй комбинации разрядов.

Другой полезной операцией, изолирующей поля, является операция **И** (**AND**). (Ее название написано большой буквой, чтобы не перепутать его с простым языковым союзом.) Операция И является поразрядной и оставляет 1 в выдаваемом результате только в том случае, если оба разряда операндов содержали 1. Например, если регистр `$t2` содержит

$0000\ 0000\ 0000\ 0000\ 0000\ 1101\ 1100\ 0000_2$,

а регистр `$t1` содержит

$0000\ 0000\ 0000\ 0000\ 0011\ 1100\ 0000\ 0000_2$,

то после выполнения MIPS-инструкции

```
and $t0,$t1,$t2        # reg $t0 = reg $t1 & reg $t2
```

значением регистра `$t0` станет число

$0000\ 0000\ 0000\ 0000\ 0000\ 1100\ 0000\ 0000_2$

Как видите, операция И может применить комбинацию разрядов к набору разрядов, заставляя появляться нули там, где нуль стоит в разряде комбинации. Такая комбинация разрядов в сочетании с операцией И традиционно называется *маской*, поскольку, как настоящая маска, «скрывает» часть разрядов.

И
Логическая поразрядная операция, работающая с двумя операндами и вычисляющая 1 только в том случае, если 1 имеется в обоих операндах.

Чтобы поместить значение в одно из этих нулевых пространств, у операции И есть напарник, называемый операцией **ИЛИ (OR)**. Она представляет собой поразрядную операцию, которая помещает 1 в результат, если разряд любого из операндов имел значение 1. Для уточнения, если регистры `$t1` и `$t2` из предыдущего примера не претерпели изменений, то результат MIPS-инструкции

```
or $t0,$t1,$t2        # reg $t0 = reg $t1 | reg $t2
```

будет представлять собой значение, помещаемое в регистр `$t0`:

$0000\ 0000\ 0000\ 0000\ 0011\ 1101\ 1100\ 0000_2$

И заключительной логической операцией будет отрицание. Операция **НЕ** использует один операнд и помещает 1 в результат, если данный разряд операнда имеет значение 0, и наоборот. Чтобы сохранить трехоперандный формат, разработчики MIPS решили включить в набор инструкций вместо инструкции НЕ инструкцию **НЕ ИЛИ (NOR)**. Если один операнд будет нулевым, то эта инструкция будет эквивалентна инструкции НЕ: А НЕ ИЛИ 0 = НЕ (А ИЛИ 0) = НЕ (А).

Если регистр `$t1` из предыдущего примера не изменился, а регистр `$t3` имеет значение 0, результатом MIPS-инструкции

```
nor $t0,$t1,$t3       # reg $t0 = ~ (reg $t1 | reg $t3)
```

будет следующее значение регистра `$t0`:

$1111\ 1111\ 1111\ 1111\ 1100\ 0011\ 1111\ 1111_2$

В табл. 2.5. показана взаимосвязь между операторами языков C и Java и инструкциями MIPS. В логических операциях И и ИЛИ, точно так же как и в арифметических операциях, удобно применять константы, поэтому MIPS также предоставляет инструкции непосредственного И (`andi`) и непосредственного ИЛИ (`ori`). Использование констант не свойственно операции НЕ ИЛИ, поскольку ее основное назначение состоит в инверсии разрядов одиночного операнда; поэтому в архитектуре набора инструкций MIPS непосредственная версия этой операции не предусмотрена.

Уточнение. Полный набор инструкций MIPS также включает в себя операцию **исключающего ИЛИ (XOR)**, которая устанавливает разряд в 1, когда два соответствующих разряда отличаются друг от друга, и в 0, когда они одинаковы. Язык C позволяет полям битов или полям быть определенным внутри слов. Все поля должны помещаться внутри отдельного слова. Поля являются беззнаковыми целыми числами, которые могут быть настолько короткими, что представлять собой всего один разряд. Компилятор C вставляет и извлекает поля, используя имеющиеся в MIPS логические инструкции: `and`, `or`, `sll` и `srl`.

ИЛИ
Логическая поразрядная операция, работающая с двумя операндами и вычисляющая единицу, если единица имеется в каком-либо из операндов (или в обоих операндах).

НЕ
Логическая поразрядная операция, работающая с одним операндом и инвертирующая разряды; то есть она заменяет каждую единицу нулем и каждый нуль единицей.

НЕ ИЛИ
Логическая поразрядная операция, работающая с двумя операндами, которая вычисляет НЕ для двух операндов, над которыми произведена операция ИЛИ. То есть она вычисляет единицу только в том случае, если в соответствующем разряде обоих операндов находится нуль.

Самопроверка

Какие операции могут изолировать поле в слове?

1. И
2. Сдвиг влево, за которым следует сдвиг вправо.

2.7. Инструкции для принятия решения

> Польза от автоматического компьютера заключается в повторном использовании заданной последовательности инструкций, с количеством итераций, зависящим от результатов вычисления. ... Этот выбор может быть сделан в зависимости от знака числа (нуль для машины рассматривается как положительное число). В результате этого мы ввели [инструкцию] ([инструкцию] условного перехода), которая будет, в зависимости от знака заданного числа, заставлять выполняться одну нужную из двух процедур.
>
> *Баркс, Голдстин и фон Нейман*

Компьютер от простого калькулятора отличает возможность принимать решения. На основе входных данных и значений, созданных в процессе вычислений, выполняются разные инструкции. Принятие решений часто представлено в языках программирования использованием инструкции `if`, иногда в сочетании с инструкцией `go to` и метками. Язык ассемблера MIPS включает две инструкции принятия решений, подобные инструкции `if` с инструкцией перехода `go to`. Первая инструкция имеет формат:

```
beq регистр1, регистр2, L1
```

Эта инструкция предусматривает переход на инструкцию, имеющую метку *L1*, если значение в *регистре1* равно значению в *регистре2*. Мнемоника `beq` означает *branch if equal*, то есть ветвление при равенстве. Вторая инструкция имеет формат:

```
bne регистр1, регистр2, L1
```

Она означает переход на инструкцию, имеющую метку *L1*, если значение в *регистре1* не равно значению в *регистре2*. Мнемоника `bne` означает *branch if not equal*, то есть ветвление при неравенстве. Эти две инструкции традиционно называются **условными ветвлениями**.

Условное ветвление
Инструкция, требующая сравнения двух значений и допускающая последующую передачу управления на новый адрес в программе на основе результата сравнения.

Упражнение

Компиляция управляющей структуры if-then-else в условные ветвления

В следующем фрагменте кода `f`, `g`, `h`, `i` и `j` — переменные. Каким будет скомпилированный MIPS-код для этой инструкции `if` при условии, что пять переменных от `f` до `j` соответствуют пяти регистрам от `$s0` до `$s4`?

```
if (i == j) f = g + h; else f = g - h;
```

Ответ

На рис. 2.4 показана блок-схема того, что должен делать MIPS-код. Первое выражение сравнение на равенство, поэтому, похоже, что нам нужно провести ветвление, если значения регистров равны друг другу, то есть воспользоваться инструкцией beq. Вообще-то код был бы эффективнее, если бы проверялось противоположное условие, чтобы провести ветвление поверх кода, который выполняет следующую часть, then, инструкции if (метка Else определена ниже), и поэтому мы используем ветвление, если значения регистров не равны, то есть используем инструкцию bne:

```
bne $s3,$s4,Else        # перейти на Else, если i ≠ j
```

Следующая инструкция присваивания выполняет одну операцию, и если все операнды размещены в регистрах, то это просто одна инструкция:

```
add $s0,$s1,$s2         # f = g + h (пропускается, если i ≠ j)
```

Теперь нам нужно перейти в конец инструкции if. В этом примере представляется новый вид ветвления, часто называемый *безусловным ветвлением*. Эта инструкция предписывает процессору всегда следовать ветвлению. Чтобы отличать условное ветвление от безусловного, в MIPS для инструкции этого типа используется название *jump* (переход), сокращенное до j (метка Exit определена ниже).

```
j Exit          # перейти на Exit
```

Инструкция присваивания в else-блоке инструкции if опять может быть скомпилирована в одну инструкцию. К этой инструкции нужно только добавить метку Else. Мы также показываем метку Exit, следующую за этой инструкцией, которая свидетельствует об окончании откомпилированного кода структуры if-then-else:

```
Else:sub $s0,$s1,$s2        # f = g - h (пропускается, если i = j)
Exit:
```

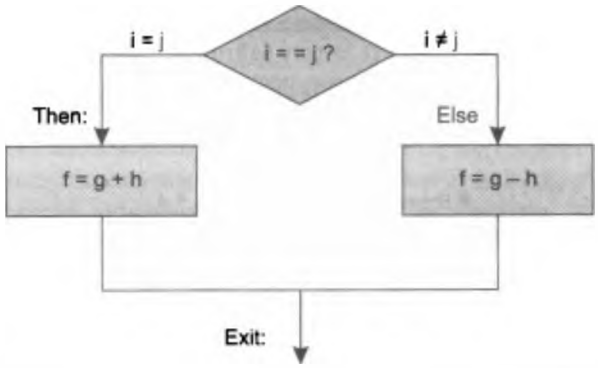


Рис. 2.4. Иллюстрация вариантов в показанной выше инструкции if. Левый прямоугольник соответствует части then инструкции if, а правый прямоугольник соответствует ее части else

Учтите, что ассемблер освобождает компилятор и программиста, работающего на ассемблере, от нудных вычислений адресов ветвлений, как это делается для вычисления адресов данных для загрузок и сохранений (см. раздел 2.12).

Интерфейс аппаратного и программного обеспечения

Компиляторы часто создают ветвления и метки там, где их нет в языках программирования. Избавление от скучного создания явных меток и ветвлений является одним из преимуществ написания программ на языках высокого уровня и причиной того, что программирование на этом уровне осуществляется быстрее.

Циклы

Решения играют важную роль как для выбора из двух альтернативных вариантов, который встречается в инструкциях `if`, так и для повторения вычисления, встречающегося в циклах. Строительными кирпичиками в обоих случаях служат одни и те же ассемблерные блоки.

Упражнение

Компиляция цикла while, реализованного на языке C

Так выглядит традиционный цикл, реализованный на языке C:

```
while (save[i] == k)
  i += 1;
```

Предположим, что `i` и `k` соотносятся с регистрами `$s3` и `$s5`, а базовый адрес массива `save` сохранен в регистре `$s6`. Каким будет ассемблерный MIPS-код, соответствующий этому фрагменту кода на языке C?

Ответ

Первым шагом будет загрузка `save[i]` во временный регистр. Перед тем как мы сможем загрузить `save[i]` во временный регистр, нам нужно получить его адрес. Перед тем как мы сможем прибавить `i` к базовому адресу массива `save` для формирования адреса, нужно будет умножить индекс `i` на 4, чтобы решить задачу байтовой адресации. К счастью, мы можем воспользоваться логическим сдвигом влево, поскольку сдвиг влево на два разряда равносилен умножению на 2^2 или на 4 (см. предыдущий раздел). Нам нужно добавить к этой инструкции метку `Loop`, чтобы в конце цикла можно было осуществить ветвление назад к этой инструкции:

```
Loop: sll  $t1,$s3,2      # Временный регистр $t1 = i * 4
```

Для получения адреса `save[i]` нужно прибавить значение `$t1` к базовому адресу `save`, хранящемуся в регистре `$s6`:

```
add $t1,$t1,$s6      # $t1 = адресу save[i]
```

Теперь можно использовать этот адрес для загрузки `save[i]` во временный регистр:

```
lw  $t0,0($t1)       # Временный регистр $t0 = save[i]
```

Следующая инструкция выполняет тест цикла, осуществляя выход, если `save[i] = k`:

```
bne  $t0,$s5, Exit    # перейти на Exit, если save[i] = k
```

Следующая инструкция прибавляет 1 к `i`:

```
addi  $s3,$s3,1       # i = i + 1
```

В конце цикла происходит ветвление назад к тесту цикла `while` в его верхней части. После этого ветвления мы просто добавим метку `Exit`, и все будет готово:

```
j    Loop # go to Loop
Exit:
```

(См. упражнения по оптимизации этой последовательности.)

Интерфейс аппаратного и программного обеспечения

Последовательности инструкций, которые заканчиваются ветвлением, настолько существенны для компиляции, что им дали свой собственный специальный термин: **базисный блок** – последовательность инструкций, не имеющая ветвлений и адресов перехода или меток ветвления. Одна из самых первых ранних фаз компиляции разбивает программу на базисные блоки.

Тест на равенство или на неравенство, наверное, самый популярный тест, но иногда полезно узнать, что переменная меньше другой переменной. Например, циклу for может понадобиться тест, чтобы проверить, что индексная переменная меньше нуля. Такое сравнение проводится в языке ассемблера MIPS с помощью инструкции, которая сравнивает два регистра и устанавливает для третьего регистра значение 1, если первый регистр меньше второго; в противном случае она устанавливает значение 0. Эта MIPS-инструкция называется *«установить, если меньше чем»*, – *set on less than*, или `slt`. Например,

```
slt     $t0, $s3, $s4          # $t0 = 1, если $s3 < $s4
```

означает, что для регистра `$t0` устанавливается значение 1, если значение в регистре `$s3` меньше значения в регистре `$s4`; в противном случае для регистра `$t0` устанавливается значение 0.

В сравнениях часто используются операнды-константы, поэтому существует непосредственная версия инструкции «установить, если меньше чем». Для проверки того, что значение регистра `$s2` меньше константы 10, можно просто написать следующий код:

```
slti    $t0,$s2,10             # $t0 = 1, если $s2 < 10
```

Интерфейс аппаратного и программного обеспечения

Компиляторы MIPS используют инструкции `slt`, `slti`, `beq`, `bne` и фиксированное значение 0 (всегда доступное путем чтения регистра `$zero`) для создания всех условий отношений: равенства, неравенства, меньше чем, меньше чем или равно, больше чем, больше чем или равно.

Учитывая предупреждение фон Неймана о простоте «оборудования», MIPS-архитектура не включает ветвление «меньше чем», потому что оно слишком сложное; либо оно будет удлинять период тактовых импульсов, либо его выполнение займет дополнительные тактовые циклы на инструкцию. Более полезны две более быстрые инструкции.

Интерфейс аппаратного и программного обеспечения

Инструкции сравнения должны иметь дело с противопоставлением чисел со знаком и без

Базисный блок
Последовательность инструкций, не имеющая ветвлений (за исключением, может быть, в самом конце) и без адресов перехода или меток ветвлений (за исключением, может быть, в самом начале).

знака. Иногда комбинация регистров с единицей в старшем разряде представляет отрицательное число, и, разумеется, она меньше любого положительного числа, которое должно иметь нуль в старшем разряде. С другой стороны, при работе с беззнаковыми целыми числами единица в старшем разряде представляет число, которое *больше*, чем любое другое число, которое начинается с нуля. (Скоро мы получим выгоду из такого двойственного значения старшего разряда для сокращения затрат на проверку границ массива.)

MIPS предлагает две версии сравнения «установить, если меньше чем» для того, чтобы справиться с этими альтернативами. *Set on less than* (`slt`) and *set on less than immediate* (`slti`) работают с целыми числами со знаком. Беззнаковые целые числа сравниваются с использованием *set on less than unsigned* (`sltu`) и *set on less than immediate unsigned* (`sltiu`).

Упражнение

Сравнение знаковых и беззнаковых чисел

Предположим, что регистр `$s0` содержит двоичное число

```
1111 1111 1111 1111 1111 1111 1111 11112
```

а регистр `$s1` содержит двоичное число

```
0000 0000 0000 0000 0000 0000 0000 00012
```

Какими будут значения регистров `$t0` и `$t1` после выполнения этих двух инструкций?

```
slt $t0, $s0, $s1       # знаковое сравнение
sltu $t1, $s0, $s1      # беззнаковое сравнение
```

Ответ

Значение в регистре `$s0` представляет собой число -1_{10}, если это целое число, и $4\,294\,967\,295_{10}$, если это беззнаковое целое число. Значение в регистре `$s1` представляет в любом случае число 1_2. Затем регистр `$t0` имеет значение 1, поскольку $-1_{10} < 1_{10}$, а регистр `$t1` имеет значение 0, поскольку $4\,294\,967\,295_{10} > 1_{10}$.

Интерпретация чисел со знаком, как будто они беззнаковые, дает нам малозатратный способ проверки, $0 \leqslant x < y$, что соответствует условию «индекс не выходит за пределы границ массива». Ключевым моментом здесь служит тот факт, что отрицательные целые числа с дополнением до двух в беззнаковой нотации выглядят как большие числа; то есть старший разряд в прежней нотации является знаковым разрядом, а в последней нотации он является наибольшей частью числа. Таким образом, беззнаковое сравнение $x < y$ также проверяет, не является ли x отрицательным числом, наряду с проверкой того, что x меньше y.

Упражнение

Прием проверки границ

Воспользуйтесь этим приемом для сокращения проверки выхода за границы массива: передайте управление на метку `IndexOutOfBounds`, если `$s1` ≥ `$t2` или если `$s1` является отрицательным числом.

Ответ

Для проведения обеих проверок код всего лишь использует инструкцию `sltu`:

```
sltu $t0,$s1,$t2                # $t0=0 если $s1>=длине, или $s1<0
beq  $t0,$zero,IndexOutOfBounds # если проверка не удалась, перейти к
                                # обработке ошибки
```

Инструкция Case или Switch

Большинство языков программирования имеют инструкцию case или инструкцию switch, позволяющую программисту выбрать одну из многих альтернатив в зависимости от отдельного значения. Простейший способ реализации инструкции `switch` заключается в использовании последовательности условных тестов, превращающей инструкцию `switch` в цепочку инструкций `if-then-else`.

Иногда альтернативы могут быть более эффективно закодированы в виде таблицы адресов альтернативных последовательностей инструкций, что называется адресной **таблицей переходов** или **таблицей переходов**, и программе нужно только перейти по индексу в таблицу, а затем к соответствующей последовательности. Таблица переходов представляет собой обыкновенный массив из слов, содержащих адреса, которые соответствуют меткам в коде. Программа загружает соответствующую запись из таблицы переходов в регистр. Затем ей нужно осуществить переход, используя адрес в регистре. Для поддержки таких ситуаций компьютеры вроде MIPS включают инструкцию перехода по регистру — *jump register* (`jr`), означающую безусловный переход на адрес, указанный в регистре. Затем, используя эту инструкцию, программа передает управление на нужный адрес, который описывается в следующем разделе.

Интерфейс аппаратного и программного обеспечения

Хотя для принятия решения и циклов в языках программирования, подобных C и Java, существует множество инструкций, основной инструкцией, которая осуществляет их реализацию на уровне набора инструкций, является условное ветвление.

Уточнение. Если вам не приходилось слышать о просроченных ветвлениях, рассматриваемые в главе 4, не стоит волноваться: ассемблер MIPS делает их невидимыми для работающего на нем программиста.

Самопроверка

1. В языке C имеется большое количество инструкций для принятия решений и циклов, в то время как в MIPS их всего несколько. Что из следующих утверждений объясняет или не объясняет причины этого дисбаланса? Почему?

Адресная таблица переходов
Также называется таблицей переходов. Таблица адресов альтернативных последовательностей инструкций.

1. Чем больше инструкций принятия решений, тем проще читать и воспринимать код.
2. Меньшее количество инструкций принятия решений упрощает задачу более низкого уровня, ответственного за их выполнение.
3. Большее количество инструкций принятия решений означает меньшее количество строк кода, что, в целом, сокращает время программирования.
4. Большее количество инструкций принятия решений означает меньшее количество строк кода, что, в целом, приводит к выполнению меньшего количества операций.

II. Почему в языке C имеются два оператора для И (`&` и `&&`) и два оператора для ИЛИ (`|` и `||`), а в MIPS такого разнообразия нет?

1. Логические операции И и ИЛИ реализуются с помощью операторов `&` и `|`, а условные переходы реализуются с помощью операторов `&&` и `||`.
2. В предыдущем утверждении все наоборот: `&&` и `||` соответствуют логическим операциям, а `&` и `|` относятся к условным переходам.
3. Это избыточные операторы, означающие одно и то же: операторы `&&` и `||` просто унаследованы от языка B, предшественника языка C.

2.8. Поддержка процедур в компьютерном оборудовании

Процедуры и функции относятся к инструментарию, используемому программистами для структурирования программ с целью сделать их понятнее и обеспечить повторное использование кода. Процедуры позволяют программисту сконцентрироваться на одной из частей задачи, а параметры служат интерфейсом между процедурами и остальной программой и данными, поскольку они могут передавать значения и возвращать результаты.

Процедуры можно представить в виде неких шпионов, живущих по секретным планам, получающим ресурсы, выполняющим задачу, заметающим следы, а затем возвращающим управление в исходную точку с желаемым результатом. Как только миссия будет выполнена, им больше уже не о чем беспокоиться. Более того, шпионы действуют исключительно по принципу минимума необходимых знаний, поэтому шпион даже не может предположить, кто является его нанимателем.

Точно так же при выполнении процедуры программа должна придерживаться следующих шести шагов:

1. Поместить параметры туда, где они будут доступны процедуре.
2. Передать управление процедуре.
3. Получить ресурсы хранения информации, необходимые процедуре.
4. Выполнить задачу по предназначению.
5. Поместить получившееся значение туда, где оно будет доступно вызывающей программе.

5. Вернуть управление в исходную точку, поскольку процедура может вызываться из нескольких мест программы.

Как уже упоминалось, наиболее быстродействующим местом хранения данных в компьютере являются регистры, поэтому их нужно использовать как можно интенсивнее. Программное обеспечение MIPS относительно распределения своих 32 регистров при вызове процедур придерживается следующих соглашений:

- `$a0–$a3`: это четыре регистра аргументов, предназначенные для передачи параметров;
- `$v0–$v1`: это два регистра значений, предназначенные для возвращаемых значений;
- `$ra`: это один регистр адреса возврата, предназначенный для передачи управления в исходную точку.

Кроме распределения этих регистров, язык ассемблера MIPS включает инструкцию, предназначенную исключительно для процедур: она осуществляет переход на адрес и одновременно сохраняет адрес следующей инструкции в регистре `$ra`. Это инструкция перехода и ссылки — **jump-and-link instruction** (`jal`), имеющая довольно простой формат:

```
jal АдресПроцедуры
```

Часть имени, называющаяся *ссылкой*, означает, что формируется адрес, или ссылка, указывающая на место вызова, которая позволяет процедуре вернуться на нужный адрес. Эта ссылка, хранящаяся в регистре `$ra` (регистр 31), называется **адресом возврата**. Необходимость в адресе возврата обусловливается тем, что одна и та же процедура может быть вызвана из нескольких фрагментов программы.

Для этого в таких компьютерах, как MIPS, используется инструкция перехода по регистру — *jump register* (`jr`), представленная ранее в качестве вспомогательного средства для работы инструкций выбора (`case`) и означающая безусловный переход на адрес, указанный в регистре:

```
jr $ra
```

Инструкция перехода по регистру осуществляет переход на адрес, сохраненный в регистре `$ra`, — что, собственно, от нее и требуется. Таким образом, **вызывающая программа** помещает значения параметров в регистры `$a0–$a3` и использует инструкцию `jal X` для перехода к процедуре `X` (которая иногда называется **вызываемой программой**). Затем вызываемая программа выполняет вычисления, помещает результаты в регистры `$v0` и `$v1` и возвращает управление вызывающей программе, используя инструкцию `jr $ra`.

Инструкция перехода и ссылки (jump-and-link)
Инструкция, осуществляющая переход на адрес и одновременно сохраняющая адрес следующей инструкции в регистре ($ra в MIPS).

Адрес возврата
Ссылка на место вызова, позволяющая процедуре вернуться на нужный адрес; в MIPS эта ссылка хранится в регистре $ra.

Вызывающая программа (CALLER)
Программа, вызывающая процедуру и предоставляющая необходимые значения параметров.

Вызываемая программа (CALLEE)
Процедура, выполняющая последовательность сохраненных инструкций на основе параметров, предоставляемых вызывающей программой, а затем возвращающая управление вызывающей программе.

Подразумевается, что идея сохраняемой в памяти программы предполагает наличие регистра для хранения адреса текущей выполняемой инструкции. В силу исторических причин в MIPS этот регистр почти всегда называют **счетчиком команд**, или сокращенно *PC* (*program counter*), хотя более подходящим названием для него было бы *регистр адреса инструкции*. При установке адреса возврата из процедуры инструкция `jal` сохраняет в регистре `$ra` для ссылки на следующую после вызова инструкцию значение PC + 4.

Использование большего количества регистров

Предположим, что компилятору требуется больше регистров для процедуры, чем четыре регистра аргументов и два регистра возвращаемых значений. Поскольку нам нужно замести свои следы после завершения миссии, все регистры, необходимые вызывающей программе, должны быть восстановлены, получив те значения, которые в них содержались *до* вызова процедуры. Эта ситуация является тем самым примером, упомянутым в разделе «Интерфейс аппаратного и программного обеспечения», когда нам необходимо сбросить регистры в память.

Идеальной структурой данных для сброса регистров является **стек** — очередь, работающая по принципу «последним пришел — первым вышел». Стеку нужен указатель на самый последний распределенный в нем адрес, чтобы показать, куда следующая процедура должна поместить сбрасываемые регистры или где найти старые значения регистров. **Указатель стека** изменяет свое значение на одно слово для каждого сохраненного или восстановленного регистра. Программное обеспечение MIPS для указателя стека выделяет регистр 29, давая ему конкретное имя `$sp`. Стеки настолько популярны, что для них существуют специальные термины для переноса данных в стек и из стека: помещение данных в стек называется **проталкиванием** (push), а удаление данных из стека называется **выталкиванием** (pop).

Счетчик команд (PC)
Регистр, содержащий адрес исполняемой в программе инструкции.

Стек
Структура данных для сброса регистров, имеющая организацию в виде очереди, работающей по принципу «последним пришел — первым вышел».

Указатель стека
Значение, указывающее на самый последний распределенный в стеке адрес, показывающее, куда нужно сбрасывать регистры или где могут быть найдены старые значения регистров. В MIPS указателем стека служит регистр `$sp`.

Проталкивание (push)
Добавление элемента к стеку.

Выталкивание (pop)
Удаление элемента из стека.

Исторически сложилось так, что стек «растет» от верхних адресов к нижним. Это соглашение означает, что вы проталкиваете значения в стек, отнимая значение от указателя стека. Добавление значения к указателю стека сокращает стек, выталкивая тем самым значения из стека.

Упражнение

Компиляция процедуры на языке C, не вызывающей другие процедуры

Давайте превратим пример из раздела 2.2 в процедуру на языке C:

```
int leaf_example (int g, int h, int i, int j)
{
 int f;
 f = (g + h) - (i + j);
 return f;
}
```

Каким будет скомпилированный ассемблерный код MIPS?

Ответ

Переменные-параметры g, h, i и j соответствуют регистрам аргументов $a0, $a1, $a2 и $a3, а переменная f соответствует регистру $s0. Скомпилированная программа начинается с метки процедуры:

```
leaf_example:
```

Следующим шагом будет сохранение регистров, используемых процедурой. Инструкция, реализующая присваивание в теле процедуры, идентична показанной в упражнении из раздела 2.2 «Компиляция сложной инструкции на языке C в MIPS-код», где используются два временных регистра. Таким образом, нужно сохранить три регистра: $s0, $t0 и $t1. Мы «проталкиваем» старые значения в стек, создавая в стеке пространство для трех слов (12 байт), а затем сохраняя регистры:

```
addi $sp, $sp, -12 # настройка стека с освобождением места для 3 записей
sw  $t1, 8($sp)    # сохранение регистра $t1 для последующего использования
sw  $t0, 4($sp)    # сохранение регистра $t0 для последующего использования
sw  $s0, 0($sp)    # сохранение регистра $s0 для последующего использования
```

На рис. 2.5 показан стек до, во время и после вызова процедуры. Следующие три инструкции соответствуют телу процедуры и практически повторяют код упражнения из раздела 2.2 «Компиляция сложной инструкции на языке C в MIPS-код»:

```
add $t0,$a0,$a1    # регистр $t0 содержит g + h
add $t1,$a2,$a3    # регистр $t1 содержит i + j
sub $s0,$t0,$t1    # f = $t0 - $t1, что соответствует (g + h)-(i + j)
```

Для возвращения значения f мы копируем его в регистр возвращаемого значения:

```
add $v0,$s0,$zero    # возвращение f ($v0 = $s0 + 0)
```

Перед возвращением из процедуры мы восстанавливаем три прежних сохраненных значения регистров путем «выталкивания» их из стека:

```
lw  $s0, 0($sp)  # восстановление регистра $s0 для вызывавшей программы
lw  $t0, 4($sp)  # восстановление регистра $t0 для вызывавшей программы
lw  $t1, 8($sp)  # восстановление регистра $t1 для вызывавшей программы
addi $sp,$sp,12  # корректировка указателя стека для удаления трех записей
```

Процедура заканчивается инструкцией перехода по регистру с использованием адреса возврата.

```
jr  $ra          # переход назад к вызывавшей программе
```

В предыдущем примере мы использовали временные регистры, при этом предполагая, что их прежние значения должны быть сохранены и восстановлены. Чтобы избежать сохранения и восстановления регистра, чье значение никогда не используется, что может иметь место в случае с временным регистром, программное обеспечение MIPS делит 18 из имеющихся регистров на две группы:

- **$t0–$t9:** десять временных регистров, которые *не* сохраняются вызываемой программой (процедурой) при вызове процедуры
- **$s0–$s7:** восемь сохраняемых регистров, которые должны быть сохранены при вызове процедуры (если они используются, вызываемая процедура их сначала сохраняет, а затем восстанавливает)

Это простое соглашение сокращает сброс регистров. Поскольку в показанном выше примере вызывающая программа не предполагает, что регистры $t0 и $t1 будут сохранены в процессе вызова процедуры, из кода можно выбросить два сохранения и две загрузки. Но мы по-прежнему должны сохранить и восстановить

регистр $s0, поскольку вызываемая программа должна предположить, что вызывающая программа нуждается в его значении.

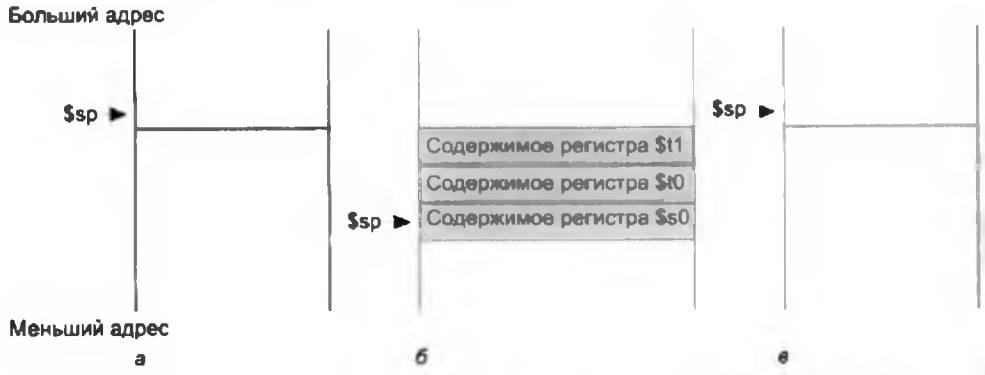


Рис. 2.5. Значения указателя стека и самого стека: а) до вызова, б) в процессе вызова, и в) после вызова процедуры. Указатель стека всегда указывает на «вершину» стека, или, как показано на этом рисунке, на последнее слово в стеке

Вложенные процедуры

Процедуры, не вызывающие других процедур, называются *конечными* (*leaf*). Если бы все процедуры были конечными, то жизнь стала бы проще, но, к сожалению, это не так. Точно так же как шпион для выполнения части своей миссии может нанять других шпионов, которые в свою очередь могут воспользоваться услугами еще большего количества шпионов, процедуры вызывают другие процедуры. Более того, рекурсивные процедуры даже вызывают «клонов», то есть самих себя. Поскольку мы должны быть внимательными при использовании регистров в процедурах, еще больше внимания нужно уделять вызовам неконечных процедур.

Предположим, к примеру, что основная программа вызывает процедуру `A` с аргументом 3, помещая значение 3 в регистр `$a0`, а затем используя инструкцию `jal A`. Затем предположим, что процедура `A` вызывает процедуру `B`, используя инструкцию `jal B` с аргументом 7, также помещаемым в регистр `$a0`. Поскольку процедура `A` еще не завершила свою задачу, использование регистра `$a0` вызывает конфликтную ситуацию. Точно так же возникает конфликт и с адресом возврата в регистре `$ra`, поскольку теперь в нем находится адрес возврата для процедуры `B`. Если мы что-нибудь не предпримем для предупреждения этой проблемы, возникший конфликт не позволит процедуре `A` вернуть управление вызывавшей ее программе.

Одним из решений будет проталкивание всех остальных регистров, которые должны быть сохранены, в стек, точно так же, как мы это делали с сохраняемыми регистрами. Вызывающая программа проталкивает в стек все регистры аргументов (`$a0–$a3`) или временные регистры (`$t0–$t9`), которые понадобятся ей после вызова. Вызываемая процедура проталкивает в стек регистр адреса возврата `$ra` и все сохраняемые регистры (`$s0–$s7`), используемые ею. Указатель стека настраивается в соответствии с числом регистров, помещаемых в стек. По возвращении регистры восстанавливаются из памяти и указатель стека соответствующим образом корректируется.

Упражнение

Компиляция рекурсивной процедуры на языке C с показом ссылок на вложения

Давайте попытаемся разобраться с рекурсивной процедурой, вычисляющей факториал:

```
int fact (int n)
{
  if (n < 1) return (1);
  else return (n * fact(n - 1));
}
```

Каким будет для нее ассемблерный код MIPS?

Ответ

Переменная параметра n соответствует регистру аргумента $a0. Скомпилированная программа начинается с метки процедуры, а затем сохраняет в стеке два регистра: адрес возврата и $a0:

```
fact:
 addi $sp, $sp, -8  # коррекция стека под 2 записи
 sw $ra, 4($sp)     # сохранение адреса возврата
 sw $a0, 0($sp)     # сохранение аргумента n
```

При первом вызове процедуры fact инструкция sw сохраняет адрес программы, вызвавшей fact. Следующие две инструкции проверяют, не является ли n меньше 1, осуществляя переход к метке L1, если $n \geq 1$.

```
 slti $t0,$a0,1     # проверка истинности выражения n < 1
 beq $t0,$zero,L1   # если n >= 1, переход к L1
```

Если n меньше 1, процедура fact возвращает единицу, помещая ее в регистр значения: она складывает единицу с нулем и помещает сумму в регистр $v0. Затем она выталкивает два сохраненных значения из стека и осуществляет переход на адрес возврата:

```
 addi $v0,$zero,1   # возвращение единицы
 addi $sp,$sp,8     # выталкивание двух записей из стека
 jr $ra             # возвращение к вызывавшей программе
```

Прежде чем выталкивать две записи из стека, туда нужно поместить регистры $a0 и $ra. Поскольку $a0 и $ra не изменились, потому что n меньше единицы, эти инструкции мы пропускаем.

Если n не меньше единицы, аргумент n уменьшается на единицу, а затем процедура fact вызывается еще раз, но уже с уменьшенным значением:

```
L1: addi $a0,$a0,-1   # n >= 1: аргумент получает значение (n - 1)
 jal  fact            # вызов fact с (n - 1)
```

Следующей идет инструкция, на которую возвращается fact. Теперь восстанавливается прежний адрес возврата, прежний аргумент, а также указатель стека:

```
 lw $a0, 0($sp)     # возвращение из вызова по jal: восстановление аргумента n
 lw $ra, 4($sp)     # восстановление адреса возврата
 addi $sp, $sp, 8   # настройка указателя стека на извлечение двух записей
```

Затем значением регистра $v0 становится результат умножения прежнего аргумента $a0 и текущего числа в регистре значения. Здесь предполагается, что нам доступна инструкция умножения, хотя до главы 3 она рассматриваться не будет:

```
 mul $v0,$a0,$v0  # возвращает n * fact (n - 1)
```

И наконец, процедура fact опять переходит на адрес возврата:

```
 jr $ra             # возвращение управления вызывавшей программе
```

Интерфейс аппаратного и программного обеспечения

Переменная в языке C в общем смысле является фрагментом памяти, и ее интерпретация зависит как от ее типа, так и от класса памяти. В упражнения включены целые числа и символы (см. раздел 2.9). В языке C имеется два класса памяти: автоматический (*automatic*) и статический (*static*). Автоматические переменные являются локальными по отношению к процедуре и аннулируются после выхода из процедуры. Статические переменные существуют независимо от входов в процедуры и выходов из них. Переменные в языке C, объявленные за пределами всех процедур, считаются статическими, так же как и все переменные, объявленные с использованием ключевого слова `static`. Все остальные переменные считаются автоматическими. Чтобы упростить доступ к статическим данным, программное обеспечение MIPS выделило другой регистр, названный **глобальным указателем**, или `$gp`.

В табл. 2.6 представлены сведения о том, что сохраняется в процессе вызова процедуры. Учтите, что ряд схем сохраняет сам стек, гарантируя, что вызывающая программа получит те же самые данные обратно путем загрузки их из стека, поскольку они в нем сохранены. Стек выше `$sp` сохраняется просто тем, что принимаются меры, не позволяющие вызываемой программе вести запись выше адреса, хранящегося в `$sp`; а значение самого `$sp` сохраняется вызываемой программой путем добавления к нему точно такого же значения, которое было из него вычтено; а все остальные регистры сохраняются за счет сохранения их в стеке (если они используются) и восстановления их оттуда.

Таблица 2.6. Что сохраняется, а что нет в процессе вызова процедуры. Если программное обеспечение зависит от регистра указателя фрейма или регистра глобального указателя, которые рассматриваются в следующих подразделах, то они также сохраняются

Сохраняются	Не сохраняются
Сохраняемые регистры: $s0–$s7	Временные регистры: $t0–$t9
Регистр указателя стека: $sp	Регистры аргументов: $a0–$a3
Регистр адреса возврата: $ra	Регистры возвращаемых значений: $v0–$v1
Стек выше указателя стека	Стек ниже указателя стека

Распределение пространства для новых данных в стеке

Последняя сложность состоит в том, что стек также используется для хранения переменных, которые носят для процедуры локальный характер, но не помещаются в регистры, например массивы или структуры. Сегмент стека, содержащий сохраненные регистры процедуры и ее локальные переменные, называется фреймом процедуры, или записью активации. На рис. 2.6 показано состояние стека до, в процессе и после вызова процедуры.

Глобальный указатель
Регистр, выделенный для указания на статическую область.

Часть программного обеспечения MIPS использует **указатель фрейма** ($fp) для указания на первое слово фрейма процедуры. В процессе работы процедуры указатель стека может изменяться, и поэтому ссылки на локальную переменную в памяти могут иметь разные смещения, в зависимости от того, где они находятся в процедуре, затрудняя понимание процедуры. В качестве альтернативного варианта указатель фрейма предлагает внутри процедуры стабильный базовый регистр для локальных ссылок в памяти. Заметьте, что запись активации появляется в стеке независимо от того, используется или нет явный указатель фрейма. Мы избегали использования $fp, чтобы избежать изменений $sp внутри процедуры: в наших упражнениях стек корректировался только при входе в процедуру и выходе из нее.

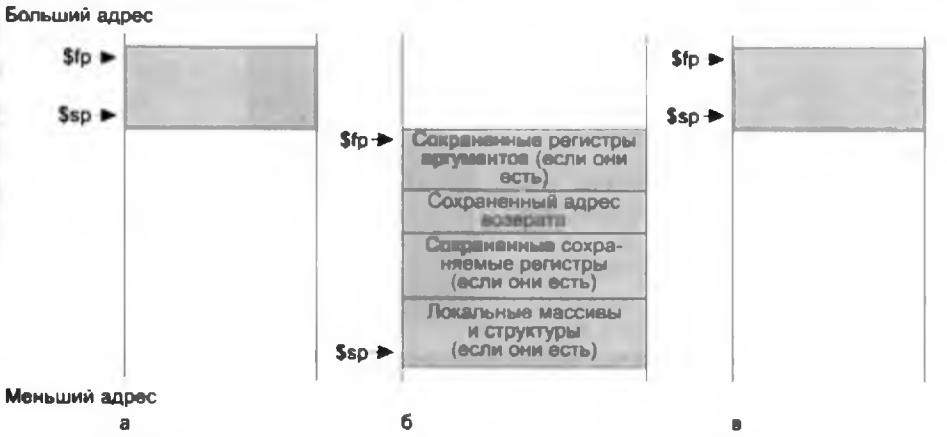


Рис. 2.6. Иллюстрация распределения пространства стека: а) до вызова, б) в процессе вызова и в) после вызова процедуры. Указатель фрейма ($fp) указывает на первое слово фрейма, зачастую это сохраненный регистр аргумента, а указатель стека ($sp) указывает на вершину стека. Стек скорректирован таким образом, чтобы создать пространство для всех сохраненных регистров и всех находящихся в памяти локальных переменных. Поскольку указатель стека в процессе выполнения программы может изменяться, программистам проще ссылаться на переменные посредством неизменного указателя фрейма, хотя это может быть сделано и с помощью указателя стека и небольших арифметических вычислений адреса. Если локальных переменных в стеке процедуры нет, компилятор сэкономит время, не устанавливая и не восстанавливая указатель фрейма. Когда указатель фрейма используется, он инициализируется при вызове адресом, хранящимся в $sp, а $sp восстанавливается с использованием $fp

Распределение пространства для данных в «куче»

Вдобавок к автоматическим переменным, локальным по отношению к процедурам, программистам, работающим на C, требуется пространство для статических переменных и для динамических структур данных. На рис. 2.7 показано соглашение, используемое в MIPS для распределения памяти. Стек начинается в верхних адресах памяти и растет вниз. Первая часть нижних адресов памяти зарезервирована, далее идет место для машинного кода MIPS,

Фрейм процедуры (PROCEDURE FRAME)
Также называется записью активации (activation record). Сегмент стека, содержащий сохраняемые процедурой регистры и локальные переменные.

Указатель фрейма
Значение, служащее индикатором нахождения сохраненных регистров и локальных переменных для заданной процедуры.

традиционно называемое **текстовым сегментом**. Выше этого кода находится *сегмент статических данных*, являющийся местом для констант и статических переменных. Хотя массивы склонны иметь фиксированную длину и поэтому хорошо вписываются в сегмент статических данных, для таких структур данных, как связанные списки, в процессе их использования характерен рост и сокращение. Сегмент для таких структур данных традиционно называется «кучей» и помещается в памяти еще выше. Заметьте, что такое распределение позволяет стеку и *куче* расти навстречу друг другу, позволяя, таким образом, эффективно использовать память по мере увеличения и уменьшения этих двух сегментов.

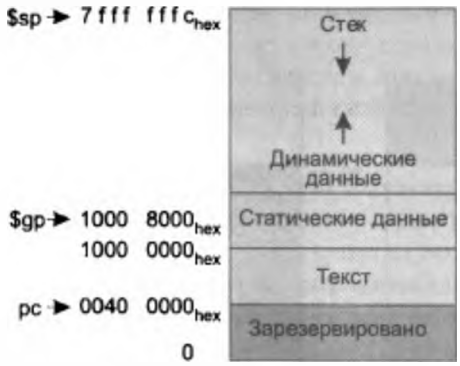


Рис. 2.7. Распределение памяти в MIPS для программ и данных. Эта адресация представляет собой всего лишь соглашение, касающееся программного обеспечения, и не является частью MIPS-архитектуры. Указатель стека получает начальное значение 7fff fffc$_{16}$ и растет вниз по направлению к сегменту данных. С другого конца программный код («текст») начинается с адреса 0040 0000$_{16}$. Статические данные начинаются с адреса 1000 0000$_{16}$. Далее следуют динамические данные, распределяемые в C функцией malloc(), а в Java функцией new. Они растут вверх по направлению к стеку в области, называемой кучей. Глобальный указатель, $gp, устанавливается на адрес, облегчающий доступ к данным. Он получает начальное значение 1000 8000$_{16}$, поэтому с помощью положительного и отрицательного 16-разрядного смещения из $gp можно получить доступ к адресам в диапазоне от 1000 0000$_{16}$ to 1000 ffff$_{16}$

Язык C выделяет и освобождает пространство в куче, применяя специальные функции. Функция `malloc()` выделяет пространство в куче и возвращает указатель на него, а функция `free()` освобождает пространство в куче согласно значению указателя. В C распределение памяти контролируется программами и является источником многих часто допускаемых и трудно обнаруживаемых ошибок. Забывчивость при освобождении пространства приводит к «утечке памяти», что со временем отнимает так много памяти, что операционная система может зависнуть. Слишком раннее освобождение памяти создает «зависшие указатели», что может привести к тому, что указатели станут указывать на то, что никогда не принадлежало программе. В Java с целью устранения подобных ошибок используется автоматическое распределение памяти и так называемая сборка мусора.

Текстовый сегмент (text segment) Сегмент объектного файла UNIX, в котором содержится код на машинном языке для подпрограмм в исходном файле.

Краткая сводка соглашения по регистрам, которое используется в языке ассемблера MIPS, приведена в табл. 2.7.

Таблица 2.7. Соглашение по регистрам в MIPS. Регистр 1, называемый $at, зарезервирован для ассемблера (см. раздел 2.12), а регистры 26–27, называемые $k0–$k1, зарезервированы для операционной системы

Имя	№ регистра	Использование	Осуществляется ли сохранение при вызове?
$zero	0	Значение константы 0	не определено
$v0-$v1	2-3	Значения для результатов и вычисления выражений	нет
$a0-$a3	4-7	Аргументы	нет
$t0-$t7	8-15	Временные данные	нет
$s0-$s7	16-23	Сохраняемые данные	да
$t8-$t9	24-25	Дополнительные временные данные	нет
$gp	28	Глобальный указатель	да
$sp	29	Указатель стека	да
$fp	30	Указатель фрейма	да
$ra	31	Адрес возврата	да

Уточнение. А что, если будет более четырех параметров? Соглашение MIPS предусматривает помещение дополнительных параметров в стек сразу же над указателем фрейма. Затем процедура ожидает наличия первых четырех параметров в регистрах от $a0 и до $a3, а остальные ищет в памяти, адресуемой с помощью указателя фрейма.

Как упоминалось в подписи к рис. 2.6, указатель фрейма удобен тем, что все ссылки на переменные в стеке из процедуры будут иметь одинаковые смещения. Но указатель фрейма использовать не обязательно. C-компилятор GNU MIPS использует указатель фрейма, а C-компилятор от MIPS его не использует — он рассматривает регистр 30 как еще один сохраняемый регистр ($s8).

Уточнение. Некоторые рекурсивные процедуры могут быть реализованы за счет повторений без использования рекурсий. Повторение может существенно повысить производительность за счет избавления от издержек, связанных с вызовами процедуры. Рассмотрим, к примеру, процедуру, используемую для аккумулирования суммы:

```
int sum (int n, int acc) {
 if (n > 0)
   return sum(n - 1, acc + n);
 else
   return acc;
```

Рассмотрим вызов процедуры sum(3,0). Он приведет к рекурсивным вызовам sum(2,3), sum(1,5) и sum(0,6), а затем будет четыре раза возвращен результат 6. Этот рекурсивный вызов процедуры sum называется *вызовом хвоста*, и этот пример, использующий хвостовую рекурсию, может быть реализован очень эффективно (предположим, что $a0 = n и $a1 = acc):

```
sum: slti $a0,1              # проверка  n <= 0
 beq $a0, $zero, sum_exit    # переход на sum_exit, если n <= 0
 add $a1, $a1, $a0           # прибавление n к acc
addi $a0, $a0, -1            # вычитание 1 из n
 j sum                       # переход на sum
sum_exit:
 add $v0, $a1, $zero         # возвращение значения acc
 jr $ra                      # возвращение управления вызывающей программе
```

Самопроверка

Какие утверждения, касающиеся C и Java, в целом соответствуют действительности?

1. Программисты, работающие на C, управляют данными явным образом, а в Java это делается автоматически.
2. При работе на C допускается намного больше ошибок, связанных с указателями и утечками памяти, чем при работе на Java.

2.9. Общение с людьми

!(@|=> (wow open tab at bar is great: ого, открытый счет в баре — это круто.)

Четвертая строка клавиатурной поэмы «Hatless Atlas»

Компьютеры были изобретены в качестве «числодробилок», но как только они приобрели коммерческую жизнеспособность, они стали применяться для обработки текста. Сегодня большинство компьютеров предлагают для представления символов восьмибитные байты, а американский стандартный код для обмена информацией — American Standard Code for Information Interchange (ASCII) — практически является отображением всего, что показано ниже. Краткая сводка по коду ASCII представлена на рис. 2.8.

Интерфейс аппаратного и программного обеспечения

Числа по основанию 2 непривычны для людей, у нас десять пальцев, и поэтому для нас более естественными являются числа по основанию 10. Почему тогда компьютеры не используют десятичные числа? В действительности первый коммерческий компьютер предлагал десятичную арифметику. Проблема была в том, что компьютер по-прежнему использовал сигналы включения и выключения, поэтому десятичные цифры просто были представлены несколькими двоичными цифрами. Применение десятичных чисел оказалось настолько неэффективным, что последующие компьютеры вернулись к использованию только двоичных чисел, преобразуя их в числа по основанию 10 только для относительно нечастых событий ввода-вывода.

ASCII значение	Символ	ASCII значение	Символ	ASCII значение	Символ	ASCII значение	Символ	ASCII значение	Символ	ASCII значение	Символ
32	space	48	0	64	@	80	P	096	`	112	p
33	!	49	1	65	A	81	Q	097	a	113	q
34	"	50	2	66	B	82	R	098	b	114	r
35	#	51	3	67	C	83	S	099	c	115	s
36	$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	|
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	DEL

Рис. 2.8. Представление символов в таблице ASCII. Обратите внимание на то, что коды букв в верхнем и нижнем регистрах различаются в точности на 32 единицы; это наблюдение может привести к удобным способам проверки или изменения регистра букв. Значения, не показанные в данной таблице, включают символы форматирования. Например, код 8 представляет символ забоя, код 9 — символ табуляции, а код 13 — возврат каретки. Еще одним полезным значением будет 0 для null. Это значение в языке программирования C используется для маркировки конца строки

Упражнение

Коды ASCII и двоичные числа

Вместо целочисленных значений числа можно представить в виде строк ASCII. Насколько увеличится объем памяти, если один миллиард будет представлено не 32-разрядным целым числом, а кодами ASCII?

Ответ

Один миллиард – это 1 000 000 000, поэтому его отображение займет 10 ASCII-цифр, длина каждой из которых составит 8 бит. Поэтому увеличение требуемого объема памяти составит (10 × 8)/32, или 2,5. Кроме увеличения требуемого объема памяти, устройство для сложения, вычитания, умножения и деления таких десятичных чисел будет слишком сложным. Подобные сложности объясняют, почему компьютерные профессионалы воспитаны на вере,

что двоичные числа является вполне естественными, а отдельные экземпляры десятичных компьютеров совершенно нелепы.

Группа инструкций может извлечь байт из слова, поэтому загрузки слова и сохранения слова вполне достаточно для передачи как байтов, так и слов. Тем не менее ввиду популярности текста в некоторых программах MIPS предоставляет инструкции для перемещения байтов. Инструкция **load byte** (`lb`) загружает байт из памяти, помещая его в правые восемь разрядов регистра. Инструкция **store byte** (`sb`) берет байт из правых восьми разрядов регистра и записывает его в память. Таким образом, байт копируется с помощью следующей последовательности инструкций:

```
lb $t0,0($sp)  # Чтение байта из источника
sb $t0,0($gp)  # Запись байта в приемник
```

Интерфейс аппаратного и программного обеспечения

Числа со знаком и без знака играют такую же роль в загрузке, как и в арифметике. Функция загрузки со знаком, которая должна повторно копировать знак для заполнения всего остального пространства регистра, называется расширением знака, но ее цель — поместить правильное представление числа в этот регистр. Загрузка без знака просто заполняет нулями все пространство слева от данных, поскольку число, представленное комбинацией разрядов, является числом без знака.

При загрузке 32-разрядного слова в 32-разрядный регистр этот вопрос носит спорный характер; знаковая и беззнаковая загрузка идентичны друг другу. MIPS предлагает две разновидности загрузки байтов: инструкция **load byte** (`lb`) рассматривает байт как число со знаком и поэтому проводит расширение знака для заполнения 24 самых левых разрядов регистра, а инструкция **load byte unsigned** (`lbu`) работает с целыми числами без знака. Поскольку C-программы почти всегда используют байты для представления символов, а не для представления очень коротких целых чисел со знаком, для загрузки байтов практически всегда используется инструкция `lbu`.

Обычно символы объединяются в строки, которые имеют переменное количество символов. Для представления строки существуют три варианта: 1) первая позиция строки резервируется для предоставления длины строки, 2) длина строки содержится в препроводительной переменной (как в структуре) или 3) последняя позиция строки обозначается символом, используемым в качестве метки конца строки. В языке C используется третий вариант, при котором строка заканчивается байтом, имеющим нулевое значение (которое в ASCII называется **null**). Таким образом, строка «Cal» представляется в C следующими четырьмя байтами, показанными в виде десятичных чисел: 67, 97, 108, 0. (Как будет показано далее, в Java используется первый вариант.)

Упражнение

Компилирование процедуры копирования строки, показывающее, как используются строки в программах на языке C

Процедура strcpy копирует строку y в строку x, используя соглашение, принятое в языке C, касающееся завершения строки нулевым байтом:

```
void strcpy (char x[], char y[])
{
  int i;
  i = 0;
  while ((x[i] = y[i]) != '\0') /* копирование и тестирование байта */
    i += 1;
}
```

Каким будет ассемблерный код MIPS?

Ответ

Ниже показан основной сегмент ассемблерного кода MIPS. Предположим, что базовые адреса для массивов x и y находятся в $a0 и в $a1, а переменная i находится в $s0. Инструкция strcpy корректирует указатель стека, а затем помещает сохраняемый регистр $s0 в стек:

```
strcpy:
  addi $sp,$sp,-4          # корректировка стека для еще одной записи
  sw $s0, 0($sp)           # сохранение $s0
```

Чтобы присвоить i начальное значение 0, следующая инструкция устанавливает значение $s0 в 0, складывая 0 и 0 и помещая эту сумму в $s0:

```
  add $s0,$zero,$zero      # i = 0 + 0
```

Далее начинается цикл. Сначала формируется адрес y[i] путем добавления i к y[]:

```
L1: add $t1,$s0,$a1        # адрес y[i] помещается в $t1
```

Обратите внимание на то, что нам не нужно умножать i на 4, поскольку y является массивом байтов, а не слов, как в предыдущих упражнениях.

Для загрузки символа в y[i] мы используем инструкцию загрузки байта без знака — load byte unsigned, которая помещает символ в $t2:

```
  lbu $t2, 0($t1)          # $t2 = y[i]
```

Такое же вычисление адреса помещает адрес x[i] в $t3, а затем символ в $t2 сохраняется по этому адресу.

```
  add $t3,$s0,$a0          # адрес x[i] помещается в $t3
  sb $t2, 0($t3)           # x[i] = y[i]
```

Затем мы выходим из цикла, если символ был нулевым. То есть мы выходим, если это последний символ строки:

```
  beq $t2,$zero,L2         # если y[i] == 0, переход на L2
```

Если нет, мы увеличиваем значение i на единицу и переходим в начало цикла:

```
  addi $s0, $s0,1          # i = i + 1
  j L1                     # переход на L1
```

Если мы не переходим в начало цикла, значит, это был последний символ в строке; мы восстанавливаем $s0 и указатель стека, а затем возвращаемся из процедуры.

```
L2: lw $s0, 0($sp)         # y[i] == 0: конец строки. Восстановление
                           # прежнего значения $s0
    addi $sp,$sp,4         # выталкивание одного слова из стека
    jr $ra                 # возвращение
```

Для исключения операций с переменной i, показанных выше, строки в C обычно копируются с использованием указателей, а не массивов. Объяснение разницы между массивами и указателями дается в разделе 2.14.

Поскольку показанная выше процедура strcpy является конечной, компилятор может разместить i во временном регистре и избежать сохранения и восстановления $s0. Поэтому, вместо того чтобы рассматривать регистры $t только в качестве временных, их нужно считать регистрами, которые вызываемая процедура может использовать там, где это удобно. Когда компилятор имеет дело с конечной процедурой, он расходует все временные регистры, прежде чем использовать те регистры, которые он должен сохранить.

Символы и строки в языке Java

Unicode является универсальной кодировкой алфавитов большинства естественных языков. На рис. 2.9 показан список алфавитов кодировки Unicode; в этой кодировке почти столько же *алфавитов*, сколько полезных *символов* в ASCII. Чтобы охватить как можно больше обычных языков, Java использует для символов кодировку Unicode. По умолчанию для представления символа в нем используется 16 бит.

Набор инструкций MIPS имеет специальные инструкции для загрузки и сохранения таких 16-разрядных величин, называемых *полусловами*. Инструкция **load** half (lh) загружает полуслово из памяти, помещая его в правые 16 разрядов регистра. Как и load byte, *load half* (lh) рассматривает полуслово как число со знаком и поэтому осуществляет расширение знака, заполняя 16 левых разрядов регистра, а вот инструкция **load halfword unsigned** (lhu) работает с целыми числами без знака. Поэтому из этих двух инструкций lhu используется чаще. Инструкция **store** half (sh) берет полуслово из правых 16 разрядов регистра и записывает его в память. Полуслово копируется с помощью следующей последовательности:

```
lhu $t0,0($sp)      # Чтение полуслова (16 бит) из источника
sh $t0,0($gp)       # Запись полуслова (16 бит) в приемник
```

Строки являются стандартным Java-классом со специальной встроенной поддержкой и предопределенными методами для объединения, сравнения и преобразования. В отличие от C язык Java включает слово, которое дает длину строки, подобно тому, как это делается для массивов Java.

Уточнение. Программное обеспечение MIPS старается выравнивать стек по адресам слов, позволяя программам всегда использовать для доступа к стеку инструкции lw и sw (которые должны быть выровнены по границе слова). Это соглашение означает, что символьная переменная, размещенная в стеке, занимает 4 байта, даже если ей нужно меньше места. Тем не менее строковая переменная языка C или массив байтов будут паковаться по четыре байта на слово, а строковая переменная языка Java или массив коротких чисел будут паковаться по два полуслова на слово.

Latin	Malayalam	Tagbanwa	Общая пунктуация
Greek	Sinhala	Khmer	Символы, модифицирующие интервалы
Cyrillic	Thai	Mongolian	Валютные символы
Armenian	Lao	Limbu	Подборка диакритических знаков
Hebrew	Tibetan	Tai Le	Подборка меток для символов
Arabic	Myanmar	Kangxi Radicals	Верхние и нижние символы
Syriac	Georgian	Hiragana	Формы чисел
Thaana	Hangul Jamo	Katakana	Математические операторы
Devanagari	Ethiopic	Bopomofo	Математические алфавитно-цифровые символы
Bengali	Cherokee	Kanbun	Шаблоны Брайля
Gurmukhi	Unified Canadian Aboriginal Syllabic	Shavian	Символы, распознаваемые считывателями
Gujarati	Ogham	Osmanya	Византийские музыкальные символы
Oriya	Runic	Cypriot Syllabary	Музыкальные символы
Tamil	Tagalog	Tai Xuan Jing Symbols	Стрелки
Telugu	Hanunoo	Yijing Hexagram Symbols	Прямоугольные фигуры
Kannada	Buhid	Aegean Numbers	Геометрические фигуры

Рис. 2.9. Пример алфавитов, поддерживаемых в Unicode. Unicode версии 4.0 имеет более 160 коллекций символов, называемых «блоками». Каждый блок кратен числу 16. Например, Греческий (Greek) начинается с 0370_{16}, а Кириллический (Cyrillic) начинается с 0400_{16}. Первые три столбца показывают 48 блоков, соответствующих обычным языкам, которые расположены примерно в том порядке, в котором они следуют в Unicode. Последние столбцы имеют 16 блоков, являющихся мультиязычными, и идут не по порядку. 16-битная кодировка, называемая UTF-16, используется по умолчанию. Кодировка переменной длины, называемая UTF-8, сохраняет подмножество ASCII как восьмибитный и использует от 16 до 32 бит для других символов. UTF-32 использует по 32 бита на символ. Чтобы получить дополнительную информацию, следует зайти на сайт www.unicode.org

Самопроверка

1. Какие из следующих утверждений о символах и строках в C и Java соответствуют действительности?
   1. Строка в C занимает вдвое меньше памяти по сравнению с такой же строкой в Java.
   2. Строки — это всего лишь неформальное название одномерных массивов символов в C и Java.
   3. Строки в C и Java используют null (0) для маркировки конца строки.

4. Операции со строками, такие как определение длины, в C работают быстрее, чем в Java.

II. Какой тип переменной, содержащей $1\,000\,000\,000_{10}$, занимает больше памяти?

1. int в C.
2. string в C.
3. string в Java.

2.10. Адресация MIPS для 32-разрядных непосредственных значений и адресов

Несмотря на то что общая для всех MIPS-инструкций длина в 32 разряда упрощает конструкцию оборудования, бывает так, что удобнее было бы иметь 32-разрядные константы или 32-разрядные адреса. Этот раздел начинается с общего решения задачи создания больших констант, а затем в нем показывается оптимизация адресов инструкций, используемых в переходах.

32-разрядные непосредственные операнды

Хотя обычно константы имеют небольшой размер и помещаются в 16-разрядное поле, иногда встречаются и более длинные. Набор инструкций MIPS включает инструкцию *загрузки непосредственного значения в верхние 16 разрядов* — **load upper immediate** (`lui`), которая специально предназначена для того, чтобы поместить верхние 16 разрядов константы в регистр, позволяя следующей инструкции указать нижние 16 разрядов константы. Действия инструкции `lui` показаны на рис. 2.10.

Упражнение

Разгрузка 32-разрядной константы

Каким будет ассемблерный код MIPS для загрузки 32-разрядной константы в регистр `$s0`?

```
0000 0000 0011 1101 0000 1001 0000 0000
```

Ответ

Сначала нужно, используя инструкцию `lui`, загрузить верхние 16 разрядов, которые в десятичной форме имеют вид 61.

```
lui $s0, 61          # 61 в десятичной форме = 0000 0000 0011 1101 в двоичной
```

После этого значением регистра `$s0` станет число

```
0000 0000 0011 1101 0000 0000 0000 0000
```

Следующим шагом будет вставка нижних 16 разрядов, которые в десятичной форме имеют вид 2304:

```
ori $s0, $s0, 2304    # 2304 в десятичной форме = 0000 1001 0000 0000
```

В конечном итоге регистр `$s0` приобретет требуемое значение:

```
0000 0000 0011 1101 0000 1001 0000 0000
```

Машинная версия инструкции lui $t0, 255 # $t0 это регистр 8:

001111	00000	01000	0000 0000 1111 1111

Содержимое регистра $t0 после выполнения lui $t0, 255:

0000 0000 1111 1111	0000 0000 0000 0000

Рис. 2.10. Результат работы инструкции `lui`. Инструкция `lui` переносит 16-разрядов значения поля непосредственной константы в левые 16 разрядов регистра, заполняя нижние 16 разрядов нулями

Интерфейс аппаратного и программного обеспечения

Разбивать большие константы на части с последующей их сборкой в регистре должен либо компилятор, либо ассемблер. Неудивительно, что ограничения, наложенные на размер непосредственного поля, могут стать проблемой не только для адресации памяти при загрузке и сохранении, но и для констант в инструкциях, использующих непосредственные данные. Если эта задача возлагается на ассемблер, как это делается в программном обеспечении MIPS, то ассемблер должен иметь доступный временный регистр, в котором создаются длинные значения. Именно для этого и предназначен регистр `$at`, зарезервированный для ассемблера.

Следовательно, символьное представление машинного языка MIPS больше не ограничено конструкцией оборудования, а ограничивается только тем, что создатель ассемблера выбрал для включения в свой код (см. раздел 2.12). Рассматривая архитектуру компьютера, мы тесно привязываемся к оборудованию, обращая внимание на те случаи, где используется расширенный язык ассемблера, не обусловленный конструкцией процессора.

Уточнение. Создание 32-разрядных констант требует особого внимания. Инструкция `addi` копирует самый левый разряд 16-разрядного поля непосредственных данных в верхние 16 разрядов слова. *Логическая инструкция непосредственного ИЛИ* из раздела 2.6 загружает нули в верхние 16 разрядов, и поэтому при создании 32-разрядной константы именно она используется ассемблером в связке с инструкцией `lui`.

Адресация при условных и безусловных переходах

В инструкциях безусловного перехода MIPS задействуется самая простая адресация. В них используется последний формат MIPS-инструкции, который называется *J-тип*. Он состоит из 6 разрядов для поля операции и всех остальных разрядов для поля адреса. Таким образом, инструкция

```
j 10000        # перейти на адрес 10000
```

может быть преобразована ассемблером в следующий формат (на самом деле этот формат, как мы увидим далее, немного сложнее):

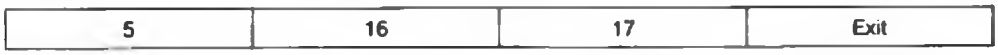
2	10000
6 разрядов	26 разрядов

где значение кода операции (opcode) перехода равно 2, а адрес перехода равен 10000.

В отличие от инструкции безусловного перехода инструкция условного перехода вдобавок к адресу перехода должна указать два операнда. Таким образом, инструкция

```
bne $s0,$s1,Exit    # перейти на метку Exit, если $s0 ≠ $s1
```

ассемблируется в следующую инструкцию, где для адреса перехода остается только 16 разрядов:



6 разрядов | 5 разрядов | 5 разрядов | 16 разрядов

Если адреса программы должны помещаться в это 16-разрядное поле, то это будет означать, что программы не могут быть больше 2^{16}, что на сегодняшний день является слишком малым значением. Альтернативой могло бы стать указание регистра, который бы всегда добавлялся к адресу условного перехода, чтобы инструкция делала следующее вычисление:

```
Счетчик команд = Регистр + Адрес перехода
```

Эта сумма позволяет программе быть больше 2^{32} и по-прежнему использовать условные переходы, решая, таким образом, проблему размера адреса перехода. Возникает вопрос: какой регистр использовать?

Ответ дает наблюдение за тем, как используется условный переход. Этот переход используется в циклах и в инструкциях `if`, поэтому он чаще всего осуществляется на какую-нибудь не слишком отдаленную инструкцию. Например, около половины всех условных переходов в контрольных задачах SPEC осуществляются не далее чем на 16 инструкций. Поскольку счетчик команд (PC) содержит адрес текущей инструкции, можно осуществлять переход в пределах $\pm 2^{15}$ слов от текущей инструкции, если PC будет использоваться в качестве регистра, добавляемого к адресу. Почти все циклы и инструкции `if` намного меньше 2^{16} слов, поэтому PC является идеальным выбором.

Эта форма адресации перехода называется **относительной адресацией по счетчику команд**. Как будет показано в главе 4, оборудованию удобнее увеличить показания PC заблаговременно, чтобы он указывал на следующую инструкцию. Поэтому MIPS-адрес обычно вычисляется относительно адреса следующей инструкции (PC + 4), а не относительно текущей инструкции (PC).

Как и многие самые современные компьютеры, MIPS использует относительную адресацию по счетчику команд для всех условных переходов, поскольку место назначения этих инструкций будет, скорее всего, не далеко от места начала перехода. С другой стороны, инструкция безусловного перехода и ссылки (jump-and-link) вызывает процедуры, которым не имеет смысла быть близкими к месту вызова, поэтому в них, как правило, используются другие виды адресации. Поэтому MIPS-архитектура предлагает длин-

Относительная адресация по счетчику команд
Режим адресации, в котором адрес является суммой показания счетчика команд (PC) и константы, имеющейся в инструкции.

ные адреса для вызова процедур, в которых используется формат J-типа, как для инструкции безусловного перехода, так и для инструкции безусловного перехода и ссылки.

Поскольку все MIPS-инструкции имеют длину 4 байта, MIPS удлиняет дистанцию условного перехода за счет того, что относительная адресация по счетчику команд ссылается на количество *слов* до следующей инструкции, а не на количество байтов. Таким образом, 16-разрядное поле может задать условный переход в четыре раза дальше, интерпретируя содержимое поля как относительный адрес слова, а не относительный адрес байта. Содержимое 26-разрядного поля в инструкции безусловного перехода также является адресом слова, что означает, что оно представляет 28-разрядный адрес байта.

Уточнение. Так как счетчик имеет размер 32 разряда, то для безусловных переходов 4 разряда должны поступать из какого-нибудь другого места. Инструкция безусловного перехода MIPS заменяет только нижние 28 разрядов счетчика команд, оставляя верхние 4 разряда этого счетчика без изменений. Загрузчик и компоновщик (раздел 2.12) должны избегать выхода программы за границу адреса 256 Мбайт (64 миллиона инструкций); в противном случае безусловный переход должен быть заменен переходом по регистру, перед которым находятся другие инструкции для загрузки в регистр полного 32-разрядного адреса.

Упражнение

Демонстрация смещения перехода на машинном языке

Цикл while в упражнении подраздела «Циклы» был скомпилирован в следующий ассемблерный код MIPS:

```
Loop: sll    $t1,$s3,2     # Временный регистр $t1 = 4 * i
      add  $t1,$t1,$s6      # $t1 = адресу save[i]
      lw   $t0,0($t1)       # Временный регистр $t0 = save[i]
      bne  $t0,$s5, Exit    # переход на метку Exit, если save[i] ≠ k
      addi $s3,$s3,1        # i = i + 1
      j    Loop                 # переход на метку Loop
Exit:
```

Если предположить, что начало цикла помещено в памяти по адресу 80000, то каким будет машинный код MIPS для этого цикла?

Ответ

Вот как выглядят инструкции ассемблера и их адреса:

80000	0	0	19	9	2	0
80004	0	9	22	9	0	32
80008	35	9	8	0		
80012	5	8	21	2		
80016	8	19	19	1		
80020	2	20000				
80024	...					

Следует помнить, что инструкции MIPS имеют байтовые адреса, поэтому адреса последовательных слов отличаются друг от друга на 4, то есть на число байтов в слове. Инструкция bne в четвертой строке прибавляет к адресу следующей инструкции (80016) 2 слова или 8 байт, задавая место назначения условного перехода относительно этой следующей инструкции (8 + 80016), а не относительно инструкции условного перехода или перехода с использованием полного адреса назначения (80024). Инструкция безусловного перехода в последней строке использует полный адрес (20000 × 4 = 80000), соответствующий метке Loop.

Интерфейс аппаратного и программного обеспечения

Большинство условных переходов нацелены на близлежащие адреса, но иногда условный переход осуществляется очень далеко, дальше, чем это может быть представлено в 16 разрядах инструкции условного перехода. Здесь на выручку приходит ассемблер, который поступает так же, как он это делает с большими адресами или константами: он вставляет безусловный переход на место назначения условного перехода и инвертирует условие, чтобы условный переход решал, нужно ли пропустить безусловный переход.

Упражнение

Условный переход на большое расстояние

Возьмите условный переход при равенстве значений регистров $s0 и $s1,

```
beq $s0, $s1, L1
```

и замените его двумя инструкциями, предлагающими намного более длинное расстояние перехода.

Ответ

Условный переход на небольшое расстояние заменяется следующими инструкциями:

```
      bne $s0, $s1, L2
      j L1
L2:
```

Краткие выводы по режимам адресации MIPS

Все многообразие форм адресации в целом называется **режимами адресации**. На рис. 2.11 показано, как идентифицируются операнды для каждого режима адресации. В MIPS используются следующие режимы адресации:

1. *Непосредственная адресация*, где операндом является константа внутри самой инструкции.
2. *Регистровая адресация*, где операндом служит регистр.
3. *Базовая адресация, или адресация со смещением*, где операнд является местом в памяти, чей адрес представлен суммой регистра и константы в инструкции.

Режим адресации
Один из нескольких режимов адресации, отличающийся от всех остальных режимов другим способом использованием операндов и (или) адресов.

4. *Относительная адресация по счетчику команд*, где адрес условного перехода является суммой значения счетчика команд и константы в инструкции.
5. *Псевдонепосредственная адресация*, где адрес перехода представлен 26 разрядами инструкции, объединенными с верхними разрядами счетчика команд.

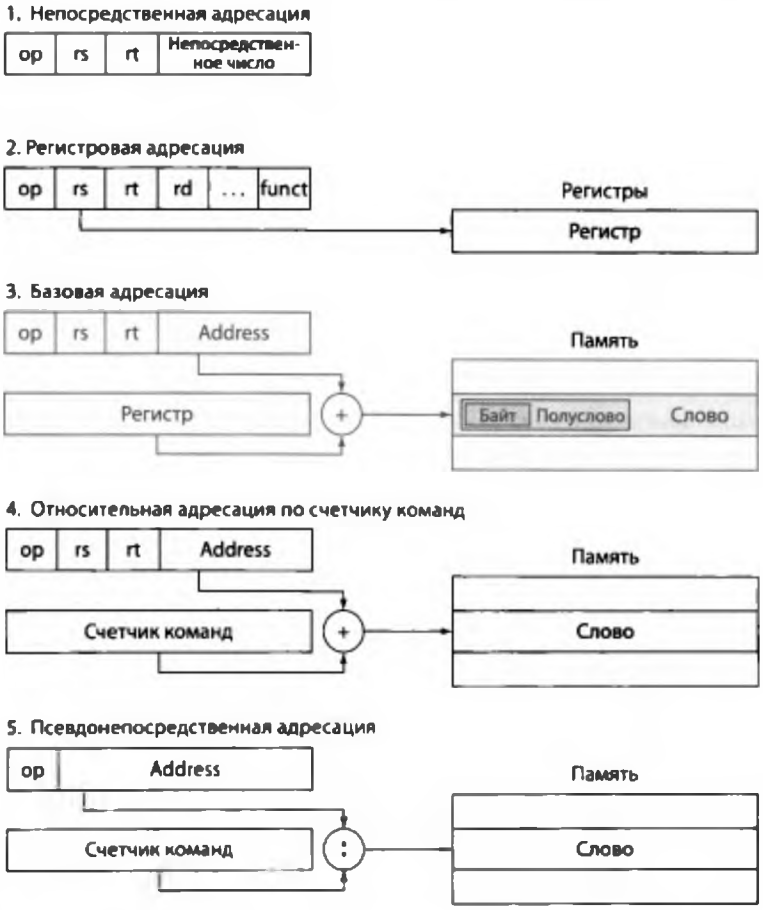


Рис. 2.11. Иллюстрация пяти режимов адресации MIPS. Операнды выделены темным фоном. Операнд в режиме 3 находится в памяти, а операнд в режиме 2 находится в регистре. Заметьте, что версии загрузки и сохранения имеют доступ к байтам, полусловам или словам. Для режима 1 операндом является 16 разрядов самой инструкции. Режимы 4 и 5 обращаются к инструкциям в памяти, где режим 4 добавляет 16-разрядный адрес, сдвинутый влево на два разряда к счетчику команд, а режим 5 объединяет 26-разрядный адрес, сдвинутый влево на два разряда с четырьмя верхними разрядами счетчика команд

Интерфейс аппаратного и программного обеспечения

Хотя мы показываем, что в MIPS имеется 32-разрядная адресация, почти все микропроцессоры (включая MIPS) обладают 64-разрядными адресными расширениями. Эти расширения стали ответом на потребности в программном обеспечении

для более крупных программ. Процесс развития набора инструкций позволяет архитектуре расширяться таким образом, чтобы программное обеспечение сохраняло совместимость и передавалось по наследству следующему поколению архитектуры.

Следует заметить, что отдельная операция может использовать более одного режима адресации. Сложение, к примеру, использует как непосредственную (`addi`), так и регистровую (`add`) адресацию.

Декодирование машинного языка

Иногда приходится выполнять обратное преобразование машинного языка для воссоздания исходного кода на языке ассемблера. Один из примеров — просмотр так называемого «дампа памяти». На рис. 2.12 показано MIPS-кодирование полей для машинного языка MIPS. Этот процесс помогает при ручном транслировании, проводимом между языком ассемблера и машинным языком.

Упражнение

Декодирование машинного кода

Какая инструкция на языке ассемблера соответствует этой машинной инструкции?

```
00af802016
```

Ответ

Первый шаг заключается в конвертировании шестнадцатеричного числа в двоичное, чтобы найти поле `op`:

```
(Разряды  31 28 26                          5  2 0)
          0000 0000 1010 1111 1000 0000 0010 0000
```

Этот поиск нужен для определения операции. Согласно схеме, показанной на рис. 2.12, когда биты 31–29 содержат 000, а биты 28–26 также содержат 000, инструкция имеет R-формат. Давайте переформатируем двоичную инструкцию в поля R-формата, перечисленные в табл. 2.8:

```
op      rs     rt     rd     shamt  funct
000000  00101  01111  10000  00000  100000
```

В нижней части рис. 2.12 дается определение операциям инструкций R-формата. В данном случае биты 5–3 содержат 100, а биты 2–0 содержат 000, что означает, что эта двоичная комбинация представляет собой инструкцию `add`.

Остальная часть инструкции декодируется путем изучения значений полей. Эти поля имеют следующие десятичные значения: поле `rs` — 5, поле `rt` — 15 и поле `rd` — 16 (поле `shamt` не используется). В табл. 2.7 показано, что эти числа представляют регистры `$a1`, `$t7` и `$s0`. Теперь можно воспроизвести всю ассемблерную инструкцию:

```
add $s0,$a1 $t7
```

op(31:26)

28-26 31-29	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(000)	R-format	Bltz/gez	jump	jump & link	branch eq	branch ne	blez	Dgtz
1(001)	add immediate	addi u	set less than imm.	set less than imm. unsigned	andi	Ori	xori	load upper immediate
2(010)	TLB	FIPt						
3(011)								
4(100)	1oad byte	load half	lwl	load word	load byte unsigned	load half unsigned	lwr	
5(101)	store byte	store half	swl	store word			swr	
6(110)	load linked word	lwcl						
7(111)	store cond. word	swcl						

op(31:26)=010000 (TLB), rs(25:21)

23-21 25-24	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(00)	Mfc0		Cfc0		Mtc0		Ctc0	
1(01)								
2(10)								
3(11)								

op(31:26)=000000 (R-format), funct(5:0)

2-0 5-3	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(000)	shift left logical		shift right logical	sra	sllv		srlv	srav
1(001)	jump register	jalr			syscall	break		
2(010)	mfhi	mthi	mflo	mtlo				
3(011)	mult	multu	div	divu				
4(100)	add	addu	sub-tract	subu	and	or	xor	not or (nor)
5(101)			set l.t.	set l.t. unsigned				
6(110)								
7(111)								

Рис. 2.12. Кодировка инструкций MIPS. Эта система обозначений дает значение поля по строкам и столбцам. Например, верхняя часть рисунка показывает инструкцию загрузки слова — load word в строке 4 (100_2 для разрядов 31–29 инструкции) и в столбце 3 (011_2 для разрядов 28–26 инструкции), то есть это соответствует значению поля op (разряды 31–26), равному 100011_2. Выделение означает, что поле используется где-нибудь еще. Например, R-format в строке 0 и столбце 0 (op = 000000_2) определяется в нижней части рисунка. Следовательно, присутствие инструкции вычитания — subtract с строке 4 и в столбце 2 в нижней части означает, что поле funct (разряды 5–0) инструкции имеет значение 100010_2, а поле op (разряды 31–26) имеет значение 000000_2. Значение плавающей точки — floating point (FIPt) в строке 2, столбце 1 определено на рис. 3.14 в главе 3. А значение Bltz/gez является кодом операции (opcode) для четырех инструкций: bltz, bgez, bltzal и bgezal. В этой главе рассматриваются инструкции, представленные полным именем и выделенные жирным шрифтом, а в главе 3 рассматриваются инструкции, также выделенные жирным шрифтом, но представленные в сокращенной форме

Таблица 2.8. Форматы MIPS-инструкций

Имя	Поля						Комментарии
Размер поля	6 р.	5 р.	5 р.	5 р.	5 р.	6 р.	Все MIPS-инструкции имеют длину 32 разряда
R-формат	op	rs	rt	rd	shamt	funct	Формат арифметических инструкций
I- формат	op	rs	rt	address / immediate (адрес / непосредственное значение)			Формат инструкций переноса, условного перехода, а также инструкций, использующих непосредственные значения
J- формат	op	target address (целевой адрес)					Формат инструкции безусловного перехода

В табл. 2.8 показаны все форматы MIPS-инструкций. В табл. 2.1 показаны инструкции языка ассемблера, рассматриваемые в данной главе. Не показанная в ней часть MIPS-инструкций работает главным образом с арифметикой и с вещественными числами и будет рассмотрена в следующей главе.

Самопроверка

I. Каков диапазон адресов для условных переходов в MIPS (K = 1024)?

1. Адреса между 0 и 64 K − 1.
2. Адреса между 0 и 256 K − 1.
3. Адреса приблизительно на 32 K вверх, до перехода, и приблизительно на 32 K вниз после него.
4. Адреса приблизительно на 128 K вверх, до перехода, и приблизительно на 128 K вниз после него.

II. Каков диапазон адресов для безусловного перехода (jump) и безусловного перехода со ссылкой (jump and link) в MIPS (M = 1024 K)?

1. Адреса между 0 и 64 M − 1.
2. Адреса между 0 и 256 M − 1.
3. Адреса приблизительно на 32 M вверх до перехода, и приблизительно на 32 M вниз после него.
4. Адреса приблизительно на 128 M вверх до перехода, и приблизительно на 128 M вниз после него.
5. Переход осуществляется в любое место внутри блока из 64 M адресов, когда счетчик команд предоставляет верхние 6 разрядов.
6. Переход осуществляется в любое место внутри блока из 256 M адресов, когда счетчик команд предоставляет верхние 4 разряда.

III. Какая инструкция языка ассемблера MIPS соответствует инструкции на машинном языке, имеющей значение $0000\ 0000_{16}$?

1. j.
2. R-формат.

3. `addi`.
4. `sll`.
5. `mfc0`.
6. Неопределенный код операции (opcode): допустимой инструкции, соответствующей значению 0, не существует.

2.11. Параллелизм и инструкции: синхронизация

При независимых задачах осуществить параллельное выполнение намного проще, но чаще всего задачи нуждаются во взаимодействии. Это взаимодействие обычно означает, что некоторые задачи записывают новые значения, которые должны быть прочитаны другими задачами. Чтобы узнать, что задача завершила запись и другая задача может свободно прочитать данные, задачи нуждаются в синхронизации. Если они не будут синхронизированы, появится опасность возникновения **соревнования за доступ к данным (data race)**, при которой результат работы программы может измениться в зависимости от того, как будут складываться события.

Вспомним, к примеру, аналогию из главы 1 с восемью репортерами, которые пишут статью. Предположим, одному репортеру перед тем, как написать заключение, нужно прочитать все предыдущие разделы. Следовательно, он должен знать, когда все остальные репортеры завершат работу над своими разделами, чтобы ему не пришлось переживать о том, что эти разделы не будут впоследствии изменены. Таким образом, должны быть синхронизированы запись и чтение каждого раздела, чтобы заключение согласовывалось с публикациями предыдущих разделов.

В компьютерных вычислениях механизмы синхронизации обычно создаются вместе с подпрограммами, относящимися к программам на уровне пользователя, которые зависят от использования инструкций синхронизации, предоставленных оборудованием. В данном разделе основное внимание будет уделено реализации операций синхронизации *lock* (*блокировка*) и *unlock* (*снятие блокировки*). Блокировка и разблокировка могут применяться непосредственно для создания областей, где может работать только один процессор, то есть обеспечивать так называемое *взаимное исключение* (*mutual exclusion*), а также использоваться для реализации более сложных механизмов синхронизации.

Для осуществления синхронизации в многопроцессорной системе требуется обязательное присутствие набора аппаратных примитивов, дающих возможность *атомарного* чтения и модификации места в памяти. Расчет делается на то, что ничто иное не может вклиниться между чтением и записью места в памяти. В отсутствие такой возможности цена создания базисных примитивов синхронизации окажется слишком высокой и будет повышаться с ростом числа процессоров.

Соревнование за доступ к данным (data race)
Соревнование за доступ к данным возникает в том случае, если из двух разных потоков осуществляется доступ к одному и тому же месту в памяти и если как минимум один из этих потоков ведет запись данных, а доступ происходит последовательно.

Существуют и другие формулировки базисных аппаратных примитивов, каждый из которых предоставляет возможность атомарного чтения и модификации места в памяти в совокупности с некими способами сообщения о том, что чтение и запись были проведены в атомарном режиме. В целом создатели архитектуры не предполагают, что пользователи будут применять базисные аппаратные примитивы, они считают, что эти примитивы будут востребованы системными программистами для создания библиотеки синхронизации, что является весьма сложным делом.

Давайте начнем с одного из таких аппаратных примитивов и покажем, как он может использоваться для построения базисного примитива синхронизации. Одним из типичных действий для создания операций синхронизации является *атомарный обмен* (*atomic exchange* или *atomic swap*), при котором меняются местами значение регистра и значение ячейки памяти.

Чтобы посмотреть, как это можно использовать для создания базисного примитива синхронизации, предположим, что нам нужно создать простую блокировку, где 0 будет свидетельствовать о доступности, а 1 — о недоступности блокировки. Процессор пытается установить блокировку путем осуществления обмена единицы, содержащейся в регистре, со значением ячейки памяти, которая отвечает за блокировку. Значение, возвращенное инструкцией обмена, будет равно единице, если доступ уже был запрошен каким-нибудь из процессоров, и нулю в противном случае. Во втором варианте значение также изменяется на единицу, предотвращая любые другие конкурирующие обмены со стороны другого процессора, который также мог бы извлечь нуль.

Рассмотрим, к примеру, два процессора, каждый из которых пытается одновременно с другим процессором провести обмен: соревнования между ними не будет, потому что только один из процессоров проведет обмен первым, возвращая нуль, а второй процессор, когда первый проведет этот обмен, вернет единицу. Ключевым моментом использования примитива обмена для осуществления синхронизации является атомарность операции: обмен неделим, и два одновременных обмена будут поставлены оборудованием в очередь. Два процессора, пытающиеся установить переменную синхронизации, установить ее одновременно не смогут.

Реализация отдельной атомарной операции с памятью создает ряд трудностей при разработке процессора, поскольку она требует как чтения из памяти, так и записи в нее путем выполнения одной непрерывной инструкции.

Альтернативный вариант предусматривает наличие пары инструкций, где вторая инструкция возвращает значение, показывающее, имела ли выполненная пара инструкций атомарный характер. Пара инструкций по-настоящему атомарна, если окажется, что все другие операции, выполненные любым из процессоров, произошли до или после выполнения этой пары инструкций. Таким образом, когда пара инструкций действительно имеет свойства атомарности, никакой другой процессор не сможет изменить значение, вклинившись между этой парой инструкций.

В MIPS такая пара инструкций включает специальную загрузку, называемую связанной загрузкой, — *load linked*, и специальное сохранение, называемое условным сохранением, — *store conditional*. Эти инструкции используются одна за другой: если содержимое места памяти, указанное в связанной загрузке, изменить до того, как будет произведено условное сохранение в то же место памяти, условное сохранение потерпит неудачу. Условное сохранение отличается тем, что оно сохраняет значение регистра в памяти *и* меняет значение этого регистра на единицу, если выполняется успешно, и на нуль, если терпит неудачу. Поскольку связанная загрузка возвращает исходное значение и условное сохранение возвращает единицу только при успешном выполнении, следующая последовательность выполняет атомарный обмен с местом памяти, указанном в содержимом регистра `$s1`:

```
try: add $t0,$zero,$s4   ;копирование значения обмена
     ll $t1,0($s1)       ;связанная загрузка
     sc $t0,0($s1)       ;условное сохранение
     beq $t0,$zero,try   ;переход, если сохранение не удалось
     add $s4,$zero,$t1   ;помещение загруженного значения в $s4
```

По завершении работы этой последовательности будет произведен атомарный обмен содержимого `$s4` и места в памяти, указанного с помощью регистра `$s1`. При любом вмешательстве процессора и модификации значения в памяти между инструкциями `ll` и `sc` инструкция `sc` возвращает в `$t0` значение 0, что заставляет кодовую последовательность предпринять повторную попытку.

Уточнение. Несмотря на то что атомарный обмен представлен для многопроцессорной синхронизации, он также применяется для операционных систем при обработке нескольких процессов на одном процессоре. Чтобы удостовериться, что при работе одного процессора ничто друг другу не противоречит, условное сохранение также терпит неудачу, если процессор между этими двумя инструкциями изменил содержимое (см. главу 5).

Поскольку условное сохранение может потерпеть неудачу после любого другого предпринятого сохранения в то место, на которое указывает связанная загрузка, нужно проявлять особое внимание при выборе инструкций, вставляемых между этими двумя инструкциями. В частности, безо всяких опасений могут быть разрешены только инструкции типа регистр-регистр; все остальные инструкции способны привести к ситуации *взаимной блокировки* (*deadlock*), когда процессор никогда не сможет завершить инструкцию `sc` из-за повторяющейся ошибки отказа при обращении к странице. Кроме того, количество инструкций между связанной загрузкой и условным сохранением должно быть как можно меньшим, чтобы свести к минимуму вероятность того, что либо не связанное с этим процессом событие, либо конкурирующий процессор станут причиной частых неудачных завершений инструкции условного сохранения.

Преимуществом механизма связанной загрузки — условного сохранения — является возможность его использования для создания других примитивов синхронизации, таких как *атомарное сравнение и обмен* или *атомарное извлечение*

и увеличение на единицу, которые используются в некоторых моделях параллельного программирования. В этих примитивах между ll и sc используется большее количество инструкций.

Самопроверка

Когда используются такие примитивы, как связанная загрузка и условное сохранение?

1. В том случае, когда взаимодействующие потоки параллельных программ нуждаются в синхронизации, чтобы правильно считывать и записывать совместно используемые данные.
2. В том случае, когда взаимодействующие процессы на однопроцессорной системе нуждаются в синхронизации при чтении и записи совместно используемых данных.

2.12. Трансляция и запуск программы

В этом разделе рассматриваются четыре этапа превращения программы на языке C в программу, запускаемую на компьютере. На рис. 2.13 показана иерархия трансляции. Некоторые системы объединяют показанные в ней этапы, чтобы сократить время трансляции, но здесь показаны четыре последовательных этапа, через которые проходит программа. В данном разделе используется именно эта иерархия трансляции.

Компилятор

Компилятор превращает программу на языке C в *программу на языке ассемблера*, символьную форму того, что понятно машине. Программы на языках высокого уровня занимают намного меньше строк кода, чем программы на языке ассемблера, поэтому продуктивность работы программиста существенно возрастает.

В 1975 году многие операционные системы и ассемблеры были написаны на **языке ассемблера**, потому что объем памяти был невелик и компиляторы работали неэффективно. Увеличение объема памяти одного DRAM-чипа в 500 000 раз снизило остроту проблемы размера программы, и сегодня оптимизирующие компиляторы способны выдавать программы на языке ассемблера, сопоставимые по качеству с теми, которые создаются специалистами по программированию на этом языке, а для больших программ результат работы компилятора порой превосходит по качеству ручную работу.

Язык ассемблера
Символьный язык, который может транслироваться в двоичный машинный язык.

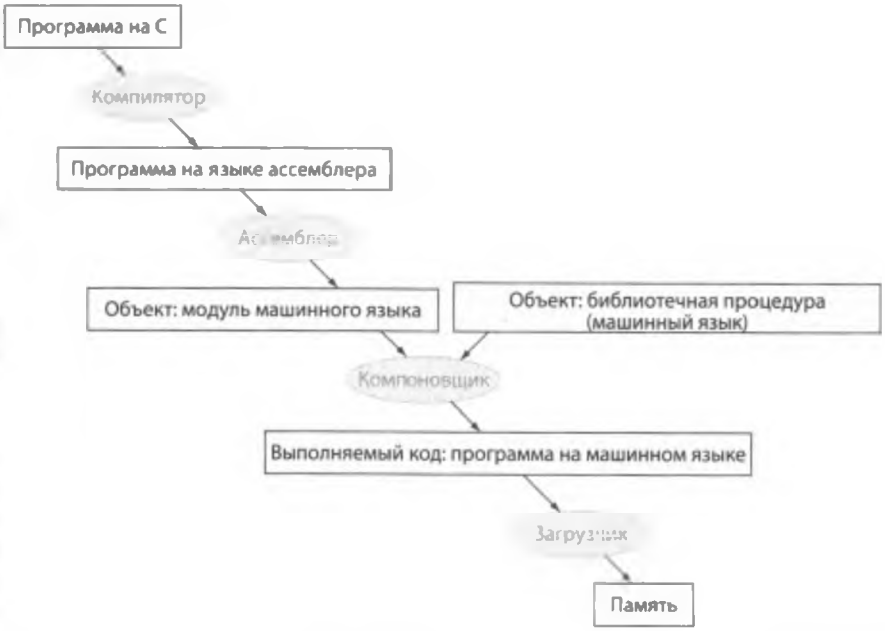


Рис. 2.13. Иерархия трансляции программы на языке С. Сначала программа на языке высокого уровня компилируется в программу на языке ассемблера, а затем собирается в объектный модуль на машинном языке. Для разрешения всех ссылок компоновщик объединяет несколько модулей с библиотечными процедурами. Затем загрузчик помещает машинный код в нужное место памяти для выполнения этого кода процессором. Для ускорения процесса трансляции некоторые этапы пропускаются или объединяются. Некоторые компиляторы сразу производят объектные модули, в то время как другие используют компонующие загрузчики, выполняющие два последних этапа. Чтобы идентифицировать тип файла, в UNIX придерживаются соглашения о суффиксах для файловых имен: исходные С-файлы называются *x*.c, ассемблерные файлы — *x*.s, объектные файлы — *x*.o, файлы статически связанных библиотечных процедур — *x*.a, файлы динамически связанных библиотечных процедур —*x*.so, а исполняемые файлы по умолчанию называются a.out. В MS-DOS для тех же целей используются суффиксы .C, .ASM, .OBJ, .LIB, .DLL и .EXE.

Ассемблер

Язык ассемблера не только служит интерфейсом с программами высокого уровня, но также может обрабатывать в качестве вполне самостоятельных общепринятые вариации инструкций машинного языка. Оборудование не должно иметь реализацию этих инструкций; но их появление в языке ассемблера упрощает трансляцию и создание программ. Такие инструкции называются **псевдоинструкциями**.

Псевдоинструкции
Общепринятые вариации инструкций на языке ассемблера, часто обрабатываемые как вполне самостоятельные инструкции.

Как уже ранее упоминалось, оборудование MIPS обеспечивает постоянное присутствие нулевого значения в регистре $zero. То есть где

бы ни использовался регистр $zero, он предоставляет значение нуль, и программист не может изменить значение этого регистра. Регистр $zero используется для создания ассемблерной инструкции move, которая копирует содержимое одного регистра в другой. Таким образом, ассемблер MIPS принимает эту инструкцию, не обращая внимания на ее отсутствие в MIPS-архитектуре:

```
move $t0,$t1          # регистр $t0 получает значение регистра $t1
```

Ассемблер превращает эту инструкцию на языке ассемблера в эквивалент машинного языка следующей инструкции:

```
add  $t0,$zero,$t1    # регистр $t0 получает значение 0 + значение регистра $t1
```

Ассемблер MIPS также превращает инструкцию перехода «если меньше чем» — blt (branch on less than) — в две инструкции: slt и bne, упоминавшиеся в упражнении раздела 2.10. Другие примеры включают инструкции bgt, bge и ble. Ассемблер также превращает условные переходы на слишком отдаленные адреса в инструкции условных и безусловных переходов. Как уже упоминалось, ассемблер MIPS позволяет 32-разрядным константам загружаться в регистр, несмотря на 16-разрядное ограничение, налагаемое на инструкции, использующие непосредственные значения.

В целом псевдоинструкции обогащают набор инструкций языка ассемблера MIPS по сравнению с тем набором, для которого имеется аппаратная реализация. Единственной ценой этого обогащения является резервирование одного регистра, $at, для использования ассемблером. Если вы собираетесь создавать программы на ассемблере, то для упрощения своих задач пользуйтесь псевдоинструкциями. Но чтобы разобраться в MIPS-архитектуре и обеспечить наилучшую производительность, нужно изучить настоящие MIPS-инструкции, показанные в табл. 2.1 и на рис. 2.12.

Ассемблеры принимают числа с разными основаниями. В дополнение к двоичным и десятичным они обычно принимают числа по основанию, позволяющему представить их в более сжатой форме, чем числа по основанию 2, и легко преобразовать их в двоичную комбинацию. Ассемблер MIPS использует шестнадцатеричные числа.

Эти свойства делают язык удобнее, но первостепенной задачей ассемблера является сборка программы в машинный код. Ассемблер превращает программу на языке ассемблера в *объектный файл*, представляющий собой комбинацию из инструкций на машинном языке, данных и информации, необходимой для правильного помещения инструкций в память.

Для создания двоичной версии каждой инструкции, имеющейся в программе на языке ассемблера, программа ассемблера должна определить адреса, соответствующие всем меткам.

Ассемблер отслеживает метки, используемые в переходах и инструкциях переноса данных, в **таблице имен**. Нетрудно догадаться, что таблица состоит из пар обозначений и адресов.

Таблица имен
Таблица соответствий имен меток адресам тех слов в памяти, в которых находятся соответствующие инструкции.

Объектный файл для UNIX-систем обычно состоит из шести отдельных частей:

- *Заголовка объектного файла*, в котором описывается размер и позиции всех остальных частей этого файла.

- *Текстового сегмента*, в котором содержится код на машинном языке.
- *Сегмента статических данных*, в котором содержатся данные, размещенные для работы программы. (UNIX позволяет программам использовать как статические данные, размещаемые по всей программе, так и динамические данные, объем которых может увеличиваться и уменьшаться в соответствии с потребностями программы — см. рис. 2.7.)
- *Настроечной информации*, которая идентифицирует слова инструкций и данных, зависящие от абсолютных адресов при загрузке программы в память.
- *Таблицы имен*, в которой содержатся оставшиеся метки, не определенные в качестве внешних ссылок.
- *Отладочной информации*, которая содержит краткое описание того, как были откомпилированы модули, чтобы отладчик мог связать машинные инструкции с исходными файлами на языке C и придать структуре данных удобный для чтения вид.

В следующем подразделе показывается, как присоединяются подпрограммы, которые уже были обработаны программой ассемблера, например библиотечные процедуры.

Компоновщик

Исходя из всего ранее представленного, предполагается, что одно-единственное изменение в одной строке одной процедуры потребует компиляции и ассемблирования всей программы. Полная перетрансляция приводит к лишней трате компьютерных ресурсов. Повторная трансляция не нужна, в частности, стандартным библиотечным процедурам, поскольку программист будет компилировать и ассемблировать процедуры, которые по определению практически никогда не изменяются. Альтернативный вариант заключается в независимой компиляции каждой процедуры, чтобы изменение в одной строке потребовало компиляции и ассемблирования только одной процедуры. Для его осуществления нужна новая системная программа, которая называется **редактором связей**, или **компоновщиком**. Эта программа «сшивает» в единое целое все программы на машинном языке, которые были оттранслированы ассемблером независимо друг от друга.

Компоновщик работает в три этапа:

1. Помещает код и данные модулей в память в символьном виде.
2. Определяет адреса данных и метки инструкций.
3. Исправляет как внутренние, так и внешние ссылки.

Для разрешения всех неопределенных меток компоновщик использует настроечную информацию и таблицу имен, присутствующую в каждом объектном модуле. Подобные ссылки имеются в инструкциях условного и безусловного перехода и в адресах данных, то есть работа этой программы во многом напоминает работу

Компоновщик (linker)
Также называется редактором связей (link editor). Системная программа, объединяющая независимо отассемблированные программы на машинном языке в исполняемый файл и разрешающая все неопределенные метки.

редактора: она находит старые адреса и заменяет их новыми. Процесс редактирования и дал программе название «редактор связей», или, для краткости, компоновщик. Компоновщик полезен тем, что исправление кода происходит намного быстрее, чем его перекомпиляция и переассемблирование.

Если все внешние ссылки разрешены, компоновщик определяет места в памяти для каждого модуля. Вспомним соглашение MIPS по распределению памяти под программы и данные, которое было показано на рис. 2.7. Поскольку файлы были ассемблированы в изоляции друг от друга, ассемблер может и не знать, куда должны быть помещены инструкции и данные модуля по отношению к другим модулям. Когда компоновщик помещает модуль в память, все абсолютные ссылки, то есть, адреса памяти, которые не связаны со значением регистра, должны быть изменены, чтобы отражать реальное расположение модуля.

Компоновщик создает **исполняемый файл**, который может быть запущен на компьютере. Обычно этот файл имеет такой же формат, что и объектный файл, за исключением того, что в нем не содержится неразрешенных ссылок. Возможна частичная компоновка таких файлов, как файлы библиотечных процедур, у которых все еще имеются неразрешенные адреса, и поэтому они остаются объектными файлами.

Исполняемый файл
Работоспособная программа в формате объектного файла, в которой не содержатся неразрешенные ссылки. Она может содержать таблицу имен и отладочную информацию. «Раздетые исполняемые файлы» такой информации не содержат. Для загрузчика может быть включена настроечная информация.

Упражнение

Компоновка объектных файлов
Нужно скомпоновать два, показанных ниже, объектных файла. Следует показать обновленные адреса первых нескольких инструкций полноценного исполняемого файла. Инструкции показываются на языке ассемблера только для того, чтобы сделать упражнение понятнее; в реальности инструкции будут представлены числами

Заметьте, что в объектных файлах мы выделили адреса и обозначения, которые должны быть обновлены в процессе компоновки: инструкции, которые ссылаются на адреса процедур `A` и `B`, а также инструкции, которые ссылаются на адреса слов данных `X` и `Y`.

Заголовок объектного файла			
	Имя	Процедура A	
	Размер текста	100_{16}	
	Размер данных	20_{16}	
Текстовый сегмент	Адрес	Инструкция	
	0	lw $a0, **0** ($gp)	
	4	jal **0**	
	...	...	
Сегмент данных	**0**	(**X**)	
	...	...	

Настроечная информация	Адрес	Тип инструкции	Зависимость
	0	lw	**X**
	4	jal	**B**
Таблица имен	Метка	Адрес	
	X	–	
	B	–	
	Имя	Процедура B	
	Размер текста	200_{16}	
	Размер данных	30_{16}	
Текстовый сегмент	Адрес	Инструкция	
	0	sw $al, **0**($gp)	
	4	jal **0**	
	...	...	
Сегмент данных	**0**	(**Y**)	
	...	...	
Настроечная информация	Адрес	Тип инструкции	Зависимость
	0	sw	**Y**
	4	jal	**A**
Таблица имен	Метка	Адрес	
	Y	–	
	A	–	

Ответ

Процедуре A нужно найти адрес для переменной, имеющей метку X, чтобы поместить его в инструкцию загрузки, и найти адрес процедуры B, чтобы поместить его в инструкцию jal. Процедура B нуждается в адресе переменной, имеющей метку Y, для инструкции сохранения и в адресе процедуры A для своей инструкции jal. Согласно рис. 2.7, текстовый сегмент начинается с адреса $40\ 0000_{16}$, а сегмент данных — с адреса $1000\ 0000_{16}$. Текст процедуры A помещается по первому адресу, а ее данные — по второму. Заголовок объектного файла для процедуры A сообщает, что ее текст занимает 100_{16} байт, а ее данные занимают 20_{16} байт, поэтому начальный адрес для текста процедуры B будет равен $40\ 0100_{16}$, а ее данные будут начинаться с адреса $1000\ 0020_{16}$.

Заголовок исполняемого файла		
	Размер текста	300_{16}
	Размер данных	50_{16}
Текстовый сегмент	Адрес	Инструкция

Заголовок исполняемого файла		
	$0040\ 0000_{16}$	lw $a0, **8000**$_{16}$($gp)
	$0040\ 0004_{16}$	jal **40 0100**$_{16}$
	...	...
	$0040\ 0100_{16}$	sw $a1, **8020**$_{16}$($gp)
	$0040\ 0104_{16}$	jal **40 0000**$_{16}$
	...	...
Сегмент данных	Адрес	
	$1000\ 0000_{16}$	(X)
	...	...
	$1000\ 0020_{16}$	(Y)
	...	...

Теперь компоновщик обновит поля address инструкций. Чтобы узнать формат редактируемого адреса, он использует поле, определяющее тип инструкции. Здесь используются поля двух типов:

1. С инструкциями jal разобраться проще, потому что они используют псевдонепосредственную адресацию. Инструкция jal по адресу $40\ 0004_{16}$ получает в свое адресное поле $40\ 0100_{16}$ (адрес процедуры B), а инструкция jal по адресу $40\ 0104_{16}$ получает в свое адресное поле $40\ 0000_{16}$ (адрес процедуры A).
2. С адресами инструкций загрузок и сохранений дело обстоит сложнее, поскольку они вычисляются относительно базового регистра. В данном примере в качестве базового регистра используется глобальный указатель.

На рис. 2.7 показано, что регистр $gp имеет начальное значение $1000\ 8000_{16}$. Для получения адреса $1000\ 0000_{16}$ (адреса слова X) мы помещаем 8000_{16} в адресное поле инструкции lw по адресу $40\ 0000_{16}$. Точно так же мы помещаем 8020_{16} в адресное поле инструкции sw по адресу $40\ 0100_{16}$ для получения адреса $1000\ 0020_{16}$ (адреса слова Y).

Уточнение. Вспомним, что MIPS-инструкции выровнены по границам слов, поэтому инструкция jal пропускает два правых разряда, чтобы расширить свой диапазон адресов. Таким образом, она использует 26 разрядов, чтобы создать 28-разрядный байтовый адрес. Следовательно, действительный адрес содержится в нижних 26 разрядах инструкции jal и имеет в данном примере значение $10\ 0040_{16}$, а не $40\ 0100_{16}$.

Загрузчик

После того как исполняемый файл будет сохранен на диске, операционная система считывает его в память и запускает на выполнение. В UNIX-системах загрузчик выполняет следующие действия:

1. Считывает заголовок исполняемого файла для определения размера текстового сегмента и сегмента данных.

2. Создает достаточное для текста и данных адресное пространство.
3. Копирует инструкции и данные из исполняемого файла в память.
4. Копирует параметры (если таковые имеются) в стек основной программы.
5. Инициализирует регистры машины и устанавливает указатель стека на первое свободное место.
6. Передает управление процедуре запуска, которая копирует параметры в регистры аргументов и вызывает основную процедуру программы. Когда основная процедура возвращает управление, процедура запуска прекращает выполнение программы с помощью системного вызова `exit`.

Динамически подключаемые библиотеки

В первой части данного раздела давалось описание традиционного подхода к компоновке библиотек до запуска программы. Хотя этот статический способ характеризуется наиболее быстрым вызовом библиотечных процедур, у него имеется ряд недостатков:

- Библиотечные процедуры становятся частью исполняемого кода. Если будет выпущена новая версия библиотеки, в которой будут устранены ошибки или осуществлена поддержка нового оборудования, статически скомпонованные программы будут по-прежнему использовать старую версию.
- Он загружает все подпрограммы, имеющиеся в библиотеке, которые вызываются из любых мест исполняемой программы, даже если эти вызовы не состоятся. Относительно программы библиотека может занимать довольно большой объем: например, стандартная библиотека C имеет размер 2,5 Мбайт.

Эти недостатки привели к использованию **динамически подключаемых библиотек** — **dynamically linked libraries (DLL)**, где библиотечные процедуры не скомпонованы и не загружаются, пока программа не будет запущена. Библиотечные процедуры, так же как программа, содержат дополнительную информацию о месте, в котором находятся нелокальные процедуры, и о том, какие у них имена. В начальной версии DLL-библиотек загрузчик запускал динамический компоновщик, используя имеющуюся в файле дополнительную информацию, чтобы найти соответствующие библиотеки и обновить все внешние ссылки.

Недостаток начальной версии DLL-библиотек заключался в том, что компоновке по-прежнему подвергались все библиотечные процедуры, которые могли быть вызваны, а не те из них, которые вызывались в процессе выполнения программы. Это наблюдение привело к использованию версии DLL-библиотек с «ленивой» привязкой процедур, где каждая процедура привязывалась только *после* ее вызова.

Загрузчик (loader)
Системная программа, помещающая объектную программу в оперативную память, чтобы она была готова к выполнению.

Динамически подключаемые библиотеки (DLL)
Библиотечные процедуры, связываемые с программой в процессе ее выполнения.

Как и многие другие нововведения в изучаемой нами области, этот прием полагается на уровень косвенной адресации. Данная технология показана на рис. 2.14. Все начинается с нелокальных процедур в конце программы, кото-

рые называются фиктивными процедурами, на каждую нелокальную процедуру отводится одна запись. Каждая из этих фиктивных записей содержит косвенный переход.

При первом вызове библиотечной процедуры программа вызывает фиктивную запись и следует по косвенному переходу. Этот переход указывает на код, который помещает в регистр номер, позволяющий идентифицировать нужную библиотечную процедуру, а затем перейти к динамическому компоновщику-загрузчику. Этот компоновщик-загрузчик находит нужную процедуру, перенастраивает в ней все ссылки и изменяет адрес в косвенном переходе, помещая туда указатель на эту процедуру. Затем он передает управление процедуре. Когда процедура завершит свою работу, он возвращает управление в исходное место вызова. Впоследствии вызов библиотечной процедуры приведет к переходу непосредственно к процедуре без осуществления дополнительных перенаправлений.

В общем, DLL-библиотеки требуют дополнительного пространства для информации, необходимой для динамической компоновки, но не требуют копирования или привязки целиком всех библиотек. При первом вызове процедуры приходится смириться с существенными издержками, но зато впоследствии этот косвенный переход так и останется единственным. Следует заметить, что возвращение из библиотеки дополнительных издержек не требует. Операционная система Windows корпорации Microsoft очень сильно зависит от динамически подключаемых библиотек, они также используются по умолчанию и при выполнении программ на современных UNIX-системах.

Запуск программ на языке Java

Все, что рассматривалось ранее, касалось традиционной модели выполнения, где ставка делалась на более короткое время выполнения программы, предназначенной для определенной архитектуры набора команд или даже для определенной реализации этой архитектуры. Программы на языке Java можно, конечно, выполнять точно так же, как и программы на языке C. Но язык Java был придуман с другими целями. Одной из этих целей был надежный запуск программ на любом компьютере, невзирая на более медленное их выполнение.

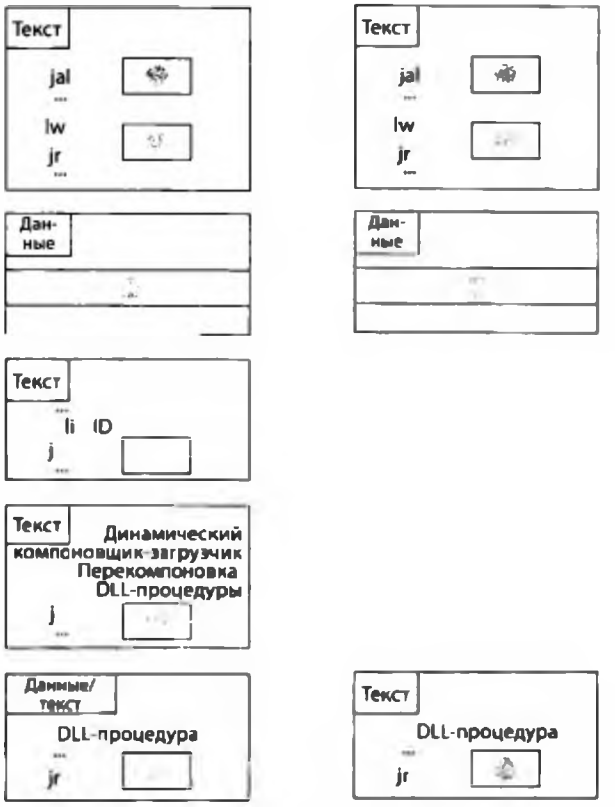
На рис. 2.15 показаны типичные для программы на языке Java этапы трансляции и выполнения. Вместо того чтобы компилировать эту программу на язык ассемблера целевого компьютера, Java компилирует ее сначала в инструкции, которые проще интерпретировать: в набор инструкций **байт-кода Java**. Этот набор инструкций был разработан для того, чтобы быть ближе к языку Java и упростить этап компиляции. Фактически на этом этапе не проводится никакой оптимизации. Как и C-компилятор, Java-компилятор проверяет тип данных и генерирует нужные операции для каждого типа. Java-программы распространяются в двоичной версии этого байт-кода.

Байт-код Java
Инструкция из набора, разработанного для интерпретации Java-программ.

Виртуальная машина Java (JVM)
Программа, интерпретирующая байт-код Java.

Выполнять байт-коды Java может программный интерпретатор, который называется **виртуальной машиной Java**, — Java Virtual Machine (JVM). Интерпретатор — это программа, эму-

рующая архитектуру набора инструкций. Например, MIPS-эмулятор, используемый при написании данной книги, является интерпретатором. Отдельный этап трансляции не требуется, потому что либо трансляция настолько проста, что компилятор проставляет адреса, либо JVM находит эти адреса в процессе выполнения программы.



а) первый вызов DLL-процедуры б) последующие вызовы DLL-процедуры

Рис. 2.14. Динамически подключаемая библиотека с ленивой привязкой процедур: а) этапы первого вызова DLL-процедуры; б) этапы поиска процедуры, ее перекомпоновки и привязки при последующих вызовах пропускаются. Как будет показано в главе 5, операционная система может избежать копирования нужной процедуры путем перекомпоновки ее с использованием системы управления виртуальной памятью

Положительной стороной интерпретации является переносимость. Доступность программного обеспечения виртуальных машин Java означала, что многие люди получили доступ к созданию и запуску Java-программ почти сразу же после первого представления этого языка. Сегодня виртуальные машины Java имеются в сотнях миллионов устройств, начиная с сотовых телефонов и заканчивая интернет-браузерами.

Недостатком интерпретации является более низкая производительность. Невероятный прогресс производительности в 1980-х и в 1990-х годах сделал интерпретаторы вполне приемлемыми для многих важных приложений, но почти

десятикратное замедление по сравнению с традиционно откомпилированными С-программами делает язык Java для некоторых приложений малопривлекательным.

Рис. 2.15. Иерархия трансляции Java-программы. Сначала программа на языке Java компилируется в двоичную версию байт-кода Java со всеми адресами, определенными компилятором. После этого Java-программа готова к запуску в интерпретаторе, который называется виртуальной машиной Java (JVM). Эта виртуальная машина в процессе выполнения программы связывается с нужными методами в Java-библиотеке. Для достижения более высокой производительности JVM может вызвать JIT-компилятор, который выборочно компилирует методы в соответствующие инструкции машинного языка той машины, на которой он запущен

Чтобы сохранить переносимость и повысить скорость выполнения, следующим этапом разработки Java стали компиляторы, которые транслировали программу в процессе ее выполнения. Такие **JIT-компиляторы** («Just In Time», то есть «своевременные») обычно исследуют запущенную программу с целью найти в ней наиболее востребованные методы, а затем компилируют их в набор инструкций того компьютера, на котором запущена виртуальная машина. Скомпилированная часть сохраняется для следующего запуска программы, чтобы с каждым запуском она могла выполняться еще быстрее. Этот баланс интерпретации и компиляции со временем эволюционирует, поэтому частый запуск Java-программ немного скрадывает издержки интерпретации.

Поскольку быстродействие компьютеров растет, расширяя тем самым возможности компиляторов, а исследователи изобретают все более совершенные способы компиляции Java-программ на лету, разрыв производительности между Java и C или C++ уменьшается.

Самопроверка

Как вы думаете, какие преимущества интерпретатора над транслятором были важнее всего для разработчиков Java?

1. Простота создания интерпретатора.
2. Более подробная система сообщения об ошибках.
3. Более компактный объектный код.
4. Независимость от машины, на которой запускается программа.

JIT-компилятор
Название, часто присваиваемое компилятору, который работает в процессе выполнения программы, транслируя интерпретируемый кодовый сегмент в машинный код того компьютера, на котором он работает.

2.13. Объединение всего ранее рассмотренного в упражнении по сортировке на языке C

Одна из опасностей демонстрации кода на языке ассемблера небольшими фрагментами заключается в том, что они не дают понять, как выглядит полноценная программа на ассемблере. В этом разделе мы получим MIPS-код из двух процедур, написанных на языке C: одна из них будет менять местами элементы массива, а другая будет их сортировать.

Листинг 2.1. Процедура на языке C, которая меняет местами содержимое двух ячеек памяти. Фрагмент, использующий эту процедуру, приведен в упражнении по сортировке.

```
void swap(int v[], int k)

{
  int temp;
  temp = v[k];
  v[k] = v[k+1];
  v[k+1] = temp;
}
```

Процедура swap

Начнем с кода процедуры обмена, показанной в листинге 2.1. Эта процедура просто меняет местами содержимое двух ячеек памяти. При ручной трансляции с языка C на язык ассемблера мы пройдем следующие основные этапы:

1. Выделение регистров под переменные программы.
2. Создание кода для тела процедуры.
3. Сохранение регистров на время вызова процедуры.

В этом разделе описывается процедура обмена, разбитая на три части, а в конце все части будут объединены.

Выделение регистров для процедуры swap

Как уже упоминалось в разделе 2.8, по действующему в MIPS соглашению о передаче параметров для них используются регистры `$a0`, `$a1`, `$a2` и `$a3`. Поскольку у процедуры `swap` всего два параметра, `v` и `k`, они будут находиться в регистрах `$a0` и `$a1`. Оставшуюся переменную `temp` мы свяжем с регистром `$t0`, поскольку `swap` является конечной процедурой (см. подраздел «Вложенные процедуры»). Такое распределение регистров соответствует объявлениям переменных в первой части процедуры `swap` в листинге 2.1.

Код тела процедуры swap

Оставшиеся строки С-кода процедуры swap имеют следующий вид:

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

Вспомним, что адресация памяти для MIPS ссылается на адреса байтов, и поэтому слова находятся друг от друга на расстоянии четырех байтов. *Последовательные адреса слов отличаются друг от друга на 4, а не на байт. Это обстоятельство упускается из виду довольно часто, что является весьма распространенной ошибкой при программировании на языке ассемблера.* Итак, первым шагом будет назначение адреса элементу v[k] путем умножения k на 4 с помощью сдвига влево на два разряда:

```
sll $t1, $a1,2          # регистр $t1 = k * 4
add $t1, $a0,$t1        # регистр $t1 = v + (k * 4)
                        # регистр $t1 содержит адрес v[k]
```

Теперь мы загрузим v[k], используя $t1, а затем v[k+1], добавляя 4 к $t1:

```
lw $t0, 0($t1)          # регистр $t0 (temp) = v[k]
lw $t2, 4($t1)          # регистр $t2 = v[k + 1]
                        # ссылка на следующий элемент массива v
```

Затем мы сохраним $t0 и $t2 для обмена содержимого адресов:

```
sw $t2, 0($t1)          # v[k] = регистру $t2
sw $t0, 4($t1)          # v[k+1] = регистру $t0 (temp)
```

Мы распределили регистры и написали код для выполнения операций процедуры. Но еще не создан код для сохранения сохраняемых регистров, используемых внутри процедуры swap. Поскольку в данной конечной процедуре сохраняемые регистры не используются, то нам просто нечего сохранять.

Полная версия процедуры swap

Теперь мы готовы к созданию всей процедуры, которая включает метку процедуры и переход, возвращающий управление. Чтобы проще было разобраться, каждый блок процедуры с указанием его предназначения показан в листингах 2.2 и 2.3.

Листинг 2.2. Тело процедуры

```
swap: sll $t1, $a1, 2       # регистр $t1 = k * 4
      add $t1, $a0, $t1     # регистр $t1 = v + (k * 4)
                            # регистр $t1 содержит адрес v[k]
      lw $t0, 0($t1)        # регистр $t0 (temp) = v[k]
      lw $t2, 4($t1)        # регистр $t2 = v[k + 1]
                            # ссылка на следующий элемент массива v
      sw $t2, 0($t1)        # v[k] = регистру $t2
      sw $t0, 4($t1)        # v[k+1] = регистр $t0 (temp)
```

Листинг 2.3. Выход из процедуры

```
      jr $ra                # возвращение к вызывающей процедуре
```

Процедура sort

Чтобы вы по-настоящему смогли оценить скрупулезность программирования на языке ассемблера, мы рассмотрим второй, более длинный пример. На этот раз будет создана процедура, которая вызывает процедуру swap. Создаваемая программа сортирует массив целых чисел, используя пузырьковую сортировку, или сортировку обмена, которая является одной из самых простых и едва ли не самой быстрой из сортировок. В листинге 2.4 показана версия программы на языке C. Мы еще раз представим эту процедуру по частям, завершив ее рассмотрение полноценным вариантом.

Листинг 2.4. Процедура на языке C, сортирующая массив v

```
void sort (int v[], int n)
{
  int i, j;
  for (i = 0; i < n; i += 1) {
    for (j = i - 1; j >= 0 && v[j] > v[j + 1]; j -= 1) {
      swap(v,j);
    }
  }
}
```

Выделение регистров для процедуры sort

Для двух параметров процедуры sort, v и n, выделяются регистры $a0 и $a1, регистр $s0 выделяется для переменной i, а регистр $s1 — для переменной j.

Код тела процедуры sort

Тело процедуры состоит из двух вложенных циклов for и вызова процедуры swap, который включает параметры. Давайте рассмотрим код от периферии к центру.

Первым шагом трансляции станет первый цикл for:

```
for (i = 0; i < n; i += 1) {
```

Вспомним, что инструкция for на языке C состоит из трех частей: инициализации, проверки условия цикла и повторяющегося инкремента значения переменной цикла. Для реализации первой части инструкции for, которая заключается в инициализации переменной i значением 0, достаточно одной ассемблерной инструкции:

```
move $s0, $zero          # i = 0
```

(Вспомним, что move является псевдоинструкцией, предоставляемой ассемблером для облегчения работы с этим языком; см. подраздел «Ассемблер».) Одна инструкция понадобится и для реализации последней части инструкции for, которая заключается в увеличении значения переменной i на единицу:

```
addi $s0, $s0, 1         # i += 1
```

Выход из цикла должен произойти, если выражение i < n не является истиной или, иначе говоря, если i ≥ n. Инструкция «установить, если меньше чем» (set on less than), устанавливает значение регистра $t0 в 1, если $s0 < $a1, и в 0 в противном

случае. Поскольку нам нужно проверить справедливость выражения $s0 ≥ $a1, условный переход будет произведен в том случае, если значение регистра $t0 будет равно нулю. Для всего этого потребуются две инструкции:

```
forltst:slt $t0, $s0, $a1     # регистр $t0 = 0, если $s0 ≥ $a1 (i≥n)
        beq $t0, $zero,exit1  # переход к метке exit1, если $s0 ≥ $a1 (i≥n)
```

В самом конце цикла осуществляется простой переход назад к проверке условия цикла:

```
        j  forltst   # переход к проверке условия внешнего цикла
exit1:
```

Схематически код первого цикла for приобретает следующий вид:

```
        move  $s0, $zero          # i = 0
forltst:slt   $t0, $s0, $a1       # регистр $t0 = 0, если $s0 ≥ $a1 (i≥n)
        beq   $t0, $zero,exit1    # переход на exit1, если $s0 ≥ $a1 (i≥n)

          (тело первого цикла for)

        addi  $s0, $s0, 1         # i += 1
        j  forltst                # переход к проверке условия внешнего цикла
exit1:
```

Готово! (В упражнениях исследуется написание наиболее быстрого кода для подобных циклов.)

Второй цикл for имеет на языке C следующий вид:

```
for (j = i - 1; j >= 0 && v[j] > v[j + 1]; j -= 1) {
```

Инициализационная часть этого цикла также занимает одну инструкцию:

```
addi $s1, $s0, -1      # j = i - 1
```

На уменьшение значения j на единицу в конце цикла также уходит одна инструкция:

```
addi $s1, $s1, -1      # j -= 1
```

Проверка условия цикла состоит из двух частей. Выход из цикла происходит при несоблюдении любого из условий, поэтому первая проверка должна привести к выходу из цикла, если она не будет успешной (j < 0):

```
for2tst: slti  $t0, $s1, 0          # регистр $t0 = 1, если $s1 < 0 (j < 0)
         bne   $t0, $zero, exit2    # переход на exit2, если $s1 < 0 (j < 0)
```

Этот условный переход пропустит вторую проверку условия. Если пропуск не состоится, значит, j ≥ 0.

Вторая проверка вызовет выход, если выражение v[j] > v[j + 1] не будет соответствовать действительности или если будет соблюдено условие v[j] ≤ v[j + 1]. Сначала путем умножения j на 4 будет вычислен адрес (поскольку нам нужен байтовый адрес), а затем он будет добавлен к базовому адресу массива v:

```
sll $t1, $s1, 2         # регистр $t1 = j * 4
add $t2, $a0, $t1       # регистр $t2 = v + (j * 4)
```

Теперь будет загружено значение v[j]:

```
lw $t3, 0($t2)          # регистр $t3 = v[j]
```

Поскольку известно, что второй элемент содержится в непосредственно следующем слове, к адресу в регистре $t2 будет добавлено число 4, чтобы получить v[j + 1]:

```
a $t4, 4($t2)          # регистр $t4 = v[j + 1]
```

Проверка v[j] ⩽ v[j + 1] эквивалентна проверке v[j + 1] ⩾ v[j], поэтому эти две инструкции проверки на выход из цикла приобретут следующий вид:

```
t $t0, $t4, $t3        # регистр $t0 = 0, если $t4 ≥ $t3
eq $t0, $zero, exit2   # переход на exit2, если $t4 ≥ $t3
```

В последней строке цикла находится инструкция перехода назад на проверку условия внутреннего цикла:

```
for2tst                # переход на проверку условия внутреннего цикла
```

Объединив все части, мы получаем следующую основу для второго цикла for:

```
      addi  $s1, $s0, -1       # j = i - 1
r2tst:slti  $t0, $s1, 0        # регистр $t0 = 1, если $s1 < 0 (j < 0)
      bne   $t0, $zero, exit2  # переход на exit2, если $s1 < 0 (j < 0)
      sll   $t1, $s1, 2        # регистр $t1 = j * 4
      add   $t2, $a0, $t1      # регистр $t2 = v + (j * 4)
      lw    $t3, 0($t2)        # регистр $t3 = v[j]
      lw    $t4, 4($t2)        # регистр $t4 = v[j + 1]
      slt   $t0, $t4, $t3      # регистр $t0 = 0, если $t4 ≥ $t3
      beq   $t0, $zero, exit2  # переход на exit2, если $t4 ≥ $t3

        (тело второго цикла for)

      addi  $s1, $s1, -1       # j -= 1
      j     for2tst            # переход к проверке условия внутреннего цикла
xit2
```

Вызов процедуры из sort

Следующим шагом станет создание тела второго цикла for:

```
wap(v,j);
```

Вызов процедуры swap выглядит довольно просто:

```
al swap
```

Передача параметров в sort

Когда нужно передать параметры, возникает проблема, потому что процедуре sort нужны значения, хранящиеся в регистрах $a0 и $a1, а кроме этого, процедуре swap нужно, чтобы ее параметры были помещены в те же самые регистры. Одним из решений может стать копирование параметров для sort в другие регистры где-нибудь выше в коде этой процедуры, что обеспечит доступность регистров $a0 и $a1 при вызове процедуры swap. (Это копирование осуществляется быстрее, чем сохранение в стеке и восстановление из него.) Сначала в самой процедуре регистры $a0 и $a1 копируются в регистры $s2 и $s3:

```
ove   $s2, $a0         # копирование параметра $a0 в $s2
ove   $s3, $a1         # копирование параметра $a1 в $s3
```

А затем из этих двух инструкций процедуре swap передаются параметры:

```
move    $a0, $s2        # первый параметр для swap - это v
move    $a1, $s1        # второй параметр для swap - это j
```

Сохранение регистров в sort

Теперь осталось только создать код сохранения и восстановления регистров. Вполне очевидно, что мы должны сохранить адрес возврата в регистре $ra, поскольку sort является вызываемой процедурой. Процедура sort использует сохраняемые регистры $s0, $s1, $s2 и $s3, и поэтому они также должны быть сохранены. Прологом процедуры sort должен стать следующий код:

```
addi  $sp,$sp,-20       # выделение места в стеке для 5 регистров
sw    $ra,16($sp)       # сохранение в стек регистра $ra
sw    $s3,12($sp)       # сохранение в стек регистра $s3
sw    $s2, 8($sp)       # сохранение в стек регистра $s2
sw    $s1, 4($sp)       # сохранение в стек регистра $s1
sw    $s0, 0($sp)       # сохранение в стек регистра $s0
```

В конце процедуры нужно просто расположить все эти инструкции в перевернутом виде, а затем добавить инструкцию jr для возврата управления.

Процедура sort в своем полном виде

Теперь нужно собрать все вместе в листинге 2.5, не забыв заменить ссылки на регистры $a0 и $a1 в циклах for ссылками на регистры $s2 и $s3. Здесь опять для лучшего восприятия кода каждому блоку дано определение, показывающее, чем именно он занимается в данной процедуре. В этом примере девять строк процедуры sort на языке C превратились в 35 строк на языке ассемблера MIPS.

Уточнение. Один из способов оптимизации, который можно применить в этом примере, заключается во встраивании процедуры. Вместо передачи аргументов в параметрах и вызова кода с помощью инструкции jal компилятор скопирует код из тела процедуры swap в то место кода, где будет находиться вызов этой процедуры. Встраивание позволит избавиться в этом примере сразу от четырех инструкций. Недостаток оптимизации встраивания заключается в том, что скомпилированный код будет больше по объему, если встраиваемая процедура вызывается из нескольких мест. Подобное расширение кода может привести к снижению производительности, если оно увеличит количество случаев отсутствия нужных данных в кэше; см. главу 5.

Листинг 2.5. Версия процедуры sort, показанной в листинге 2.4, на языке ассемблера MIPS

Сохранение регистров

```
sort: addi $sp,$sp, -20   # выделение места в стеке для 5 регистров
      sw   $ra, 16($sp)   # сохранение в стеке регистра $ra
      sw   $s3, 12($sp)   # сохранение в стеке регистра $s3
      sw   $s2, 8($sp)    # сохранение в стеке регистра $s2
      sw   $s1, 4($sp)    # сохранение в стеке регистра $s1
      sw   $s0, 0($sp)    # сохранение в стеке регистра $s0
```

Тело процедуры	
Перемещение параметров	`move $s2, $a0  # копирование параметра $a0 в $s2 (сохр. $a0)` `move $s3, $a1  # копирование параметра $a1 в $s3 (сохр. $a1)`

Внешний цикл	`move $s0, $zero  # i = 0`
	`for1tst: slt $t0, $s0, $s3  # $t0 = 0, если $s0 ≥ $s3 (i ≥ n)`
	`beq $t0, $zero, exit1 # на exit1, если $s0 ≥ $s3 (i ≥ n)`

Внутренний цикл	`addi $s1, $s0, -1  # j = i - 1`
	`for2tst: slti $t0, $s1, 0  # $t0 = 1, если $s1 < 0 (j < 0)`
	`bne $t0, $zero, exit2 # на exit2, если $s1 < 0 (j < 0)`
	`sll $t1, $s1, 2  # $t1 = j * 4`
	`add $t2, $s2, $t1  # $t2 = v + (j * 4)`
	`lw $t3, 0($t2)  # $t3 = v[j]`
	`lw $t4, 4($t2)  # $t4 = v[j + 1]`
	`slt $t0, $t4, $t3  # $t0 = 0, если $t4 ≥ $t3`
	`beq $t0, $zero, exit2 # на exit2, если $t4 ≥ $t3`

Передача параметров и вызов	`move $a0, $s2  # 1-й параметр для swap – v (прежний $a0)`
	`move $a1, $s1  # 2-й параметр для swap – j`
	`jal swap  # код процедуры swap – в листингах 2.2 - 2.3`

Внутренний цикл	`addi $s1, $s1, -1  # j -= 1` `j for2tst  # переход к проверке условия внутреннего цикла`
Внешний цикл	`exit2: addi $s0, $s0, 1  # i += 1` `j for1tst  # переход к проверке условия внешнего цикла`
Восстановление регистров	

	`exit1: lw $s0, 0($sp)  # восстановление из стека $s0`
	`lw $s1, 4($sp)  # восстановление из стека $s1`
	`lw $s2, 8($sp)  # восстановление из стека $s2`
	`lw $s3,12($sp)  # восстановление из стека $s3`
	`lw $ra,16($sp)  # восстановление из стека $ra`
	`addi $sp,$sp, 20  # восстановление указателя стека`

Возвращение из процедуры	
	`jr $ra  # возвращение к вызывавшей процедуре`

Представление о производительности программ

В табл. 2.9 показано влияние оптимизации в процессе компиляции на производительность программы сортировки, время компиляции, количество тактовых циклов, количество инструкций и показатель CPI. Обратите внимание на то, что наилучший показатель CPI принадлежит неоптимизированному коду, а оптимизация O1 выдает наименьшее количество инструкций, но оптимизация O3 выдает самый быстрый код, напоминая о том, что единственной точной единицей измерения производительности является время.

В табл. 2.10 сравнивается влияние выбора языка программирования, компиляции вместо интерпретации и алгоритма на производительность сортировки. В четвер-

том столбце показано, что неоптимизированная С-программа работает для пузырьковой сортировки в 8,3 раза быстрее интерпретируемого Java-кода. Использование JIT-компилятора позволяет Java-коду работать в 2,1 раза быстрее неоптимизированной программы на С, и он в 1,13 раза отличается по скорости работы от С-кода, прошедшего наивысшую оптимизацию. Соотношения для сортировки Quicksort в пятом столбце не дают картины, близкой к этой, возможно потому, что за более короткое время выполнения труднее получить выгоду от компиляции в процессе выполнения программы. В последнем столбце демонстрируется влияние более удачного алгоритма, предлагающего увеличение производительности на три порядка при сортировке 100 000 записей. Даже при сравнении показателей работы интерпретируемого Java-кода в пятом столбце с показателями работы С-кода, прошедшего наивысшую оптимизацию, показанными в столбце 4, сортировка Quicksort демонстрирует в 50 раз более высокую эффективность, чем пузырьковая сортировка (0,05 × 2468, или в 123 раза быстрее, чем неоптимизированный С-код в сравнении с ускорением в 2,41 раза).

Уточнение. MIPS-компиляторы всегда оставляют место в стеке для аргументов на тот случай, когда они должны быть сохранены, поэтому в действительности они всегда уменьшают значение $sp на 16, чтобы оставить место для всех четырех регистров аргументов (16 байт). Одна из причин состоит в том, что язык С предоставляет настройку vararg, которая позволяет по указателю выбирать процедуре, скажем, третий аргумент. Когда компилятор встречает такую редкую настройку, как vararg, он копирует четыре регистра аргументов в стек в четыре зарезервированных места.

Таблица 2.9. Сравнение производительности, количества инструкций и показателя CPI при использовании оптимизации в процессе компилирования процедуры пузырьковой сортировки. Программы сортируют 100 000 слов в массиве, заполненном случайными значениями. Эти программы были запущены на компьютере Pentium 4 с тактовой частотой 3,06 ГГц и системной шиной, работающей на частоте 533 МГц, имеющем память PC2100 DDR SDRAM объемом 2 Гбайт. Компьютер работал под управлением операционной системы Linux версии 2.4.20

gcc-оптимизация	Относительная производительность	Количество тактовых циклов, млн	Количество инструкций, млн	CPI
Нет	1,00	158 615	114 938	1,38
O1 (средняя)	2,37	66 990	37 470	1,79
O2 (полная)	2,38	66 521	39 993	1,66
O3 (объединение процедур)	2,41	65 747	44 993	1,46

Таблица 2.10. Производительность двух алгоритмов сортировки в C и Java при использовании интерпретации и оптимизирующих компиляторов относительно производительности неоптимизированной C-версии. В последнем столбце показано преимущество в производительности сортировки Quicksort над пузырьковой сортировкой для каждого языка и выбора метода выполнения. Эти программы запускались на той же системе, что и программы, чья работа была показана в табл. 2.8. В качестве JVM использовалась виртуальная машина Sun версии 1.3.1 и JIT-компилятор Sun Hotspot версии 1.3.1

Язык	Метод выполнения	Оптимизация	Относительная производительность пузырьковой сортировки	Относительная производительность сортировки Quicksort	Коэффициент ускорения Quicksort по сравнению с пузырьковой сортировкой
C	Компилятор	Нет	1,00	1,00	2468
	Компилятор	O1	2,37	1,50	1562
	Компилятор	O2	2,38	1,50	1555
	Компилятор	O3	2,41	1,91	1955
Java	Интерпретатор	–	0,12	0,05	1050
	JIT-компилятор	–	2,13	0,29	338

2.14. Сравнение массивов и указателей

Начинающим программистам, работающим на языке C, довольно трудно понять, что такое указатели. Разобраться с ними можно путем сравнения ассемблерного кода, использующего массивы и индексы массивов, с ассемблерным кодом, использующим указатели. В этом разделе показаны версии на языке C и на языке ассемблера MIPS двух процедур для очистки последовательности слов в памяти: в одной из них используются индексы массива, а в другой указатели. В листинге 2.6 показаны две процедуры на языке C.

Цель этого раздела — показать, как указатели отображаются в MIPS-инструкциях, и при этом отойти от устаревшего стиля программирования. В конце раздела будет показано влияние современной оптимизации в процессе компиляции этих двух процедур.

Листинг 2.6. Две С-процедуры для установки всех значений массива в нуль
Процедура clear1 использует индексы, а процедура clear2 использует указатели. Вторая процедура для тех, кто не знаком с языком С, нуждается в некотором пояснении. Адрес переменной обозначается символом &, а на объект указывает указатель, обозначаемый символом *. В объявлении указывается, что массив array и указатель p являются указателями на целые числа. Первая часть цикла for в процедуре clear2 присваивает адрес первого элемента массива указателю p. Вторая часть цикла for проверяет, не указывает ли указатель за пределы последнего элемента массива. Увеличение значения указателя на единицу в последней части цикла for означает перемещение указателя к следующему объекту в последовательности согласно объявленному размеру. Поскольку p является указателем на целые числа, компилятор генерирует MIPS-инструкции для увеличения p на четыре, то есть на количество байтов в целом числе MIPS. Присваивание в цикле помещает нулевое значение в объект, на который указывает p

```
clear1(int array[], int size)
{
  int i;
  for (i = 0; i < size; i += 1)
    array[i] = 0;
}
clear2(int *array, int size)
{
  int *p;
  for (p = &array[0]; p < &array[size]; p = p + 1)
    *p = 0;
}
```

Версия процедуры Clear, использующая массив

Начнем с версии clear1, использующей массив, уделяя при этом основное внимание телу цикла и игнорируя компоновочный код процедуры. Предположим, что два параметра, array (массив) и size (размер), находятся в регистрах $a0 и $a1, а переменная i находится в регистре $t0.

Инициализация i, первая часть цикла for, сложности не представляет:

```
move $t0,$zero          # i = 0 (регистр $t0 = 0)
```

Для присвоения элементу array[i] нулевого значения нужно сначала получить адрес этого элемента. Начнем с умножения i на 4, чтобы получить байтовый адрес:

```
loop1: sll $t1,$t0,2          # $t1 = i * 4
```

Поскольку стартовый адрес массива находится в регистре, мы должны, используя инструкцию add, добавить его к индексу, чтобы получить адрес array[i]:

```
add $t2,$a0,$t1         # $t2 = адрес array[i]
```

И наконец, нам нужно сохранить нуль по этому адресу:

```
sw $zero, 0($t2)        # array[i] = 0
```

Этой инструкцией заканчивается тело цикла, поэтому следующим шагом станет приращение значения i:

```
addi $t0,$t0,1          # i = i + 1
```

Проверка условия цикла устанавливает, не является ли i меньше, чем значение параметра size (размер):

```
slt $t3,$t0,$a1      # $t3 = (i < размер)
bne $t3,$zero,loop1  # если (i < размер), переход на loop1
```

Мы рассмотрели все части процедуры. Код MIPS для очистки массива с помощью индексов имеет следующий вид:

```
      move $t0,$zero       # i = 0
loop1: sll  $t1,$t0,2      # $t1 = i * 4
      add  $t2,$a0,$t1     # $t2 = адрес array[i]
      sw   $zero, 0($t2)   # array[i] = 0
      addi $t0,$t0,1       # i = i + 1
      slt  $t3,$t0,$a1     # $t3 = (i < размер)
      bne  $t3,$zero,loop1 # если (i < размер), переход на loop1
```

(Этот код работает, пока size, то есть размер, больше нуля; ANSI C требует проверки размера до запуска цикла, но здесь это правило будет проигнорировано.)

Версия процедуры Clear, использующая указатель

Вторая процедура, которая использует указатели, размещает два параметра, array и size, в регистрах $a0 и $a1, а переменную p помещает в регистр $t0. Код второй процедуры начинается с присваивания указателю p адреса первого элемента массива:

```
move $t0,$a0         # p = адрес array[0]
```

Затем следует код тела цикла for, сохраняющий нулевое значение в том месте, на которое указывает p:

```
loop2  sw   $zero,0($t0)     # Память[p] = 0
```

Эта инструкция реализует тело цикла, поэтому следующий код является приращением итерации, изменяющим значение указателя p, чтобы он указывал на следующее слово:

```
addi $t0,$t0,4       # p = p + 4
```

Увеличение значения указателя на единицу означает в C перемещение указателя на следующий последовательный объект. Поскольку p является указателем на целые числа, каждое из которых использует 4 байта, компилятор увеличивает p на 4.

Затем следует проверка условия цикла. Сначала вычисляется адрес последнего элемента массива. Для получения его байтового адреса нужно начать с умножения размера (size) на 4:

```
sll  $t1,$a1,2       # $t1 = size * 4
```

а затем добавить полученный результат к начальному адресу массива, чтобы получить адрес первого слова, идущего после массива:

```
add  $t2,$a0,$t1     # $t2 = адрес array[size]
```

Проверка условия цикла просто следит за тем, чтобы значение указателя p было меньше адреса последнего элемента массива:

```
slt  $t3,$t0,$t2     # $t3 = (p<&array[size])
bne  $t3,$zero,loop2 # если (p<&array[size]), перейти на loop2
```

Если собрать все части вместе, получится версия процедуры обнуления всех значений массива, использующая указатель:

```
        move $t0,$a0          # p = адрес array[0]
loop2:  sw   $zero,0($t0)     # Память[p] = 0
        addi $t0,$t0,4        # p = p + 4
        sll  $t1,$a1,2        # $t1 = size * 4
        add  $t2,$a0,$t1      # $t2 = адрес array[size]
        slt  $t3,$t0,$t2      # $t3 = (p<&array[size])
        bne  $t3,$zero,loop2  # если (p<&array[size]), перейти на loop2
```

Как и в первом примере, предполагается, что размер массива больше нуля.

Следует учесть, что эта программа вычисляет адрес конца массива при каждом проходе цикла, даже если этот адрес не изменился. В более быстродействующей версии кода это вычисление вынесено за пределы цикла:

```
        move $t0,$a0          # p = адрес array[0]
        sll  $t1,$a1,2        # $t1 = size * 4
        add  $t2,$a0,$t1      # $t2 = адрес array[size]
loop2:  sw   $zero,0($t0)     # Память[p] = 0
        addi $t0,$t0,4        # p = p + 4
        slt  $t3,$t0,$t2      # $t3 = (p<&array[size])
        bne  $t3,$zero,loop2  # если (p<&array[size]), перейти на loop2
```

Сравнение двух версий процедуры Clear

Сравнение поставленных рядом двух кодовых последовательностей показывает разницу между использованием индексов массива и указателей (изменения, внесенные в версию с указателями, выделены жирным шрифтом):

move $t0,$zero	# i = 0
loop1: sll $t1,$t0,2	# $t1 = i * 4
add $t2,$a0,$t1	# $t2 = &array[i]
sw $zero, 0($t2)	# array[i] = 0
addi $t0,$t0,1	# i = i + 1
slt $t3,$t0,$a1	# $t3 = (i < size)
bne $t3,$zero,loop1	# если (), перейти на loop1
move $t0 $a0	**# p = & array[0]**
sll $t1,$a1,2	# $t1 = size * 4
add $t2,$a0,$t1	# $t2 = &array[size]
loop2 sw $zero,0($t0)	**# Память[p] = 0**
addi $t0,$t0,4	**# p = p + 4**
slt $t3,$t0,$t2	# $t3=(p<&array[size])
bne $t3,$zero,loop2	# если (), перейти на loop2

В версии, показанной слева, «умножение» и сложение должны быть внутри цикла, поскольку значение переменной i увеличивается на единицу, и под новое значение индекса нужно заново вычислять каждый адрес. Версия с указателем ячейки памяти, показанная справа, проводит непосредственное приращение указа-

теля p. Вышеупомянутые операции в этой версии вынесены за пределы цикла, что позволило уменьшить количество инструкций, выполняемых при каждом проходе цикла, с шести до четырех. Эта ручная оптимизация соответствует оптимизации во время компиляции, направленной на упрощение выражения (сдвиг вместо умножения) и исключение индукционных переменных (исключение вычисления адреса массива внутри циклов).

Уточнение. Как уже ранее упоминалось, компилятор C добавит проверку, позволяющую убедиться, что значение переменной `size` больше нуля. Одним из способов может быть добавление непосредственно перед первой инструкцией цикла перехода к инструкции `slt`.

Представление о производительности программ

Для получения при работе с языком C более высокой производительности, чем при работе с массивами, нужно освоить применение указателей: «Используйте указатели даже в том случае, если не можете понять, как работает код». Современные оптимизирующие компиляторы могут создавать сопоставимый по качеству код и для версии с массивом. Сегодня многие программисты предпочитают всю тяжелую работу перекладывать на компиляторы.

2.15. Реальное оборудование: инструкции ARM

ARM является наиболее популярной архитектурой набора инструкций для встроенных устройств и используется на более чем трех миллиардах производимых за каждый год устройств. Изначально предназначенная для системы Acorn RISC Machine, название которой позднее изменилось на Advanced RISC Machine, архитектура ARM появилась в том же году, что и архитектура MIPS, и следовала схожим принципам. Все совпадения перечислены в табл. 2.11. Принципиальное отличие заключалось в том, что у MIPS было больше регистров, а у ARM — больше режимов адресации.

Как показано в табл. 2.12, для MIPS и для ARM использовались похожие основные наборы арифметическо-логических инструкций и инструкций переноса данных.

Режимы адресации

В табл. 2.13 показаны режимы адресации данных, поддерживаемые в ARM. В отличие от MIPS ARM не выделяет регистр для содержания нулевого значения. В MIPS имеется всего три простых режима адресации данных (см. рис. 2.12), а вот у ARM имеется девять режимов, которые включают довольно сложные вычисления. Например, в ARM имеется режим адресации, который для формирования адреса

может сдвигать значение одного регистра на любое количество позиций, добавлять его к значениям других регистров, а затем обновлять значение одного из регистров, присваивая ему значение полученного нового адреса.

Таблица 2.11. Сходство наборов инструкций ARM и MIPS

	ARM	MIPS
Год представления	1985	1985
Размер инструкции (в разрядах)	32	32
Адресное пространство (размер, модель)	32 разряда, плоское	32 разряда, плоское
Выравнивание данных	выровненные	выровненные
Модели адресации данных	9	3
Целочисленные регистры (количество, модель, размер)	15 регистров общего назначения по 32 разряда	31 регистр общего назначения по 32 разряда
Ввод-вывод	С отображением на память	С отображением на память

Таблица 2.12. ARM-инструкции типа регистр-регистр и инструкции переноса данных, эквивалентные основным инструкциям MIPS. Прочерки означают, что операция в этой архитектуре недоступна или не синтезирована в виде нескольких инструкций. Если имеется сразу несколько инструкций, эквивалентных основным инструкциям MIPS, они разделены запятыми. Сдвиги в ARM включены в виде части каждой инструкции обработки данных, поэтому сдвиги с верхним индексом 1 являются всего лишь разновидностью инструкции move, например lsr[1]. Заметьте, что в ARM нет инструкции деления

	Название инструкции	ARM	MIPS
Регистр-регистр	Сложение	add	addu, addiu
	Сложение (с перехватом при переполнении)	adds; swivs	add
	Вычитание	sub	subu
	Вычитание (с перехватом при переполнении)	subs; swivs	sub
	Умножение	mul	mult, multu
	Деление	–	div, divu
	И	and	and
	ИЛИ	orr	or
	Исключающее ИЛИ	eor	xor
	Загрузка верхней части регистра	–	lui
	Логический сдвиг влево	lsl[1]	sllv, sll
	Логический сдвиг вправо	lsr[1]	srlv, slr
	Арифметический сдвиг вправо	asr[1]	srav, sra
	Сравнение	cmp, cmn, tst, teq	slt/i, slt/iu

	Название инструкции	ARM	MIPS
Перенос данных	Загрузка байта со знаком	ldrsb	lb
	Загрузка байта без знака	ldrb	lbu
	Загрузка полуслова со знаком	ldrsh	lh
	Загрузка полуслова без знака	ldrh	lhu
	Загрузка слова	ldr	sw
	Сохранение байта	strb	sb
	Сохранение полуслова	strh	sh
	Сохранение слова	str	sw
	Чтение, запись специальных регистров	mrs, msr	move
	Атомарный обмен	swp, swpb	ll;sc

Таблица 2.13. Сводка режимов адресации данных. В ARM имеются отдельные режимы косвенной регистровой адресации и адресации «регистр + смещение», простое помещения нуля в смещение во втором режиме не используется. Чтобы получить более широкий диапазон адресации, ARM сдвигает смещение на один или два разряда влево, если размер данных составляет полуслово или слово

Режим адресации	ARM v.4	MIPS
Регистровый операнд	X	X
Непосредственный операнд	X	X
Регистр + смещение (сдвиговое или по базе)	X	X
Регистр + регистр (индексированный)	X	—
Регистр + масштабируемый регистр (масштабируемый)	X	—
Регистр + смещение и обновление регистра	X	—
Регистр + регистр и обновление регистра	X	—
Автоинкрементный, автодекрементный	X	—
С указанием данных относительно счетчика команд	X	—

Сравнение и условный переход

В MIPS для вычисления условий переходов используется содержимое регистров. В ARM используются традиционные четыре разряда кода условия, сохраняемые в слове состояния программы: *отрицательный*, *нулевой*, *переноса* и *переполнения* — *negative*, *zero*, *carry* и *overflow*. Они могут устанавливаться при выполнении любой арифметической или логической инструкции; в отличие от более ранних архитектур эта установка для каждой инструкции является выборочной. Явный выбор создает меньше проблем при конвейерной реализации. В ARM используются условные переходы для проверки кодов условий, чтобы определить все знаковые и беззнаковые соответствия.

Инструкция **сравнения** — CMP *вычитает* один операнд из другого, а по разнице устанавливает коды условия. Инструкция **отрицательного сравнения** — **compare negative** (CMN) — *прибавляет* один операнд к другому, а по сумме устанавливает коды условий. Инструкция TST выполняет логическое И двух операндов для установки всех кодов условий, кроме переполнения — overflow, а инструкция TEQ использует исключающее ИЛИ для установки значений первых трех кодов условий.

Одно из необычных свойств ARM заключается в том, что каждая инструкция имеет настройку на условное выполнение в зависимости от кодов условий. Каждая инструкция начинается с четырехразрядного поля, определяющего, будет ли она работать как **холостая инструкция** — **no operation instruction (nop)** — или как настоящая инструкция, в зависимости от кодов условий. Следовательно, по сути условные переходы рассматриваются как условное выполнение инструкции безусловного перехода. Условное выполнение позволяет избежать перехода путем обхода его инструкции. На простое условное выполнение одной инструкции требуется меньше времени и пространства для кода.

На рис. 2.16 показаны форматы инструкций для ARM и MIPS. Принципиальная разница заключается в четырехразрядном поле условного выполнения в каждой инструкции и меньшем по размеру поле регистра, потому что в ARM используется в два раза меньше регистров.

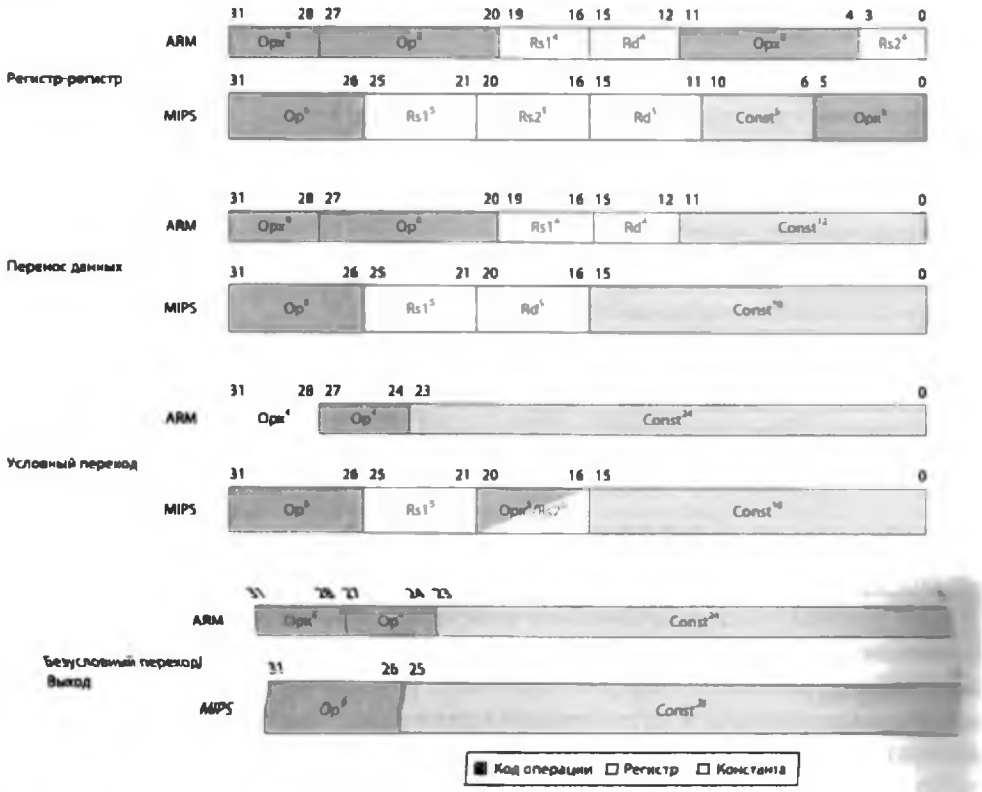


Рис. 2.16. Форматы инструкций ARM и MIPS. Разница заключается в том, что в одной текстуре 16, а в другой 32 регистра

Уникальные характеристики ARM

В табл. 2.14 показаны несколько арифметическо-логических инструкций, отсутствующих в MIPS. За неимением специально выделенного регистра, содержащего нуль, в этой архитектуре есть отдельные коды для выполнения тех операций, которые в MIPS реализуются с помощью регистра $zero. Кроме того, в ARM имеется поддержка для арифметики, оперирующей несколькими словами.

Поле непосредственного значения ARM, состоящее из 12 разрядов, имеет новую интерпретацию. Восемь самых младших разрядов расширяются нулями до 32-разрядного значения, затем прокручиваются вправо на количество разрядов, указанное в первых четырех разрядах поля и умноженное на два. Одно из преимуществ заключается в том, что такая схема может представить все степени двойки в 32-разрядном слове. Было бы интересно исследовать вопрос, позволяет ли такое разделение охватить больше непосредственных значений, чем простое 12-разрядное поле.

Операнд, подвергающийся сдвигу, не ограничивается непосредственными значениями. У второго регистра всех арифметических и логических операций по обработке данных есть возможность подвергнуться сдвигу еще до того, как он будет использован в операции. Вариантами сдвига являются логический сдвиг влево, логический сдвиг вправо, арифметический сдвиг вправо и вращение вправо.

Таблица 2.14. Арифметические и логические инструкции ARM, отсутствующие в MIPS

Название	Определение	ARM v.4	MIPS
Загрузка непосредственного значения	Rd = Imm	mov	addi, $0,
НЕ	Rd = ~(Rs1)	mvn	nor, $0,
Перемещение	Rd = Rs1	mov	or, $0,
Вращение вправо	Rd = Rs i >> i $Rd_{0 \ldots i-1} = Rs_{i \ldots 31}$	ror	
И НЕ	Rd = Rs1 & ~(Rs2)	bic	
Обратное вычитание	Rd = Rs2 – Rs1	rsb, rsc	
Поддержка сложения многословного целого числа	CarryOut, Rd = Rd + Rs1 + Old-CarryOut	adcs	—
Поддержка вычитания многословного целого числа	CarryOut, Rd = Rd – Rs1 + Old-CarryOut	sbcs	—

В ARM также имеются инструкции для сохранения групп регистров, называемые *блочными загрузками и сохранениями*. Под управлением 16-разрядной маски внутри инструкций любой из 16 регистров может быть загружен из памяти или сохранен в ней с помощью одной инструкции. Эти инструкции могут сохранять и восстанавливать регистры при входе в процедуру и при возвращении из нее. Эти инструкции могут также использоваться для блочного копирования памяти, и сегодня блочное копирование приобретает все больший вес при их использовании.

2.16. Реальное оборудование: инструкции x86

Красота всегда в глазах очарованного
Маргарет Вульф Хангерфорд

Иногда разработчики наборов инструкций предоставляют возможность реализации более мощных операций по сравнению с теми, которые имеются в ARM и MIPS. Главная цель заключается в сокращении количества инструкций, выполняемых программой. Опасность такого сокращения может проявиться в цене, которую приходится платить за такое упрощение, и в увеличении времени, затрачиваемого на выполнение программы, поскольку инструкции работают медленнее. Это замедление может быть результатом снижения тактовой частоты или увеличения количества тактовых циклов по сравнению с менее сложной последовательностью инструкций.

Таким образом, усложнение операций чревато опасными последствиями. Чтобы избежать подобных проблем, разработчики пошли по пути упрощения инструкций. Все недостатки усложнения показаны в разделе 2.17.

Развитие семейства Intel x86

В 1985 году ARM и MIPS были замыслами отдельных небольших групп; части этих архитектур неплохо соответствовали друг другу, а описание всей архитектуры не занимало много места. Чего нельзя сказать про архитектуру x86, которая представляет собой продукт работы нескольких независимых групп, которые совершенствовали ее в течение 30 лет, добавляя новые свойства к исходному набору инструкций. Архитектура x86 прошла следующие важные этапы развития:

- **1978**: Была представлена архитектура Intel 8086, являвшаяся расширением успешной в то время архитектуры восьмиразрядного микропроцессора Intel 8080, совместимым на уровне языка ассемблера. Микропроцессор 8086 имел 16-разрядную архитектуру, и все его внутренние регистры имели величину в 16 разрядов. В отличие от MIPS эти регистры использовались в специальных целях, и поэтому архитектура 8086 не считалась архитектурой, имеющей **регистры общего назначения (general-purpose register, GPR)**.
- **1980**: Был представлен сопроцессор Intel 8087 для работы с числами с плавающей точкой. Архитектура 8086 была расширена примерно 60 инструкциями для работы с такими числами. Вместо использования регистров это расширение полагалось на работу со стеком (см. раздел 3.7).
- **1982:** Микропроцессор 80286 расширил архитектуру 8086, увеличив адресное пространство до 24 разрядов за счет создания более развитого управления памятью и модели работы в защищенном режиме (см. главу 5), а также добавления новых инструкций,

Регистр общего назначения (GPR)
Регистр, который может использоваться для адресов или для данных фактически с любой инструкцией.

совершенствующих их общий набор и позволяющих работать в защищенном режиме.

1985: Микропроцессор 80386 расширил архитектуру 80286 до 32 разрядов. В дополнение к 32-разрядной архитектуре с 32-разрядными регистрами и 32-разрядным адресным пространством в архитектуру 80386 были добавлены новые режимы адресации и дополнительные операции. Добавленные инструкции приблизили микропроцессор 80386 к машинам с регистрами общего назначения. В микропроцессоре 80386 к сегментной адресации была также добавлена поддержка страничной адресации (см. главу 5). Как и микропроцессор 80286, его потомок 80386 имел режим выполнения программ, разработанных для процессора 8086, не требовавший их изменения.

1989–1995: Последующие микропроцессоры: 80486 в 1989 году, Pentium в 1992 году и Pentium Pro в 1995 году — были нацелены на достижение более высокой производительности и имели в обозримом пользователем наборе всего четыре новые инструкции: три для помощи в многопроцессорной обработке данных (см. главу 7) и одну инструкцию условного перемещения данных.

1997: После выпуска Pentium и Pentium Pro Intel объявила, что собирается расширить архитектуру Pentium и Pentium Pro инструкциями MMX (Multi Media Extensions — мультимедийные расширения). Этот новый набор из 57 инструкций использует для ускорения работы мультимедийных и коммуникационных приложений стек, работающий с числами с плавающей точкой. Инструкции MMX обычно работают одновременно с несколькими короткими элементами данных согласно принципу архитектуры «одна инструкция, применяемая к нескольким элементам данных» (single instruction, multiple data — SIMD) (см. главу 7). В Pentium II новые инструкции представлены не были.

1999: Компания Intel в качестве составляющих микропроцессора Pentium III добавила еще 70 инструкций, получивших название SSE (Streaming SIMD Extensions — потоковые SIMD-расширения). Главным обновлением было добавление восьми отдельных регистров с удвоением их ширины до 128 разрядов и типа данных с плавающей точкой одинарной точности. Следовательно, параллельно могли быть выполнены четыре 32-разрядные операции над числами с плавающей точкой. Для увеличения производительности памяти набор SSE стал включать инструкции предварительной выборки кэша и инструкции потокового сохранения, которые обходят кэш и ведут запись непосредственно в память.

2001: Компания Intel добавила еще 144 инструкции, которые на этот раз получили название SSE2. Появился новый тип данных для арифметики двойной точности, позволяющий параллельно выполнять пары 64-разрядных операций с числами с плавающей точкой. Почти все из этих 144 инструкций являлись версиями существовавших MMX- и SSE-инструкций, которые параллельно обрабатывают 64-разрядные данные. Это изменение позволило не только использовать больше мультимедийных операций, но и дало компилятору другой

адресат данных для операций с числами с плавающей точкой вместо уникальной стековой архитектуры. Компиляторы получили выбор из восьми SSE-регистров для чисел с плавающей точкой, подобных тем четырем, которые имелись в других компьютерах. Эти изменения увеличили производительность при работе с числами с плавающей точкой в Pentium 4, первом микропроцессоре, включавшем инструкции SSE2.

- **2003:** На сей раз архитектура x86 была расширена компанией, не имеющей отношения к Intel. Компания AMD представила набор архитектурных расширений, предназначенный для увеличения адресного пространства с 32- до 64-разрядного. Подобно переходу с 16- к 32-разрядному адресуемому пространству в 1985 году, который был связан с появлением микропроцессора 80386, на AMD64 все регистры были расширены до 64 разрядов. Число регистров было также расширено до 16, и число 128-разрядных SSE-регистров было расширено до 16. Главное изменение архитектуры набора команд произошло благодаря добавлению нового, так называемого *длинного режима* (*long mode*), который переопределил расширение всех инструкций x86 для работы с 64-разрядными адресами и данными. Чтобы иметь возможность обращения к большему количеству регистров, к инструкциям был добавлен новый префикс. В зависимости от способа вычислений длинный режим также добавлял от 4 до 10 новых инструкций и исключал применение 27 старых инструкций. Еще одним расширением стала адресация данных относительно счетчика команд. В AMD64 по-прежнему сохранялся режим, идентичный имевшемуся в x86 (*режим старых команд — legacy mode*), плюс режим, который ограничивал пользовательские программы набором x86, но разрешал операционной системе использовать набор AMD64 (*режим совместимости — compatibility mode*). Эти режимы делали более привлекательным переход к 64-разрядной адресации по сравнению с применением архитектуры HP/Intel IA-64.
- **2004:** Intel сдается и перенимает технологию AMD64, дав ей новое название: *технология расширенной памяти 64 — Extended Memory 64 Technology* (*EM64T*). Основное отличие заключалось в том, что Intel добавила 128-разрядную инструкцию атомарного сравнения и обмена, которой, наверное, не хватало в AMD64. В то же самое время Intel представила новое поколение мультимедиа-расширений. В SSE3 было добавлено 13 инструкций для поддержки комплексной арифметики, графических операций над массивами структур, кодирования видео, преобразований чисел с плавающей точкой и синхронизации потоков (см. раздел 2.11). AMD предложил SSE3 в своих следующих чипах и, конечно же, добавил отсутствующую атомарную инструкцию обмена к своей технологии AMD64, чтобы обеспечить двоичную совместимость с изделиями Intel.
- **2006:** Intel представила 54 новые инструкции в рамках расширения набора инструкций SSE4. Эти расширения выполняли такие новые операции, как получение суммы основных отличий, получение скалярных произведений для массивов структур, знаковое или нулевое расширение данных с малым числом

разрядов до большего числа разрядов, POPCNT (population count)[1] и т. д. Также была добавлена поддержка виртуальных машин (см. главу 5).

- 2007: AMD представила 170 инструкций в рамках SSE5, включая 46 инструкций основного набора, с добавлением таких же трехоперандных инструкций, как в MIPS.
- 2008: Intel представила *улучшенное векторное расширение — Advanced Vector Extension*, которое расширяло SSE-регистр со 128 до 256 разрядов, из-за чего были переопределены около 250 инструкций и добавлены 128 новых инструкций.

Эта история иллюстрирует влияние «золотых наручников» совместимости в семействе x86, поскольку существующее программное обеспечение на каждом этапе совершенствования играло слишком важную роль, не позволяя ставить его работоспособность под угрозу из-за внесения существенных архитектурных изменений. Если проследить весь период существования семейства x86, то в среднем архитектура расширялась с темпом в одну инструкцию в месяц!

И неважно, что семейство x86 не было востребовано в высокопрофессиональной сфере, следует иметь в виду, что в настольных компьютерах представителей архитектуры этого семейства намного больше, чем представителей любой другой архитектуры, и их количество увеличивается с каждым годом более чем на 250 миллионов экземпляров. Однако столь извилистая история развития этой архитектуры привела к тому, что ее трудно объяснить и невозможно полюбить.

Подготовьтесь к восприятию новой информации! *Не пытайтесь* читать этот раздел с прицелом на то, что вам придется создавать программы для x86; наша задача состоит всего лишь в знакомстве с сильными и слабыми сторонами наиболее распространенной архитектуры для настольных компьютеров.

Вместо того чтобы показывать полный набор 16- и 32-разрядных инструкций, в этом разделе мы сконцентрируемся на его 32-разрядном поднаборе, впервые появившемся на микропроцессоре 80386 как часть той архитектуры, которая используется сегодня. Сначала будут рассмотрены регистры и режимы адресации, затем мы перейдем к целочисленным операциям и завершим обзор изучением порядка кодирования инструкций.

Регистры и режимы адресации данных x86

Регистры микропроцессора 80386 отображают эволюцию набора инструкций (рис. 2.17). В микропроцессоре 80386 все 16-разрядные регистры были расширены до 32 разрядов (за исключением сегментных регистров) и к их именам был добавлен префикс E, показывающий, что это 32-разрядная версия. Чаще всего мы будем обращаться к ним как к регистрам общего назначения — GPR (general-purpose

[1] Операция «population count» — это собственное имя операции вычисления веса Хэмминга для битовой строки. Фактически — подсчет суммы цифр в ней (или, что то же самое, — числа единичных символов). — *Примеч. ред.*

registers). Микропроцессор 80386 содержит всего восемь GPR. Это означает, что MIPS-программы могут использовать в 4 раза, а ARM-программы в 2 раза больше регистров.

В табл. 2.15 показаны арифметические, логические инструкции и инструкции переноса данных. Эти инструкции являются двухоперандными. У них есть две важные характерные особенности. У арифметических и логических инструкций x86 один операнд должен действовать и как источник, и как приемник, в то время как ARM и MIPS предлагают для источника и приемника отдельные регистры. Это ограничение увеличивает нагрузку на сокращенное количество регистров, поскольку один из регистров-источников должен быть изменен. Вторая характерная особенность заключается в том, что один из операндов должен быть в памяти. Таким образом, в отличие от ARM и MIPS практически каждая инструкция должна иметь один операнд в памяти.

Режимы адресации данных в памяти, подробно рассматриваемые ниже, предлагают два размера адресов внутри инструкций. Это так называемое смещение может быть 8- или 32-разрядным.

Хотя для операндов в памяти может использоваться любой режим адресации, существуют ограничения по выбору регистров, которые могут использоваться в данном режиме. В табл. 2.16 показаны режимы адресации x86 и указаны те регистры общего назначения, которые не могут быть использованы в каждом из режимов, а также то, как можно получить такой же результат с использованием MIPS-инструкций.

Целочисленные операции x86

Микропроцессор 8086 предоставляет поддержку как для 8-разрядных (байт), так и для 16-разрядных (слово) типов данных. Микропроцессор 80386 добавляет к архитектуре x86 32-разрядные адреса и данные (двойные слова). (AMD64 добавляет 64-разрядные адреса и данные, называемые четверными словами; но в этом разделе мы будем придерживаться того, что имеется в микропроцессоре 80386.) Различия в типе данных относятся к регистровым операциям, а также к обращениям к памяти.

Таблица 2.15. Типы арифметических, логических инструкций и инструкций переноса данных. В x86 доступны все показанные комбинации. Единственное ограничение заключается в отсутствии режима память-память. Непосредственные данные могут иметь длину 8, 16 или 32 разряда; в качестве регистра может использоваться любой из 14 старших регистров, показанных на рис. 2.15 (не могут использоваться регистры EIP или EFLAGS)

Тип операнда источника-приемника	Второй операнд-источник
Регистр	Регистр
Регистр	Непосредственное значение
Регистр	Память
Память	Регистр
Память	Непосредственное значение

Рис. 2.17. Набор регистров микропроцессора 80386. Начиная с этого микропроцессора верхние 8 регистров были расширены до 32 разрядов, а также получили возможность использоваться в качестве регистров общего назначения

Почти каждая операция воздействует как на 8-разрядные данные, так и на данные, имеющие более крупный размер. Этот размер определяется режимом, который может быть 16-, либо 32-разрядным.

Разумеется, некоторым программам требуется работать с данными всех трех размеров, поэтому архитектура 80386 предоставляет удобный способ указания каждой версии без существенного увеличения размера кода. Было решено, что в большинстве программ доминируют либо 16-разрядные, либо 32-разрядные данные, и поэтому имеет смысл предоставить возможность установки большего размера по умолчанию. Этот размер данных по умолчанию устанавливается в одном из разрядов в регистре сегмента кода. Для отмены размера данных, используемого по умолчанию, к инструкции прикрепляется 8-разрядный *префикс*, сообщающий машине, что для этой инструкции нужно использовать больший размер.

Таблица 2.16. 32-разрядные режимы адресации x86 с ограничениями по использованию регистров и эквивалентными им MIPS-кодами. Режим адресации «база плюс масштабируемый индекс», отсутствующий в ARM или MIPS, включен для того, чтобы избежать умножения на 4 (показатель масштабирования 2) и превратить индекс в регистре в адрес байта (см. листинги 2.2, 2.3 и 2.5). Показатель масштабирования 1 используется для 16-разрядных данных, а показатель масштабирования 3 — для 64-разрядных данных. Показатель масштабирования 0 означает, что адрес не масштабируется. Если смещение во втором или четвертом режиме длиннее 16 разрядов, то MIPS-эквиваленту режима понадобятся две дополнительные инструкции: `lui` для загрузки верхних 16-разрядов смещения и `add` для суммирования верхнего адреса с базовым регистром `$s1`. (Intel дала тому, что она называет базовым режимом адресации, два разных имени — базовый и индексный — но по сути это одно и то же, и мы здесь будем чередовать эти названия.)

Режим	Описание	Ограничения	MIPS-эквивалент
Косвенно-регистровый	Адрес находится в регистре	Только не ESP и EBP	`lw $s0,0($s1)`
Режим, использующий базу с 8- или 32-разрядным смещением	Адрес содержится в базовом регистре плюс смещение	Только не ESP	`lw $s0,100($s1) # <= 16-разр.` `# смещение`

База плюс масштабируемый индекс	Адрес — это База + ($2^{масштаб}$ × Индекс), где масштаб имеет значение 0,1,2 или 3	База: любой GRP
		Индекс: только не ESP
			`mul $t0,$s2,4`
			`add $t0,$t0,$s1`
			`lw $s0,0($t0)`

База плюс масштабируемый индекс с 8- или 32-разрядным смещением	Адрес — это База + ($2^{масштаб}$ × Индекс) + смещение, где масштаб имеет значение 0,1,2 или 3	База: любой GRP
		Индекс: только не ESP
			`mul $t0,$s2,4`
			`add $t0,$t0,$s1`
			`lw $s0,100($t0) # <= 16-разр.`
			`# смещение`

Решение о применении префикса было позаимствовано у микропроцессора 8086, который позволял использовать несколько префиксов для изменения характера поведения инструкции. Три изначальных префикса отменяли регистр сегмента, используемый по умолчанию, блокировали шину для поддержки синхронизации (см. раздел 2.11) или повторяли следующие за ними инструкцию до тех пор, пока значение регистра ECX не снижалось до нуля. Этот последний префикс был предназначен для работы в паре с инструкцией перемещения байта, чтобы перемещать разное количество байтов. Микропроцессор 80386 также добавляет префикс, отменяющий размер адреса, используемый по умолчанию.

Целочисленные операции x86 можно разделить на четыре основных класса.

1. Инструкции перемещения данных, включая перемещение, проталкивание данных в стек и выталкивание их оттуда.

2. Арифметические и логические инструкции, включая тестирование и целочисленные и десятичные арифметические операции.
3. Инструкции управления порядком выполнения программы, включая условные и безусловные переходы, вызовы и возвраты.
4. Строковые инструкции, включая перемещение строки и сравнение строк.

Первые две категории ничего особенного собой не представляют, за исключением того, что инструкции арифметических и логических операций позволяют быть приемником либо регистру, либо месту в памяти. В табл. 2.17 показаны некоторые типичные инструкции x86 и их функции.

Таблица 2.17. Некоторые типичные инструкции x86 и их функции. Список часто используемых операций показан в табл. 2.17. Инструкция CALL сохраняет EIP следующей инструкции в стеке. (В изделиях Intel EIP используется в качестве счетчика команд — PC)

Инструкция	Функция
je имя	если равно (код условия) (EIP=имя), EIP–128 <= имя < EIP+128
jmp имя	EIP=имя
call имя	SP=SP–4; M[SP]=EIP+5; EIP=имя;
movw EBX,[EDI+45]	EBX=M[EDI+45]
push ESI	SP=SP–4; M[SP]=ESI
pop EDI	EDI=M[SP]; SP=SP+4
add EAX,#6765	EAX= EAX+6765
test EDX,#42	Установка кода условий (флагов) с помощью EDX и 42
movsl	M[EDI]=M[ESI]; EDI=EDI+4; ESI=ESI+4

Условные переходы в x86, как и в ARM, основаны на *кодах условий*, или *флагах*. Коды условий устанавливаются в качестве побочного результата операции и используются чаще всего для сравнения значения результата с нулем. Затем переходы проверяют коды условий. Переход на основе значения счетчика команд должен быть указан в виде количества байтов, поскольку, в отличие от ARM и MIPS, не все инструкции 80386 имеют длину 4 байта.

Строковые инструкции являются той самой частью, которая досталась семейству x86 по наследству от микропроцессора 8080 и в большинстве программ обычно не используются. Зачастую они работают медленнее, чем их программные эквиваленты (см. описание заблуждения в разделе 2.17 «Заблуждения и недоразумения»).

Некоторые целочисленные инструкции x86 показаны в табл. 2.18. Многие инструкции доступны как формате байтов, так и в формате слов.

Кодирование инструкций x86

Оставив напоследок все самое наихудшее, следует отметить, что кодирование инструкций микропроцессора 80386 представляет собой весьма нелегкую задачу, в решении которой используется множество различных форматов инструкций — от 1 байта при отсутствии операндов до 15 байт.

Формат некоторых инструкций из табл. 2.18 показан на рис. 2.18. Байт кода операции (opcode) обычно содержит бит, сообщающий о том, какой операнд используется, 8- или 32-разрядный. Для некоторых инструкций opcode может включать режим адресации и регистр; это справедливо для многих инструкций, имеющих форму «регистр = регистр операция непосредственное значение». Другие инструкции используют «постбайт» или дополнительный байт opcode, обозначаемый «mod, reg, r/m», который содержит информацию о режиме адресации. Этот постбайт используется для многих инструкций, которые обращаются к памяти. Для режима адресации «база плюс масштабируемый индекс» используется второй постбайт, обозначаемый «sc, index, base».

Таблица 2.18. Некоторые типичные операции, осуществляемые в системах x86. Многие операции используют формат регистр-память, где либо источник, либо приемник может быть местом в памяти, а другой операнд может быть регистром или непосредственным значением

Инструкция	Значение
Управление	**Условные и безусловные переходы**
jnz, jz	Переход при соответствующем условии на EIP + 8-разрядное смещение; JNE (для JNZ), JE (для JZ) являются альтернативными именами
jmp	Безусловный переход — 8- или 16-разрядное смещение
call	Вызов подпрограммы — 16-разрядное смещение; адрес возврата проталкивается в стек
ret	Выталкивание адреса возврата из стека и переход на этот адрес
loop	Циклический переход — декремент ECX; переход на EIP + 8-разрядное смещение, если ECX ≠ 0
Перенос данных	**Перемещение данных между регистрами или между регистром и памятью**
move	Перемещение между двумя регистрами или между регистром и памятью
push, pop	Проталкивание операнда-источника в стек; выталкивание операнда из вершины стека в регистр.
les	Загрузка ES и одного из регистров общего назначения (GPR) из памяти
Арифметика, логика	**Арифметические и логические операции, использующие регистры данных и память**
add, sub	Сложение источника с приемником; вычитание источника из приемника; формат регистр-память

Инструкция	Значение
cmp	Сравнение источника и приемника; формат регистр-память
shl, shr, rcr	Сдвиг влево; логический сдвиг вправо; вращение вправо с установкой кода условия carry (перенос) по заполнению
cbw	Превращение байта в восьми самых правых битах регистра EAX в 16-разрядное слово в правой части EAX
test	Логическое И над источником и приемником, установка кодов условий
inc, dec	Инкремент приемника; декремент приемника
or, xor	Логическое ИЛИ; исключающее ИЛИ; формат регистр-память
Работа со строками	**Перемещение между строковыми операндами; длина задается префиксом повторения**
movs	Копирование из строки-источника в место назначения путем инкремента ESI и EDI; может повторяться
lods	Загрузка байта, слова или двойного слова из строки в регистр EAX

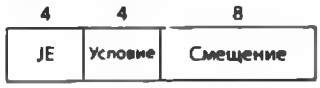
На рис. 2.19 показано кодирование двух постбайтовых спецификаторов адресов, указанных для 16-разрядного и 32-разрядного режима. К сожалению, чтобы до конца понять, какие регистры и какой режим адресации доступны, нужно разобраться со всеми режимами адресации, а иногда даже и с кодированием инструкций.

Итог знакомства с x86

У Intel 16-разрядный микропроцессор появился на два года раньше, чем более элегантные по архитектуре изделия его конкурентов, например микропроцессор Motorola 68000, и это обстоятельство стало главным фактором выбора микропроцессора 8086 в качестве центрального процессора для компьютера IBM PC. Инженеры Intel в целом согласны с тем, что x86 труднее создавать, чем такие компьютерные системы, как ARM и MIPS, но более широкие рыночные возможности позволяют выделять больше ресурсов на преодоление дополнительных сложностей. Стилевые недостатки x86 компенсируются количественными показателями, создавая этому семейству привлекательность в обозримой перспективе.

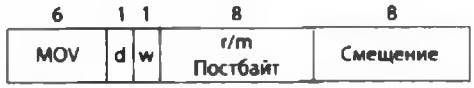
Этому способствует то обстоятельство, что наиболее часто используемые компоненты архитектуры x86 не слишком сложны в реализации, судя по тому, что AMD и Intel продемонстрировали, быстро подняв производительность программ целочисленного вычисления с того уровня, который был в 1978 году. Чтобы достичь этого уровня производительности, компиляторы должны избегать тех частей архитектуры, от которых трудно добиться работы с высокой скоростью.

а) JE EIP + смещение

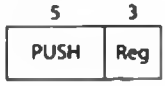


б) CALL

в) MOV EBX, [EDI + 45]



г) PUSH ESI



д) ADD EAX, #6765

е) TEST EDX, #42

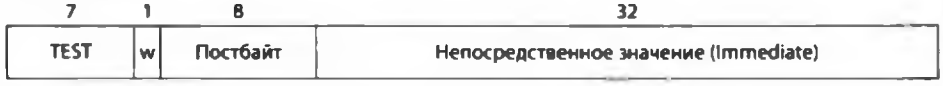


Рис. 2.18. Форматы типичной инструкции x86. Кодирование постбайта показано на рис. 2.19. Многие инструкции имеют одноразрядное поле w, которое сообщает о том, является ли операция байтом или двойным словом. Поле d в инструкции MOV используется в инструкциях, которые могут перемещать данные в память или из памяти, и показывает направление перемещения. Инструкция ADD требует 32 бита для поля непосредственного значения — immediate, поскольку в 32-разрядном режиме непосредственные значения имеют размер либо 8, либо 32 разряда. В инструкции TEST поле immediate имеет длину 32 разряда, поскольку 8-разрядных непосредственных значений при тестировании в 32-разрядном режиме не существует. В общем, длина инструкций может варьироваться от 1 до 17 байт. Большая длина обусловливается дополнительными однобайтными префиксами, наличием как 4-байтных непосредственных значений, так и 4-байтных адресов перемещений, использованием кода операции (opcode) из 2 байтов и спецификатора режима масштабируемого индекса, при котором добавляется еще 1 байт

reg	w = 0	W =	= 1	r/m	mod =0		mod = I		mod = 2		mod = 3
		16b	32 b		16b	32b	16b	32b	16b	32b	
0	AL	AX	EAX	0	addr=BX+SI	=EAX	Тот же адрес, что и в mod=0 + dispS	Тот же адрес, что и в mod=0 + disp8	Тот же адрес, что и в mod=0 + displ6	Тот же адрес, что и в mod=0 + disp32	То же, что и в поле reg
1	CL	cx	ECX	1	addr=BX+DI	=ECX					
2	DL	DX	EDX	2	addr=BP+SI	=EDX					
3	BL	BX	EBX	3	addr=BP+SI	=EBX					
4	AH	SP	ESP	4	addr=SI	=(sb)	SI+disp8	(S/OHdispS	SI+disp8	(s/ty+disp32	•
5	CH	BP	EBP	5	addr=DI	=disp32	DI+disp8	EBP+disp8	DI+displ6	EBP+disp32	•
6	DH	SI	ESI	6	addr=displ6	=ESI	BP+d!sp8	ESI+disp8	BP+displ6	ESI•disp32	•
7	BH	DI	EDI	7	addr=BX	=EDI	BX+disp8	EDI+disp8	BX+displ6	EDI+disp32	•

Рис. 2.19. Кодирование первого спецификатора адреса x86: mod, reg, r/m. Первые четыре столбца показывают кодирование 3-битового поля reg, которое зависит от значения бита w кода операции (opcode) и от режима, в котором находится машина: 16-разрядного режима (8086) или 32-разрядного режима (80386). Остальные столбцы раскрывают значения полей mod и r/m. Значение 3-битового поля r/m зависит от значения 2-битового поля mod и размера адреса. В основном регистры, используемые в вычислениях адреса, перечислены в шестом и седьмом столбцах под общим заголовком mod = 0; в режиме mod = 1 к ним добавляется 8-разрядное смещение (displacement, сокращенно disp), а в режиме mod = 2 — 16-разрядное или 32-разрядное смещение, в зависимости от режима адресации. Исключения следующие: 1) при r/m = 6, когда mod = 1 или mod = 2, в 16-разрядном режиме выбирается BP плюс смещение; 2) при r/m = 5, когда mod = 1 или mod = 2, в 32-разрядном режиме выбирается EBP плюс смещение; и 3) при r/m = 4, в 32-разрядном режиме, когда значение mod ≠ 3, (sib) означает использование режима масштабируемого индекса, показанного в табл. 2.15. Когда mod = 3, поле r/m показывает регистр, используя то же самое кодирование, которое используется в поле reg в сочетании с битом w

2.17. Заблуждения и недоразумения

Заблуждение. *Более впечатляющие инструкции являются признаком более высокой производительности.*

Одной из сильных сторон Intel x86 являются префиксы, которые могут модифицировать выполнение следующих инструкций. Один префикс может задать повторение следующей инструкции до тех пор, пока счетчик не обнулится. Поэтому может показаться вполне естественным использовать для перемещения 32-разрядных слов из памяти в память последовательность, состоящую из инструкции move с префиксом повторения.

Альтернативный метод, использующий стандартные инструкции, имеющиеся во всех компьютерах, заключается в загрузке данных в регистры и в сохранении регистров в памяти. Эта вторая версия данной программы, с повторяющимся кодом для сокращения издержек на создание цикла, осуществляет копирование почти в полтора раза быстрее. Третья версия, в которой используются не целочисленные регистры x86, а более крупные по размеру регистры для чисел с плавающей точкой, копирует почти в два раза быстрее, чем составная инструкция move.

Заблуждение. *Для достижения наивысшей производительности нужно писать программу на языке ассемблера.*

Когда-то компиляторы для языков программирования создавали весьма простые последовательности инструкций; усложнение компиляторов означает сокращение разрыва между кодом, созданным компилятором, и кодом, созданным вручную, и этот разрыв сокращается очень быстро. Фактически, чтобы соревноваться с современными компиляторами, программист, работающий на языке ассемблера, должен полностью овладеть понятиями, изложенными в главах 4 и 5 (конвейеризация процессора и иерархия памяти).

Это соревнование компиляторов и программистов, работающих на языке ассемблера, является одним из примеров тех ситуаций, где люди утрачивают свои позиции. Например, язык C предоставляет возможность программисту подсказать компилятору, какие переменные нужно хранить в регистрах, не сбрасывая их в память. Когда компиляторы были не сильны в распределении регистров, такие подсказки были жизненно необходимы для их работы. Действительно, в некоторых старых книгах, посвященных языку C, уделяется довольно много места примерам, эффективно использующим подсказки, касающиеся регистров. Современные C-компиляторы, как правило, игнорируют такие подсказки, потому что они способны распределить ресурсы лучше программиста.

Даже если написанный вручную код будет работать быстрее, риски, связанные с написанием кода на ассемблере, заключаются в затратах большего времени на его написание и отладку, в потере переносимости кода и в трудностях его дальнейшей поддержки. Одной из общепринятых аксиом разработки программного обеспечения является то, что программирование занимает больше времени, если приходится писать больше строк кода, и совершенно очевидно, что программа на ассемблере потребует куда больше строк, чем программа на C или Java. Кроме того, после создания кода этой программы возникает еще одна угроза, заключающаяся в том, что программа приобретет популярность. Жизнь таких программ всегда оказывается дольше предполагаемого срока, а это значит, что кому-то придется

обновлять код через каждые несколько лет, заставляя его работать с новыми версиями операционных систем или на машинах новой модели. Написание программы на языке высокого уровня, а не на языке ассемблера, не только позволяет будущим компиляторам привязывать код к будущим машинам, но и упрощает поддержку программного обеспечения и позволяет программе работать на большем количестве разнотипных компьютеров.

Заблуждение. *Коммерческая важность совместимости на уровне двоичных кодов означает, что удачный набор инструкций не будет подвергаться изменениям.*

Хотя на обратную совместимость на уровне двоичных кодов никто не покушается, на рис. 2.20 показано, что архитектура x86 прошла очень быстрый путь развития. За тридцать лет существования в среднем добавлялось по одной инструкции в месяц!

Ловушка. *Некоторые забывают о том, что адреса последующих слов в машинах с байтовой адресацией отличаются друг от друга больше чем на единицу.*

Многие программисты на языке ассемблера мучились в поисках ошибок, допущенных из-за предположений, что адрес следующего слова может быть получен путем увеличения адреса в регистре на единицу, а не на размер слова в байтах. Но предупрежден — значит, вооружен!

Ловушка. *Использование указателя на автоматическую переменную за пределами той процедуры, в которой она определена.*

При работе с указателями часто допускается ошибка, заключающаяся в передаче из процедуры результата с указателем на массив, являющийся локальным по отношению к этой процедуре. В соответствии с порядком использования стека, показанным на рис. 2.6, память, в которой содержится локальный массив, будет использована повторно, как только произойдет выход из процедуры. Использование указателей на автоматические переменные может привести к возникновению полного хаоса.

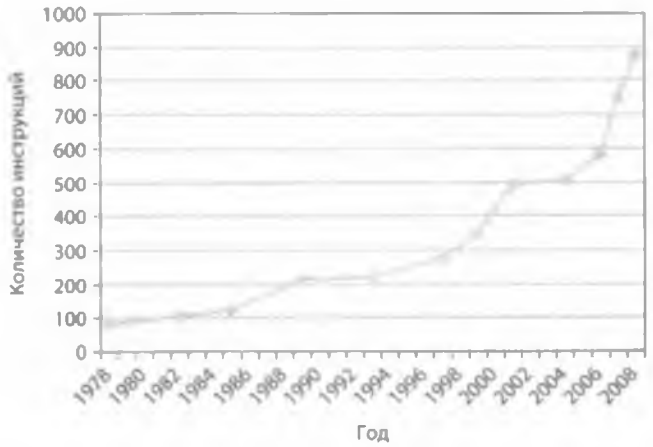


Рис. 2.20. График роста количества инструкций в наборе x86 с течением времени. Хотя некоторые из расширений набора инструкций имели вполне очевидную техническую ценность, скорость изменений, помимо всего, усложняла другим компаниям задачу создания совместимых процессоров

2.18. Заключительные комментарии

Less is more. («В меньшем заключается большее»).
Роберт Браунинг

К двум принципам компьютеров *с программами, хранящимися в памяти*, относятся использование инструкций, неотличимых от чисел, и использование изменяемой памяти для программ.

Эти принципы позволяют использовать одну и ту же машину и экологам, и финансистам, и писателям. Выбор набора инструкций, понимаемых машиной, требует тонкого баланса между количеством инструкций, необходимых для выполнения программы, количеством тактовых циклов, необходимых для выполнения одной инструкции, и тактовой частотой. Как показано в данной главе, к соблюдению этого тонкого баланса создателей набора инструкций приводят четыре принципа конструирования:

1. *Простота предпочитает постоянство.* Постоянство послужило мотивацией для многих свойств набора инструкций MIPS: выдерживание единого размера всех инструкций, неизменное требование трех регистровых операндов в арифметических инструкциях и нахождение полей регистров в одних и тех же местах в формате каждой инструкции.
2. *Чем меньше, тем быстрее.* Желание получить наивысшую скорость работы явилось причиной того, что в MIPS имеется не более 32 регистров.
3. *Часто встречающиеся задачи должны выполняться быстрее.* В качестве примеров ускорения часто встречающихся в MIPS задач можно привести адресацию относительно счетчика команд для условных переходов и непосредственную адресацию для больших операндов-констант.
4. *Хорошая конструкция требует удачных компромиссов.* Одним из примеров следования этому принципу в MIPS может послужить компромисс между предоставлением для инструкций более длинных адресов и констант и сохранением одинаковой длины для всех инструкций.

Над этим машинным уровнем находится язык ассемблера, который уже может быть прочитан человеком. Ассемблер транслируется в понятные машине двоичные числа и даже «расширяет» набор за счет создания символьных инструкций, поддержка которых отсутствует в оборудовании. В качестве примеров можно привести разбиение на приемлемые по размеру части слишком длинных констант или адресов, присваивание собственных имен часто используемым комбинациям инструкций и т. д. В табл. 2.19 перечислены рассмотренные нами MIPS-инструкции, в числе которых имеются как настоящие, так и псевдоинструкции.

Каждая категория MIPS-инструкций ассоциируется с конструкциями, имеющимися в языках программирования:

- Арифметические инструкции соответствуют операциям, имеющимся в инструкциях присваивания.
- Инструкции переноса данных чаще всего применяются при работе с такими структурами данных, как массивы или структуры.

- Условные переходы используются в инструкциях `if` и циклах.
- Безусловные переходы используются в вызовах процедур и возвратах, а также в инструкциях `case-switch`.

Отношение к этим инструкциям разное, популярность некоторых из них доминирует над популярностью всех остальных. Востребованность каждого класса инструкций при выполнении контрольных задач SPEC2006 показана, к примеру, в табл. 2.20. Изменяющаяся востребованность инструкций играет важную роль в главах, посвященных операционному блоку, блоку управления и конвейеризации.

После того как в главе 3 будет рассмотрена компьютерная арифметика, мы покажем всю остальную часть архитектуры набора инструкций MIPS.

Таблица 2.19. Рассмотренные нами MIPS-инструкции, где сначала указаны реальные, а затем псевдоинструкции. Более подробная информация о представленных в данной главе инструкциях дана в табл. 2.1

MIPS-инструкции	Имя	Формат
add (сложение)	add	R
subtract (вычитание)	sub	R
add immediate (непосредственное сложение)	addi	I
load word (загрузка слова)	lw	I
store word (сохранение слова)	sw	I
load half (загрузка полуслова)	lh	I
load half unsigned (загрузка полуслова без знака)	lhu	I
store half (сохранение полуслова)	sh	I
load byte (загрузка байта)	lb	I
load byte unsigned (загрузка байта без знака)	lbu	I
store byte (сохранение байта)	sb	I
load linked (загрузка связанная)	ll	I
store conditional (сохранение условное)	sc	I
load upper immediate (загрузка непосредственного значения в верхние разряды)	lui	I
И (and)	and	R
ИЛИ (or)	or	R
ИЛИ НЕ (nor)	nor	R
and immediate (непосредственное И)	andi	I
or immediate (непосредственное ИЛИ)	ori	I
shift left logical (логический сдвиг влево)	sll	R
shift right logical (логический сдвиг вправо)	srl	R
branch on equal (переход при равенстве)	beq	I
branch on not equal (переход при неравенстве)	bne	I
set less than (установка «если меньше чем»)	slt	R

продолжение ⇩

Таблица 2.19 (*продолжение*)

MIPS-инструкции	Имя	Формат
set less than immediate (установка «если меньше чем непосредственное значение»)	slti	I
set less than immediate unsigned (установка «если меньше чем непосредственное значение без знака»)	sltiu	I
jump (переход)	j	J
jump register (переход по регистру)	jr	R
jump and link (переход и ссылка)	jal	J
MIPS-псевдоинструкции	**Имя**	**Формат**
move (перемещение)	move	R
Multiply (умножение)	mult	R
multiply immediate (умножение на непосредственное значение)	multi	I
load immediate (загрузка непосредственного значения)	li	I
branch less than (переход «если меньше чем»)	blt	I
branch less than or equal (переход «если меньше чем или равно»)	ble	I
branch greater than (переход «если больше чем»)	bgt	I
branch greater than or equal (переход «если больше чем или равно»)	bge	I

Таблица 2.20. Классы MIPS-инструкций, примеры, соответствия в языковых конструкциях программ высокого уровня и процентное соотношение MIPS-инструкций, выполняемых по категориям для усредненных контрольных задач SPEC2006. На рис. 3.26 в главе 3 показано среднее процентное соотношение отдельных выполняемых инструкций MIPS

Класс инструкций	Примеры из MIPS	Соответствия в языках высокого уровня	Частота использования	
			Целочисл.	С плав. точкой
Арифметические	add, sub, addi	Операции в инструкциях присваивания	16%	48%
Переноса данных	lw, sw, lb, lbu, lh, lhu, sb, lui	Ссылки на структуры данных, например на массивы	35%	36%
Логические	and, or, nor, andi, ori, sll, srl	Операции в инструкциях присваивания	12%	4%
Условного перехода	beq, bne, slt, slti, sltiu	Инструкции `if` и циклы	34%	8%
Безусловного перехода	j, jr, jal	Вызов процедур, возврат из процедур и инструкции `case-switch`	2%	0%

2.19. Упражнения

Предоставлены Джоном Оливером (John Oliver), Калифорнийский политехнический государственный университет, Сан-Луис-Обисно, с добавлениями от Николь Кайян (Nicole Kaiyan) (Университет Аделаиды) и Милоша Првуловика (Milos Prvulovic) (Технологический институт Джорджии).

Ваша цель должна заключаться в изучении набора реальных MIPS-инструкций, и если ставится вопрос о подсчете количества инструкций, нужно брать в расчет не псевдоинструкции, а реальные инструкции, подлежащие выполнению.

Встречаются случаи, требующие применения псевдоинструкций (например, инструкции `la`, когда реальное значение на время ассемблирования еще не известно). Во многих случаях такие инструкции очень удобны и позволяют получить код, который легче читается (например, инструкции `li` и `move`). Если псевдоинструкции выбираются из этих соображений, то, пожалуйста, добавьте пару предложений к вашему решению с обоснованием того, какие псевдоинструкции применены и почему.

Упражнение 2.1

Следующие задачи связаны с трансляцией с языка C на язык ассемблера MIPS. Предположим, что даны переменные `g`, `h`, `i` и `j`, которые могут рассматриваться, согласно объявлению в C-программе, в качестве 32-разрядных целых чисел.

а	`f = g + h + i + j;`
б	`f = g + (h + 5);`

2.1.1 [5] <2.2> Каким будет ассемблерный код MIPS для каждой из показанных выше инструкций на языке C? Используйте минимальное количество ассемблерных инструкций MIPS.

2.1.2 [5] <2.2> Сколько потребуется инструкций на языке ассемблера MIPS, чтобы выполнить каждую из показанных выше инструкций на языке C?

2.1.3 [5] <2.2> Каким будет конечное значение переменной `f`, если переменные `f`, `g`, `h`, `i` и `j` имеют соответственно значения 1, 2, 3, 4 и 5?

Следующие задачи связаны с трансляцией с MIPS на C. Предположим, что даны переменные `g`, `h`, `i` и `j`, которые могут рассматриваться, согласно объявлению в C-программе, в качестве 32-разрядных целых чисел.

а	`add f, g, h`
б	`addi f, f, 1` `add f, g, h`

2.1.4 [5] <2.2> Какой будет соответствующая инструкция на языке C для показанных выше MIPS-инструкций?

2.1.5 [5] <2.2> Каким будет конечное значение переменной `f`, если переменные `f`, `g`, `h` и `i` имеют соответственно значения 1, 2, 3 и 4?

Упражнение 2.2

Следующие задачи связаны с трансляцией с C на MIPS. Предположим, что даны переменные g, h, i и j, которые могут рассматриваться, согласно объявлению в C-программе, в качестве 32-разрядных целых чисел.

а	`f = f + f + i;`
б	`f = g + (j + 2);`

2.2.1 [5] <2.2> Каким будет ассемблерный код MIPS для каждой из показанных выше инструкций на языке C? Используйте минимальное количество ассемблерных инструкций MIPS.

2.2.2 [5] <2.2> Сколько потребуется инструкций на языке ассемблера MIPS, чтобы выполнить каждую из показанных выше инструкций на языке C?

2.2.3 [5] <2.2> Каким будет конечное значение переменной f, если переменные f, g, h и i имеют соответственно значения 1, 2, 3 и 4?

Следующие задачи связаны с трансляцией с MIPS на C. Предположим, что даны переменные g, h, i и j, которые могут рассматриваться, согласно объявлению в C-программе, в качестве 32-разрядных целых чисел.

а	`add  f, f, h`
б	`sub  f, $0, f` `addi  f, f, 1`

2.2.4 [5] <2.2> Какой будет соответствующая инструкция на языке C для показанных выше MIPS-инструкций?

2.2.5 [5] <2.2> Каким будет конечное значение переменной f, если переменные f, g, h и i имеют соответственно значения 1, 2, 3 и 4?

Упражнение 2.3

Следующие задачи связаны с трансляцией с C на MIPS. Предположим, что даны переменные g, h, i и j, которые могут рассматриваться, согласно объявлению в C-программе, в качестве 32-разрядных целых чисел.

а	`f = f + g + h + i + j + 2;`
б	`f = g - (f + 5);`

2.3.1 [5] <2.2> Каким будет ассемблерный код MIPS для каждой из показанных выше инструкций на языке C? Используйте минимальное количество ассемблерных инструкций MIPS.

2.3.2 [5] <2.2> Сколько потребуется инструкций на языке ассемблера MIPS, чтобы выполнить каждую из показанных выше инструкций на языке C?

2.3.3 [5] <2.2> Каким будет конечное значение переменной f, если переменные f, g, h, i и j имеют соответственно значения 1, 2, 3, 4 и 5?

Следующие задачи связаны с трансляцией с MIPS на C. Предположим, что даны переменные g, h, i и j, которые могут рассматриваться, согласно объявлению в C-программе, в качестве 32-разрядных целых чисел.

а	`add f, g, h`
б	`addi h, f, 1` `sub f, g, h`

2.3.4 [5] <2.2> Какой будет соответствующая инструкция на языке C для показанных выше MIPS-инструкций?

2.3.5 [5] <2.2> Каким будет конечное значение переменной f, если переменные g, h и i имеют соответственно значения 1, 2, 3 и 4?

Упражнение 2.4

Следующие задачи связаны с трансляцией с C на MIPS. Предположим, что значения переменных f, g, h, i и j присвоены соответственно регистрам $s0, $s1, $s2, $s3 и $s4. Предположим, что базовый адрес массивов A и B находится соответственно в регистрах $s6 и $s7.

а	`f = g + h + B[4];`
б	`f = g - A[B[4]];`

2.4.1 [10] <2.2, 2.3> Каким будет ассемблерный код MIPS для каждой из показанных выше инструкций на языке C?

2.4.2 [5] <2.2, 2.3> Сколько потребуется инструкций на языке ассемблера MIPS, чтобы выполнить каждую из показанных выше инструкций на языке C?

2.4.3 [5] <2.2, 2.3> Сколько разных регистров необходимо для выполнения каждой из показанных выше инструкций на языке C?

Следующие задачи связаны с трансляцией с MIPS на C. Предположим, что значения переменных f, g, h, i и j присвоены соответственно регистрам $s0, $s1, $s2, $s3 и $s4. Предположим, что базовый адрес массивов A и B находится соответственно в регистрах $s6 и $s7.

а	`add $s0, $s0, $s1`
	`add $s0, $s0, $s2`
	`add $s0, $s0, $s3`
	`add $s0, $s0, $s4`

б	`lw $s0, 4($s6)`

2.4.4 [10] <2.2, 2.3> Какой будет соответствующая инструкция на языке C для показанных выше MIPS-инструкций?

2.4.5 [5] <2.2, 2.3> Перепишите код показанных выше ассемблерных MIPS-инструкций с целью минимизации их количества, необходимого для выполнения той же самой функции (если это возможно).

2.4.6 [5] <2.2, 2.3> Сколько регистров потребуется для выполнения показанного выше ассемблерного кода MIPS? Если вам удастся переписать показанный выше код, то какое минимальное количество регистров для него потребуется?

Упражнение 2.5

В следующих задачах будут исследоваться операции с памятью в контексте MIPS-процессора. Показанная ниже таблица содержит значения массива, сохраненного в памяти.

а	Адрес	Данные
	12	1
	8	6
	4	4
	0	2
б	Адрес	Данные
	16	1
	12	2
	8	3
	4	4
	0	5

2.5.1 [10] <2.2, 2.3> Используя показанные выше состояния памяти, напишите код на языке C, сортирующий данные по возрастающей и помещающий самое маленькое значение в самый младший из показанных адресов памяти. Предположим, что показанные данные представляют собой переменную C по имени `Array`, которая является целочисленным массивом (со значениями типа `int`). Предположим, что данная конкретная машина использует байтовую адресацию и ее слово состоит из 4 байт.

2.5.2 [10] <2.2, 2.3> Используя показанные выше состояния памяти, напишите код на языке ассемблера MIPS, сортирующий данные по возрастающей и помещающий самое маленькое значение в самый младший из показанных адресов памяти. Используйте для этого минимально возможное количество MIPS-инструкций. Предположим, что базовый адрес массива `Array` хранится в регистре `$s6`.

2.5.3 [5] <2.2, 2.3> Сколько инструкций ассемблерного кода MIPS потребуется для сортировки показанного выше массива? Сколько инструкций понадобится для этого, если запретить использование в инструкциях `lw` и `sw` поля непосредственного значения?

В следующих задачах будет исследоваться преобразование шестнадцатеричных чисел в числа других форматов.

а	`0x12345678`
б	`0xbeadf00d`

2.5.4 [5] <2.3> Преобразуйте показанные выше шестнадцатеричные числа в десятичные.

2.5.5 [5] <2.3> Покажите, как данные, приведенные в таблице, были бы помещены в память машины с форматом записи младшего байта по наименьшему адресу (little-endian) и в память машины с форматом записи младшего байта по

наибольшему адресу (big-endian). Предположим, что данные помещаются в память начиная с адреса 0.

Упражнение 2.6

Следующие задачи связаны с трансляцией с C на MIPS. Предположим, что значения переменных f, g, h, i и j присвоены соответственно регистрам $s0, $s1, $s2, $s3 и $s4. Предположим, что базовый адрес массивов A и B находится соответственно в регистрах $s6 и $s7.

а	`f = -g + h + B[1];`
б	`f = A[B[g]+1];`

2.6.1 [10] <2.2, 2.3> Каким будет ассемблерный код MIPS для каждой из показанных выше инструкций на языке C?

2.6.2 [5] <2.2, 2.3> Сколько потребуется инструкций на языке ассемблера MIPS, чтобы выполнить каждую из показанных выше инструкций на языке C?

2.6.3 [5] <2.2, 2.3> Сколько регистров потребуется для выполнения каждой из показанных выше инструкций на языке C, при использовании ассемблерного кода MIPS?

Следующие задачи связаны с трансляцией с MIPS на C. Предположим, что значения переменных f, g, h, i и j присвоены соответственно регистрам $s0, $s1, $s2, $s3 и $s4. Предположим, что базовый адрес массивов A и B находится соответственно в регистрах $s6 и $s7.

а	`add $s0, $s0, $s1`
	`add $s0, $s3, $s2`
	`add $s0, $s0, $s3`

б	`addi $s6, $s6, -20`
	`add $s6, $s6, $s1`
	`lw $s0, 8($s6)`

2.6.4 [5] <2.2, 2.3> Какой будет соответствующая инструкция на языке C для показанных выше MIPS-инструкций?

2.6.5 [5] <2.2, 2.3> Используя показанный выше код на языке ассемблера MIPS, предположим, что регистры $s0, $s1, $s2, $s3 содержат соответственно значения 10, 20, 30 и 40. Также предположим, что регистр $s6 содержит значение 256, а в памяти содержатся следующие значения:

Адрес	Значение
256	100
260	200
264	300

Определите значение регистра $s0 в конце ассемблерного кода.

2.6.6 [10] <2.3, 2.5> Покажите для каждой MIPS-инструкции значения полей op, rs и rt. Для инструкций I-типа покажите содержимое поля непосредственного значения, а для инструкций R-типа покажите значение поля rd.

Упражнение 2.7

В следующих задачах исследуются преобразования двоичных чисел со знаком и без знака в десятичные числа.

а	1010 1101 0001 0000 0000 0000 0000 00102
б	1111 1111 1111 1111 1011 0011 0101 00112

2.7.1 [5] <2.4> Каким числам по основанию 10 соответствуют показанные выше примеры, при условии, что они представляют собой целые числа с дополнением до двух?

2.7.2 [5] <2.4> Каким числам по основанию 10 соответствуют показанные выше примеры, при условии, что они представляют собой целые числа без знака?

2.7.3 [5] <2.4> Каким шестнадцатеричным числам соответствуют показанные выше примеры?

В следующих задачах исследуются преобразования десятичных чисел в двоичные числа со знаком и без знака.

а	214748364710
б	100010

2.7.4 [5] <2.4> Преобразуйте показанные выше десятичные числа в двоичные с дополнением до двух.

2.7.5 [5] <2.4> Преобразуйте показанные выше десятичные числа в шестнадцатеричные с дополнением до двух.

2.7.6 [5] <2.4> Превратите показанные выше десятичные числа в отрицательные и преобразуйте их в шестнадцатеричные с дополнением до двух.

Упражнение 2.8

Следующие задачи связаны с расширением и переполнением. В регистрах `$s0` и `$s1` содержатся значения, показанные в таблице. Вам нужно будет выполнить MIPS-операции над этими регистрами и показать результат.

а	$s0 = 7000000016, $s1 = 0x0FFFFFFF16
б	$s0 = 0x4000000016, $s1 = 0x4000000016

2.8.1 [5] <2.4> Каким будет значение регистра `$t0` после выполнения следующего ассемблерного кода, если содержимое регистров `$s0` и `$s1` будет таким, как показано выше:

```
add $t0, $s0, $s1
```

Будет ли результат в `$t0` приемлемым или возникнет переполнение?

2.8.2 [5] <2.4> Каким будет значение регистра `$t0` после выполнения следующего ассемблерного кода, если содержимое регистров `$s0` и `$s1` будет таким, как показано выше:

```
sub $t0, $s0, $s1
```

Будет ли результат в `$t0` приемлемым или возникнет переполнение?

2.8.3 [5] <2.4> Каким будет значение регистра $t0 после выполнения следующего ассемблерного кода, если содержимое регистров $s0 и $s1 будет таким, как показано выше:

```
add $t0, $s0, $s1
add $t0, $t0, $s0
```

Будет ли результат в $t0 приемлемым или возникнет переполнение?

В следующих задачах будут выполняться различные MIPS-операции с парой регистров $s0 и $s1. Используя указанные в каждой задаче значения $s0 и $s1, определите, будет ли возникать переполнение.

а	`add $s0, $s0, $s1`
б	`sub $s0, $s0, $s1` `sub $s0, $s0, $s1`

2.8.4 [5] <2.4> Предположим, что регистр $s0 = 0x70000000, а регистр $s1 = 0x10000000. Возникнет ли переполнение при выполнении кода, показанного в таблице?

2.8.5 [5] <2.4> Предположим, что регистр $s0 = 0x40000000, а регистр $s1 = 0x20000000. Возникнет ли переполнение при выполнении кода, показанного в таблице?

2.8.6 [5] <2.4> Предположим, что регистр $s0 = 0x8FFFFFFF, а регистр $s1 = 0xD0000000. Возникнет ли переполнение при выполнении кода, показанного в таблице?

Упражнение 2.9

В показанной ниже таблице содержатся различные значения для регистра $s1. В задачах спрашивается, будет ли возникать переполнение при вычислении указанных операций.

а	2147483647D
б	0xD000000016

2.9.1 [5] <2.4> Предположим, что регистр $s0 = 0x70000000, а регистр $s1 имеет значение, указанное в таблице. Возникнет ли переполнение при выполнении инструкции `add $s0, $s0, $s1`?

2.9.2 [5] <2.4> Предположим, что регистр $s0 = 0x80000000, а регистр $s1 имеет значение, указанное в таблице. Возникнет ли переполнение при выполнении инструкции `sub $s0, $s0, $s1`?

2.9.3 [5] <2.4> Предположим, что регистр $s0 = 0x7FFFFFFF, а регистр $s1 имеет значение, указанное в таблице. Возникнет ли переполнение при выполнении инструкции `sub $s0, $s0, $s1`?

В показанной ниже таблице содержатся различные значения для регистра $s1. В задачах спрашивается, будет ли возникать переполнение при вычислении указанных операций.

а	1010 1101 0001 0000 0000 0000 0000 00102
б	1111 1111 1111 1111 1011 0011 0101 00112

2.9.4 [5] <2.4> Предположим, что регистр `$s0` = 0x70000000, а регистр `$s1` имеет значение, указанное в таблице. Возникнет ли переполнение при выполнении инструкции `add $s0, $s0, $s1`?

2.9.5 [5] <2.4> Предположим, что регистр `$s0` = 0x70000000, а регистр `$s1` имеет значение, указанное в таблице. Каким будет результат в шестнадцатеричном формате при выполнении инструкции `add $s0, $s0, $s1`?

2.9.6 [5] <2.4> Предположим, что регистр `$s0` = 0x70000000, а регистр `$s1` имеет значение, указанное в таблице. Каким будет результат в десятичном формате при выполнении инструкции `add $s0, $s0, $s1`?

Упражнение 2.10

В следующих задачах таблица данных содержит биты, представляющие собой значение opcode инструкции. Вам нужно будет перевести записи в ассемблерный код и определить, какой формат MIPS-инструкции представляют эти биты.

а	1010 1110 0000 1011 0000 0000 0000 01002
б	1000 1101 0000 1000 0000 0000 0100 00002

2.10.1 [5] <2.5> Какую инструкцию представляют показанные выше двоичные записи?

2.10.2 [5] <2.5> Какой тип инструкции (I-тип, R-тип) представляют показанные выше двоичные записи?

2.10.3 [5] <2.4, 2.5> Если показанные выше двоичные записи были бы данными, то какое шестнадцатеричное число они бы представляли?

В следующих задачах таблица данных содержит MIPS-инструкции. Вам нужно будет перевести записи в биты opcode и определить формат MIPS-инструкции.

а	`add $t0, $t0, $zero`
б	`lw $t1, 4($s3)`

2.10.4 [5] <2.4, 2.5> Покажите шестнадцатеричное представление для каждой из заданных выше инструкций.

2.10.5 [5] <2.5> Какой тип инструкции (I-тип, R-тип) представляют показанные выше инструкции?

2.10.6 [5] <2.5> Какое шестнадцатеричное представление имеют поля opcode, `rs` и `rt` в этой инструкции? Какое шестнадцатеричное представление имеют поля `rd` и `funct` для инструкций R-типа? Какое шестнадцатеричное представление имеет поле `immediate` для инструкций I-типа?

Упражнение 2.11

В следующих задачах таблица данных содержит биты, представляющие собой значение opcode инструкции. Вам нужно будет перевести записи в ассемблерный код и определить, какой формат MIPS-инструкции представляют эти биты.

а	0xAE0BFFFC
б	0x8D08FFC0

2.11.1 [5] <2.4, 2.5> Какие двоичные числа являются представлением показанных выше шестнадцатеричных чисел?

2.11.2 [5] <2.4, 2.5> Какие десятичные числа являются представлением показанных выше шестнадцатеричных чисел?

2.11.3 [5] <2.5> Какие инструкции представляют показанные выше шестнадцатеричные числа?

В следующих задачах таблица данных содержит значения различных полей MIPS-инструкций. Вам нужно будет определить инструкцию и MIPS-формат инструкции.

а	op=0, rs=1, rt=2, rd=3, shamt=0, funct=32
б	op=0x2B, rs=0x10, rt=0x5, const=0x4

2.11.4 [5] <2.5> К какому типу инструкций (I-тип, R-тип) относятся показанные выше инструкции?

2.11.5 [5] <2.5> Какие ассемблерные инструкции описаны выше?

2.11.6 [5] <2.4, 2.5> Как выглядит двоичное представление показанных выше инструкций?

Упражнение 2.12

В следующих задачах таблица данных содержит различные модификации, которым может подвергнуться архитектура набора команд MIPS. Вам нужно исследовать влияние этих изменений на формат инструкций MIPS-архитектуры.

а	8 регистров
б	10-разрядные непосредственно указанные константы

2.12.1 [5] <2.5> Если изменился набор инструкций MIPS-процессора, то должен измениться и формат инструкции. Покажите для каждого предложенного выше изменения размер полей в разрядах для инструкции R-типа. Сколько всего разрядов потребуется для каждой инструкции?

2.12.2 [5] <2.5> Если изменился набор инструкций MIPS-процессора, то должен измениться и формат инструкции. Покажите для каждого предложенного выше изменения размер полей в разрядах для инструкции I-типа. Сколько всего разрядов потребуется для каждой инструкции?

2.12.3 [5] <2.5, 2.10> Могут ли предложенные выше изменения уменьшить размер программы на языке ассемблера MIPS и почему? Могут ли предложенные выше изменения увеличить размер программы на языке ассемблера MIPS и почему?

Таблица данных для следующих задач содержит шестнадцатеричные значения. Вам нужно определить, какую MIPS-инструкцию представляет значение, формат этой инструкции.

а	0x01090010
б	0x8D090012

2.12.4 [5] <2.5> Каким будет десятичное значение показанных выше записей?

2.12.5 [5] <2.5> Какую инструкцию представляют показанные выше шестнадцатеричные записи?

2.12.6 [5] <2.4, 2.5> К какому типу инструкций (I-типа, R-типа) относится двоичное представление показанных выше записей? Каким будут значения полей `op` и `rt`?

Упражнение 2.13

Таблица данных для следующих задач содержит значения регистров `$t0` и `$t1`. Вам нужно будет провести над этими регистрами ряд логических операций MIPS.

а	`$t0` = 0x55555555, `$t1` = 0x12345678
б	`$t0` = 0xBEADFEED, `$t1` = 0xDEADFADE

2.13.1 [5] <2.6> Какое значение получит регистр `$t2` после выполнения следующей последовательности инструкций при использовании показанных выше значений?

```
sll $t2, $t0, 4
or  $t2, $t2, $t1
```

2.13.2 [5] <2.6> Какое значение получит регистр `$t2` после выполнения следующей последовательности инструкций при использовании показанных выше значений?

```
sll  $t2, $t0, 4
andi $t2, $t2, -1
```

2.13.3 [5] <2.6> Какое значение получит регистр `$t2` после выполнения следующей последовательности инструкций при использовании показанных выше значений?

```
srl  $t2, $t0, 3
andi $t2, $t2, 0xFFEF
```

Таблица данных для следующих задач содержит различные логические операции MIPS. Вам нужно будет найти результат этих операций для заданных значений регистров $t0 и $t1.

а	sll $t2, $t0, 1 or $t2, $t2, $t1
б	srl $t2, $t0, 1 andi $t2, $t2, 0x00F0

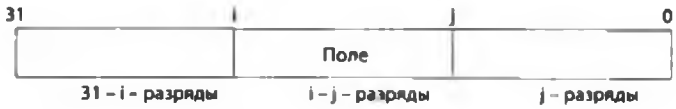
2.13.4 [5] <2.6> Предположим, что $t0 = 0x0000A5A5, а $t1 = 00005A5A. Каким будет значение $t2 после выполнения двух инструкций, показанных в таблице?

2.13.5 [5] <2.6> Предположим, что $t0 = 0xA5A50000, а $t1 = A5A50000. Каким будет значение $t2 после выполнения двух инструкций, показанных в таблице?

2.13.6 [5] <2.6> Предположим, что $t0 = 0xA5A5FFFF, а $t1 = A5A5FFFF. Каким будет значение $t2 после выполнения двух инструкций, показанных в таблице?

Упражнение 2.14

На рисунке показано размещение в регистре $t0 поля, состоящего из нескольких разрядов.



При решении следующих задач вам нужно будет написать MIPS-инструкции для извлечения битов «Поле» из регистра $t0 и помещения их в регистр $t1, в место, показанное в следующей таблице:

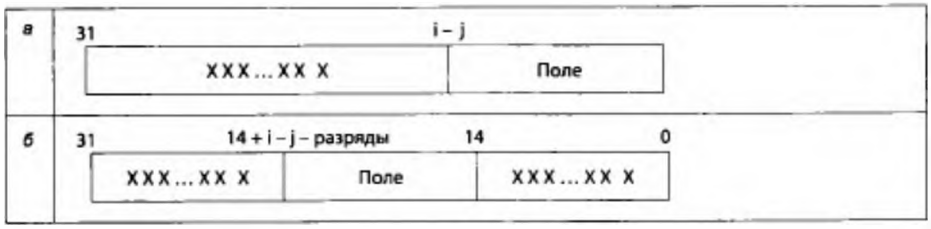
а	31 i – j 000...00 0 Поле
б	31 14 + i - j - разряды 14 0 000...00 0 Поле 000...00 0

2.14.1 [20] <2.6> Найдите самую короткую последовательность MIPS-инструкций, извлекающую поле из $t0 для значений констант i = 22 и j = 5 и помещающую поле в $t1, в формате, показанном в таблице данных.

2.14.2 [5] <2.6> Найдите самую короткую последовательность MIPS-инструкций, извлекающую поле из $t0 для значений констант i = 4 и j = 0 и помещающую поле в $t1, в формате, показанном в таблице данных.

2.14.3 [5] <2.6> Найдите самую короткую последовательность MIPS-инструкций, извлекающую поле из $t0 для значений констант i = 31 и j = 28 и помещающую поле в $t1, в формате, показанном в таблице данных.

При решении следующих задач вам нужно будет написать MIPS-инструкции для извлечения битов «Поле» из регистра $t0, показанного на рисунке, и помещения их в регистр $t1, в те места, которые показаны в таблице. Разряды, обозначенные символами «XXX», не должны измениться.



2.14.4 [20] <2.6> Найдите самую короткую последовательность MIPS-инструкций, извлекающую поле из $t0 для значений констант i = 17 и j = 11 и помещающую поле в $t1, в формате, показанном в таблице данных.

2.14.5 [5] <2.6> Найдите самую короткую последовательность MIPS-инструкций, извлекающую поле из $t0 для значений констант i = 5 и j = 0 и помещающую поле в $t1, в формате, показанном в таблице данных.

2.14.6 [5] <2.6> Найдите самую короткую последовательность MIPS-инструкций, извлекающую поле из $t0 для значений констант i = 31 и j = 29 и помещающую поле в $t1, в формате, показанном в таблице данных.

Упражнение 2.15

Для следующих задач в таблице содержатся логические операции, не включенные в набор инструкций MIPS. Как могут быть реализованы эти инструкции?

а	`andn $t1, $t2, $t3`	`// поразрядное И $t2, !$t3`
б	`xnor $t1, $t2, $t3`	`// поразрядное исключающее ИЛИ НЕ`

2.15.1 [5] <2.6> Показанные выше логические инструкции не включены в набор инструкций MIPS, но имеют описания. Каким будет результат в $t1, если $t2 = 0x00FFA5A5, а $t3 = 0xFFFF003C?

2.15.2 [10] <2.6> Показанные выше логические инструкции не включены в набор инструкций MIPS, но они могут быть синтезированы с использованием одной или нескольких ассемблерных инструкций MIPS. Предоставьте минимальный набор MIPS-инструкций, который мог бы использоваться взамен инструкций, показанных в расположенной выше таблице.

2.15.3 [5] <2.6> Покажите, как выглядит ваша последовательность инструкций, полученная при решении задачи 2.15.2, на уровне битов, выделив каждую из инструкций.

В следующей таблице показаны различные логические инструкции на языке C. В данном упражнении нужно будет вычислить результат выполнения этих ин-

струкций и реализовать инструкции на языке C, используя для этого инструкции на языке ассемблера MIPS.

а	`A = B & C[0];`
б	`A = A ? B : C[0]`

2.15.4 [5] <2.6> В показанной выше таблице даны различные инструкции на языке C, использующие логические операторы. Каким будет конечное значение `A`, если по адресу памяти `C[0]` содержится целочисленное значение 0x00001234, а начальные целочисленные значения переменных `A` и `B` равны 0x00000000 и 0x00002222?

2.15.5 [5] <2.6> Напишите для представленных выше C-инструкций минимальную последовательность инструкций на языке ассемблера MIPS, выполняющую ту же операцию.

2.15.6 [5] <2.6> Покажите, как выглядит ваша последовательность инструкций, полученная при решении задачи 2.15.5, на уровне битов, выделив каждую из инструкций

Упражнение 2.16

Для следующих задач в таблице содержатся различные двоичные значения для регистра `$t0`. Используя значения `$t0`, нужно будет вычислить итог различных переходов.

а	1010 1101 0001 0000 0000 0000 0000 0010$_2$
б	1111 1111 1111 1111 1111 1111 1111 1111$_2$

2.16.1 [5] <2.7> Предположим, что в регистре `$t0` содержится значение из таблицы, а `$t1` имеет значение

1011 1111 1111 1000 0000 0000 0000 0000$_2$

Каким будет значение регистра `$t2` после выполнения следующих инструкций?

```
      slt  $t2, $t0, $t1
      beq  $t2, $zero, ELSE
      j    DONE
ELSE: addi $t2, $zero, 2
DONE:
```

2.16.2 [5] <2.7> Предположим, что в регистре `$t0` содержится значение из таблицы, и оно сравнивается со значением `X` с помощью показанных ниже MIPS-инструкций. Существует ли такое значение `X`, при котором значение `$t2` будет равно 1, и какое это значение?

```
slti $t2, $t0, X
```

2.16.3 [5] <2.7> Предположим, что счетчик команд (`PC`) имеет значение 0x0000 0020. Можно ли воспользоваться инструкцией перехода на языке ассемблера MIPS (`j`) для установки `PC` на адрес, показанный в таблице? Можно ли вос-

пользоваться инструкцией перехода при равенстве на языке ассемблера MIPS (beq) для установки PC на адрес, показанный в расположенной выше таблице?

Для следующих задач в таблице содержатся различные двоичные значения для регистра $t0. Используя значения $t0, нужно будет вычислить итог различных переходов.

а	0x00001000
б	0x20001400

2.16.4 [5] <2.7> Предположим, что в регистре $t0 содержится значение из таблицы. Каким будет значение регистра $t2 после выполнения следующих инструкций?

```
      slt  $t2, $t0, $t0
      bne  $t2, $zero, ELSE
      j    DONE
ELSE: addi $t2, $t2, 2
DONE:
```

2.16.5 [5] <2.6, 2.7> Предположим, что в регистре $t0 содержится значение из таблицы. Каким будет значение регистра $t2 после выполнения следующих инструкций?

```
sll $t0, $t0, 2
slt $t2, $t0, $zero
```

2.16.6 [5] <2.7> Предположим, что счетчик команд (PC) имеет значение 0x2000 0000. Можно ли воспользоваться инструкцией перехода на языке ассемблера MIPS (j) для установки PC на адрес, показанный в таблице? Можно ли воспользоваться инструкцией перехода при равенстве на языке ассемблера MIPS (beq) для установки PC на адрес, показанный в таблице?

Упражнение 2.17

В следующих задачах будут использованы несколько инструкций, не входящих в набор инструкций MIPS.

а	`abs  $t2, $t3        # R[rd] = |R[rt]|`
б	`sgt  $t1, $t2, $t3   # R[rd] = (R[rs] > R[rt]) ? 1:0`

2.17.1 [5] <2.7> В таблице содержатся инструкции, не входящие в набор инструкций MIPS, и описание каждой из этих инструкций. Почему эти инструкции не включены в набор инструкций MIPS?

2.17.2 [5] <2.7> В таблице содержатся инструкции, не входящие в набор инструкций MIPS, и описание каждой из этих инструкций. Каким бы был наиболее подходящий формат для этих инструкций, если они были бы реализованы в набор инструкций MIPS?

2.17.3 [5] <2.7> Найдите для каждой инструкции из таблицы самую короткую последовательность MIPS-инструкций, выполняющую ту же операцию.

Для следующих задач в таблице содержатся фрагменты ассемблерного кода MIPS. Вам нужно будет изучить каждый фрагмент кода и ознакомиться с различными инструкциями перехода MIPS.

а	LOOP: slt $t2, $0, $t1 bne $t2, $zero, ELSE j DONE ELSE: addi $s2, $s2, 2 sub1 $t1, $t1, 1 j LOOP DONE:
б	LOOP: add1 $t2, $0, 0xA LOOP2: add1 $s2, $s2, 2 sub1 $t2, $t2, 1 bne $t2, $0, LOOP2 sub1 $t1, $t1, 1 bne $t1, $0, LOOP DONE:

2.17.4 [5] <2.7> Предположим, что для циклов, написанных на языке ассемблера MIPS, регистр `$t1` имеет начальное значение 10. Каким будет значение регистра `$s2`, если предположить, что начальное значение регистра `$s2` равно нулю?

2.17.5 [5] <2.7> Напишите для каждого из циклов эквивалентную подпрограмму на языке C, исходя из того, что регистрам `$s1`, `$s2`, `$t1` и `$t2` соответствуют целочисленные переменные `A`, `B`, `i` и `temp`.

2.17.6 [5] <2.7> Предположим, что для показанных выше циклов, написанных на языке ассемблера MIPS, регистр `$t1` имеет начальное значение `N`. Сколько MIPS-инструкций будет выполнено?

Упражнение 2.18

Для следующих задач в таблице содержится код на языке C. Вам нужно будет перевести инструкции на языке C в инструкции на языке ассемблера MIPS.

а	for(i=0; i<10; i++) a += b;

б	while (a < 10){
	D[a] = b + a;
	a += 1;
	}

2.18.1 [5] <2.7> Нарисуйте для показанного кода на языке C блок-схему управления последовательностью выполнения программы.

2.18.2 [5] <2.7> Переведите показанный код на языке C в код на языке ассемблера MIPS. Используйте минимально возможное количество инструкций, при условии, что переменные `a`, `b`, `i`, `j` представлены соответственно регистрами `$s0`, `$s1`, `$t0`, `$t1`. Кроме этого, для хранения базового адреса массива `D` используйте регистр `$s2`.

2.18.3 [5] <2.7> Сколько MIPS-инструкций понадобится для реализации кода на языке C? Каким будет общее количество MIPS-инструкций, выполненных для

завершения цикла, если у переменных a и b будут начальные значения 10 и 1, а все элементы массива D будут иметь нулевые начальные значения?

Для следующих задач в таблице содержатся фрагменты кода на языке ассемблера MIPS. Вам нужно будет изучить каждый фрагмент кода и ознакомиться с различными инструкциями перехода MIPS.

а

```
        addi  $t1, $0, 100
LOOP    lw    $s1, 0($s0)
        add   $s2, $s2, $s1
        addi  $s0, $s0, 4
        subi  $t1, $t1, 1
        bne   $t1, $0, LOOP
```

б

```
        addi  $t1, $s0, 400
LOOP    lw    $s1, 0($s0)
        add   $s2, $s2  $s1
        lw    $s1, 4($s0)
        add   $s2, $s2, $s1
        addi  $s0, $s0, 8
        bne   $t1, $s0, LOOP
```

2.18.4 [5] <2.7> Каково общее количество выполняемых MIPS-инструкций?

2.18.5 [5] <2.7> Переведите показанные циклы на язык C, при условии, что в регистре `$t1` содержится целочисленная C-переменная `i`, в регистре `$s2` содержится целочисленная C-переменная `result`, а в регистре `$s0` содержится базовый адрес массива целочисленных значений `MemArray`.

2.18.6 [5] <2.7> Перепишите цикл на языке ассемблера MIPS с целью сократить количество выполняемых MIPS-инструкций.

Упражнение 2.19

Для следующих задач в таблице содержится код функций на языке C. Предположим, что первая функция, представленная в таблице, вызывается первой. Вам нужно будет перевести код этих процедур на языке C в код на языке ассемблера MIPS.

а

```
int compare(int a, int b) {
    if (sub(a, b) >= 0)
        return 1;
    else
        return 0;
}
int sub (int a, int b) {
    return a-b;
}
```

б

```
int fib_iter(int a, int b, int n){
    if(n == 0)
        return b;
    else
        return fib_iter(a+b, a, n-1);
}
```

2.19.1 [15] <2.8> Реализуйте показанный в таблице код на языке C в коде ассемблера MIPS. Сколько потребуется MIPS-инструкций для выполнения функции

2.19.2 [5] <2.8> Функции зачастую могут реализовываться компилятором путем их встраивания в общий код. Такая встроенная функция представляет собой тело функции, скопированное в пространство программы и позволяющее избежать издержек, возникающих при вызове функции. Реализуйте «встроенную» версию показанного в таблице кода на языке C в коде ассемблера MIPS. На сколько увеличится количество инструкций на языке ассемблера MIPS, необходимое для выполнения функции? Предположим, что начальное значение C-переменной n равно 5.

2.19.3 [5] <2.8> Покажите для каждого вызова функции содержимое стека после того, как функция уже была вызвана, при условии, что указатель стека изначально был установлен на адрес `0x7ffffffc` и соблюдаются соглашения по использованию регистров, показанные в табл. 2.5.

В следующих трех задачах этого упражнения используется функция `f`, которая вызывает другую функцию `func`. Код для Си-функции `func` уже откомпилирован в другой модуль с использованием имеющегося в MIPS соглашения по вызовам, показанного в табл. 2.6. В объявлении функции `func` содержится строка `int func(int a, int b)`. Функция `f` имеет следующий код:

a	`int f(int a, int b, int c){`
	`  return func(func(a,b),c);`
	`}`

б	`int f(int a, int b, int c){`
	`  return func(a,b)+func(b,c);`
	`}`

2.19.4 [10] <2.8> Переведите функцию `f` на язык ассемблера MIPS, используя при этом соглашение по вызовам MIPS, показанное в табл. 2.6. Если потребуется использовать регистры от `$t0` до `$t7`, то сначала используйте регистры с меньшими номерами.

2.19.5 [5] <2.8> Можно ли в этой функции задействовать оптимизацию, использующую так называемый вызов хвоста? Если нет, то объясните почему. А если да, то какой будет разница в количестве выполняемых инструкций в `f` при использовании и без использования такой оптимизации?

2.19.6 [5] <2.8> Что мы знаем о содержимом регистров `$t5`, `$s3`, `$ra` и `$sp` непосредственно перед тем, как ваша функция `f`, созданная при выполнении задачи 2.19.4, вернет управление основной программе? Следует помнить, что нам известно содержимое функции `f`, но о функции `func` мы можем судить только по строке ее объявления.

Упражнение 2.20

В этом упражнении используются рекурсивные вызовы процедуры. Для следующих задач в таблице содержится фрагмент кода на языке ассемблера, вычисляющий факториал числа. Но в записях таблицы имеются ошибки, которые вам придется исправить

а	FACT: addi $sp, $sp, -8
	sw $ra, 4($sp)
	sw $a0, 0($sp)
	slti $t0, $a0, 1
	beq $t0, $0, L1
	addi $v0, $0, 1
	addi $sp, $sp, 8
	jr $ra
	L1: addi $a0, $a0, -1
	jal FACT
	lw $a0, 4($sp)
	lw $ra, 0($sp)
	addi $sp, $sp, 8
	mul $v0, $a0, $v0
	jr $ra

б	FACT addi $sp, $sp, -8
	sw $ra, 4($sp)
	sw $a0, 0($sp)
	slti $t0, $a0, 1
	beq $t0, $0, L1
	addi $v0, $0, 1
	addi $sp, $sp, 8
	jr $ra
	L1: addi $t0, $t0, -1
	jal FACT
	lw $a0, 4($sp)
	lw $ra, 0($sp)
	addi $sp, $sp, 8
	mul $v0, $a0, $v0
	jr $ra

2.20.1 [5] <2.8> Показанная выше программа на языке ассемблера MIPS вычисляет факториал заданного числа. Это целое число передается в регистре `$a0`, а результат возвращается в регистре `$v0`. В ассемблерном коде имеется несколько ошибок. Исправьте эти ошибки.

2.20.2 [10] <2.8> Предположим, что для показанной выше программы вычисления факториала введенное значение было равно 4. Перепишите программу вычисления факториала так, чтобы она работала без использования рекурсивного вызова. Ограничьтесь при использовании регистров диапазоном `$s0–$s7`. Каким будет общее количество инструкций, необходимое для выполнения вашего решения задачи 2.20.2 по сравнению с рекурсивной версией программы вычисления факториала?

2.20.3 [5] <2.8> Покажите содержимое стека после каждого вызова функции при введенном значении, равном 4.

Для следующих задач в таблице содержится фрагмент кода на языке ассемблера, вычисляющий ряд Фибоначчи. Но в записях таблицы имеются ошибки, которые вам придется исправить.

```
a   FIB:  addi  $sp,$sp, -12
          sw    $ra, 0($sp)
          sw    $s1, 4($sp)
          sw    $a0, 8($sp)
          slti   $t0, $a0, 1
          beq   $t0, $0, L1
          addi  $v0,$a0, $0
          j     EXIT
    L1:   addi  $a0,$a0, -1
          jal   FIB
          addi  $s1,$v0, $0
          addi  $a0,$a0, -1
          jal   FIB
          add   $v0, $v0, $s1
    EXIT: lw    $ra, 0($sp)
          lw    $a0, 8($sp)
          lw    $s1, 4($sp)
          addi  $sp, $sp, 12
          jr    $ra
```

```
б   FIB:  addi  $sp,$sp, -12
          sw    $ra, 0($sp)
          sw    $s1, 4($sp)
          sw    $a0, 8($sp)
          slti  $t0  $a0, 1
          beq   $t0, $0, L1
          addi  $v0,$a0, $0
          j     EXIT
    L1:   addi  $a0,$a0, -1
          jal   FIB
          addi  $s1,$v0, $0
          addi  $a0,$a0, -1
          jal   FIB
          add   $v0, $v0, $s1
    EXIT: lw    $ra, 0($sp)
          lw    $a0, 8($sp)
          lw    $s1, 4($sp)
          addi  $sp, $sp, 12
          jr    $ra
```

2.20.4 [5] <2.8> Показанная программа на языке ассемблера MIPS вычисляет ряд Фибоначчи для заданного числа. Это целое число передается в регистре `$a0`, а результат возвращается в регистре `$v0`. В ассемблерном коде имеется несколько ошибок. Исправьте эти ошибки.

2.20.5 [10] <2.8> Предположим, что для показанной программы вычисления ряда Фибоначчи введенное значение равно 4. Перепишите программу вычисления так, чтобы она работала без использования рекурсивного вызова. Ограничьтесь при использовании регистров диапазоном `$s0`–`$s7`. Каким будет общее количество инструкций, используемое для выполнения вашего решения задачи 2.20.5 по сравнению с рекурсивной версией программы вычисления ряда Фибоначчи?

2.20.6 [5] <2.8> Покажите содержимое стека после каждого вызова функции при введенном значении, равном 4.

Упражнение 2.21

Предположим, что стек и сегменты статических данных пусты, а указатель стека и глобальный указатель имеют начальные значения, содержащие соответственно адреса 0x7fff fffc и 0x1000 8000. Предположим, что используется соглашение по вызовам, показанное в табл. 2.5, и что входной параметр передается функции в регистре `$a0`, а ее результат возвращается в регистре `$v0`. Также предположим, что конечные (leaf) функции могут использовать только сохраняемые регистры.

а

```
main()
{
    leaf_function(1);
}
int leaf_function (int f)
{
    int result;
    result = f + 1;
    if (f > 5)
        return result;
    leaf_function(result);
}
```

б

```
int my_global = 100;
main()
{
    int x = 10;
    int y = 20;
    int z;
    z = my_function(x, my_global)
}
int my_function(int x, int y)
{
    return x - y;
}
```

2.21.1 [5] <2.8> Покажите содержимое стека и содержимое сегментов статических данных после каждого вызова функции.

2.21.2 [5] <2.8> Напишите MIPS-код для программ на C в приведенной выше таблице.

2.21.3 [5] <2.8> Напишите MIPS-код для программ на C в приведенной выше таблице, если конечная (leaf) функция может использовать временные регистры (`$t0`, `$t1` и т. д.).

Следующие три задачи этого упражнения касаются функции, написанной на языке ассемблера MIPS в соответствии с соглашением по вызовам, показанным в табл. 2.6:

а

```
f  sub   $s0,$a0,$a3
   sll   $v0,$s0,0x1
   add   $v0,$a2,$v0
   sub   $v0,$v0,$a1
   jr    $ra
```

```
f: addi  $sp,$sp,8
   sw    $ra,4($sp)
   sw    $s0,0($sp)
   move  $s0,$a2
   jal   g
   add   $v0,$v0,$s0
   lw    $ra,4($sp)
   lw    $s0,0($sp)
   addi  $sp,$sp,-8
   jr    $ra
```

2.21.4 [10] <2.8> В этом коде имеется ошибка, нарушающая имеющееся в MIPS соглашение по вызовам. Что это за ошибка и как ее можно исправить?

2.21.5 [10] <2.8> Каким будет эквивалент этого кода на языке C? Предположим, что в C-версии этой функции используются аргументы по имени a, b, c и т. д.

2.21.6 [10] <2.8> В месте вызова этой функции регистры `$a0`, `$a1`, `$a2` и `$a3` имеют соответственно значения 1, 100, 1000 и 30. Каким будет значение, возвращенное функцией? Предположим, что функция g, вызываемая из функции f, всегда возвращает значение 500.

Упражнение 2.22

В этом упражнении исследуется соглашение по ASCII и Unicode. В таблице показаны строки символов.

а	A byte
б	computer

2.22.1 [5] <2.9> Преобразуйте строки в десятичные значения ASCII-байтов.

2.22.2 [5] <2.9> Преобразуйте строки в 16-разрядный Unicode (используя шестнадцатеричную форму записи и набор символов Basic Latin).

В следующей таблице показаны шестнадцатеричные значения ASCII-символов.

а	61 64 64
б	73 68 69 66 74

2.22.3 [5] <2.5, 2.9> Преобразуйте шестнадцатеричные ASCII-значения в текст.

Упражнение 2.23

В этом упражнении вам нужно будет написать программу на языке ассемблера MIPS, преобразующую строки в числовой формат, указанный в таблице.

а	строки из положительных целых десятичных чисел
б	шестнадцатеричные целые числа с дополнением до двух

2.23.1 [10] <2.9> Напишите программу на языке ассемблера MIPS, предназначенную для преобразования строки ASCII-чисел в целое число, соблюдая условия,

перечисленные в приведенной выше таблице. Ваша программа должна предполагать, что в регистре $a0 содержится адрес строки, оканчивающейся нулевым значением и имеющей некоторую комбинацию цифр в диапазоне от 0 до 9. Ваша программа должна вычислить целочисленный эквивалент этой состоящей из цифр строки, а затем поместить число в регистр $v0. Если где-нибудь в строке попадется нецифровой символ, ваша программа должна остановиться и выдать в регистре $v0 значение –1. Например, если регистр $a0 указывает на последовательность из 3 байт: 50_{10}, 52_{10}, 0_{10} (строка «24», завершающаяся нулевым значением), то, когда программа остановится, в регистре $v0 должно содержаться значение 24_{10}.

Упражнение 2.24

Предположим, что регистр $t1 содержит адрес 0x1000 0000, а регистр $t2 содержит адрес 0x1000 0010.

а	`lb $t0, 0($t1)` `sw $t0, 0($t2)`
б	`lb $t0, 0($t1)` `sb $t0, 0($t2)`

2.24.1 [5] <2.9> Предположим, что по адресу 0x1000 0000 содержатся следующие данные (в шестнадцатеричном виде):

1000 0000	12	34	56	78

Какое значение сохранено по адресу, на который указывает регистр $t2? Предположим, что изначально в памяти, указанной $t2, хранилось значение 0xFFFF FFFF.

2.24.2 [5] <2.9> Предположим, что по адресу 0x1000 0000 содержатся следующие данные (в шестнадцатеричном виде):

1000 0000	80	80	80	80

Какое значение сохранено по адресу, на который указывает регистр $t2? Предположим, что изначально в памяти, указанной $t2, хранилось значение 0x0000 0000.

2.24.3 [5] <2.9> Предположим, что по адресу 0x1000 0000 содержатся следующие данные (в шестнадцатеричном виде):

1000 0000	11	00	00	FF

Какое значение сохранено по адресу, на который указывает регистр $t2? Предположим, что изначально в памяти, указанной $t2, хранилось значение 0x5555 5555.

Упражнение 2.25

В этом упражнении будут исследованы используемые в MIPS 32-разрядные константы. Для следующих задач будут использованы двоичные данные, показанные в таблице.

а	1010 1101 0001 0000 0000 0000 0000 0010$_2$
б	1111 1111 1111 1111 1111 1111 1111 1111$_2$

2.25.1 [10] <2.10> Напишите MIPS-код, который создает показанные выше 32-разрядные константы и сохраняет их значение в регистре $t1.

2.25.2 [5] <2.6, 2.10> Можно ли использовать одну инструкцию перехода для получения адреса счетчика команд, показанного в расположенной выше таблице, если текущее значение счетчика команд (PC) равно 0x00000000?

2.25.3 [5] <2.6, 2.10> Можно ли использовать одну инструкцию условного перехода для получения адреса счетчика команд, показанного в расположенной выше таблице, если текущее значение счетчика команд (PC) равно 0x00000600?

2.25.4 [5] <2.6, 2.10> Можно ли использовать одну инструкцию условного перехода для получения адреса счетчика команд, показанного в расположенной выше таблице, если текущее значение счетчика команд (PC) равно 0x00400600?

2.25.5 [10] <2.10> Напишите MIPS-код, создающий 32-разрядные константы и сохраняющий их в регистре $t1, при условии, что поле непосредственного значения (immediate) в MIPS-инструкции стало шириной всего 8 разрядов. При этом нужно обойтись без использования инструкции lui.

Для следующих задач будет использоваться ассемблерный код MIPS, показанный в таблице.

а	lui $t0, 0x1234 ori $t0, $t0, 0x5678
б	ori $t0, $t0, 0x5678 lui $t0, 0x1234

2.25.6 [5] <2.6, 2.10> Каким будет значение регистра $t0 после выполнения показанной выше кодовой последовательности?

2.25.7 [5] <2.6, 2.10> Напишите код на языке C, являющийся эквивалентом показанного в таблице кода на языке ассемблера, при условии, что наибольшая константа, которую можно загрузить в 32-разрядный регистр, должна быть не шире 16 разрядов.

Упражнение 2.26

В этом упражнении будут исследованы диапазоны действия MIPS-инструкций условных и безусловных переходов. В следующих задачах используются шестнадцатеричные данные из таблицы.

а	0x00001000
б	0xFFFC0000

2.26.1 [10] <2.6, 2.10> Сколько инструкций условных переходов (без использования инструкций безусловных переходов) понадобится для получения адреса, показанного в расположенной выше таблице, если счетчик команд (PC) содержит адрес 0x00000000?

2.26.2 [10] <2.6, 2.10> Сколько инструкций безусловных переходов (без использования инструкций переходов по регистру или инструкций условных переходов) понадобится для получения адреса, показанного в расположенной выше таблице, если счетчик команд (PC) содержит адрес 0x00000000?

2.26.3 [10] <2.6, 2.10> В целях сокращения размера MIPS-программ разработчики MIPS решили сократить размер поля непосредственного значения (immediate) в инструкциях I-типа с 16 до 8 разрядов. Сколько инструкций условных переходов понадобится для получения адреса, показанного в расположенной выше таблице, если счетчик команд (PC) содержит адрес 0x00000000?

Для следующих задач будут использоваться модификации, предпринятые в отношении архитектуры набора инструкций MIPS.

а	8 регистров
б	10-разрядное поле непосредственного значения и адреса (immediate и address)

2.26.4 [10] <2.6, 2.10> При изменении набора инструкций MIPS-процессора должен также измениться и формат инструкций. Каким будет для каждого из изменений влияние на диапазон адресов в инструкции beq? Предположим, что все инструкции сохранили длину 32 разряда, и любые изменения, касающиеся формата инструкций I-типа, выражаются только лишь в увеличении или уменьшении поля непосредственного значения (immediate) инструкции beq.

2.26.5 [10] <2.6, 2.10> При изменении набора инструкций MIPS-процессора должен также измениться и формат инструкций. Каким будет для каждого из изменений влияние на диапазон адресов инструкции безусловного перехода? Предположим, что все инструкции сохранили длину 32 разряда, и любые изменения, касающиеся формата инструкций J-типа, отражаются только лишь на поле адреса (address) инструкции безусловного перехода.

2.26.6 [10] <2.6, 2.10> При изменении набора инструкций MIPS-процессора должен также измениться и формат инструкций. Каким будет для каждого из изменений влияние на диапазон адресов инструкции перехода по регистру, при условии, что каждая инструкция должна иметь размер 32 разряда?

Упражнение 2.27

При решении следующих задач будет использоваться исследование различных режимов адресации в архитектуре набора инструкций MIPS. Эти разные режимы адресации перечислены в таблице.

а	Регистровая адресация
б	Относительная адресация по счетчику команд

2.27.1 [5] <2.10> В таблице показаны разные режимы адресации набора инструкций MIPS. Приведите пример MIPS-инструкций, показывающих режим адресации MIPS.

2.27.2 [5] <2.10> Формат какого типа используется для инструкций, названных при выполнении задания 2.27.1?

2.27.3 [5] <2.10> Перечислите все сильные и слабые стороны конкретного режима адресации MIPS. Напишите MIPS-код, показывающий эти плюсы и минусы.

В следующих задачах будет использоваться код ассемблера MIPS, предназначенный для исследования всех сильных и слабых сторон поля immediate в MIPS-инструкциях I-типа:

а	0x00000000 0x00000004	lui ori	$s0, 100 $s0, $s0, 40
б	0x00000100 0x00000104	addi lw	$t0, $0, 0x0000 $t1, 0x4000($t0)

2.27.4 [15] <2.10> Покажите для каждой из MIPS-инструкций ее двоичное представление в виде шестнадцатеричного числа.

2.27.5 [10] <2.10> Сокращая размер полей immediate в инструкциях I- и J-типа, можно сэкономить на числе разрядов, необходимых для представления инструкций. Перепишите показанный выше MIPS-код так, чтобы он отражал следующие изменения: поле immediate инструкций I-типа приобрело размер 8 разрядов, а такое же поле инструкций J-типа стало содержать 18 разрядов. При этом нужно обойтись без использования инструкции lui.

2.27.6 [5] <2.10> Сколько дополнительных MIPS-инструкций понадобится для выполнения вашего кода из задачи 2.27.5 по сравнению с кодом, представленном в таблице?

Упражнение 2.28

Следующий ассемблерный код MIPS предназначен для блокировки.

```
try:  MOV    R3,R4
      MOV    R6,R7
      LL     R2,0(R2)
      LL     R5,0(R1)
      SC     R3,0(R1)
      SC     R6,0(R1)
      BEQZ   R3,try
      MOV    R4,R2
      MOV    R7,R5
```

2.28.1 [5] <2.11> Сколько инструкций должно быть выполнено для каждой проверки и отказа в условном сохранении?

2.28.2 [5] <2.11> Объясните, почему выполнение показанного выше кода связанной загрузки — условного сохранения может закончиться неудачей.

2.28.3 [15] <2.11> Перепишите показанный выше код, чтобы он мог работать правильно. Убедитесь в отсутствии соревновательных условий за доступ к данным.

Каждая запись в следующей таблице содержит код, а также показывает содержимое различных регистров. Обозначение «($s1)» означает содержимое ячейки памяти, на которую указывает значение регистра $s1. Ассемблерный код, показанный в каждой таблице, выполняется в соответствующие тактовые циклы на двух параллельных процессорах с общим пространством памяти.

а

Процессор 1	Процессор 2	Цикл	Процессор 1		Память ($s1)	Процессор 2	
			$t1	$t0		$t1	$t0
		0	1	2	99	30	40
ll $t1, 0($s1)	ll $t1, 0($s1)	1					
sc $t0, 0($s1)		2					
	sc $t0, 0($s1)	3					

б

Процессор 1	Процессор 2	Цикл	Процессор 1			Память ($s1)	Процессор 2		
			$s4	$t1	$t0		$s4	$t1	$t0
		0	2	3	4	99	10	20	30
	try: add $t0, $0, $s4	1							
try: add $t0, $0, $s4	ll $t1, 0($s1)	2							
ll $t1, 0($s1)		3							
sc $t0, 0($s1)		4							
beqz $t0, try	sc $t0, 0($s1)	5							
add $s4, $0, $t1	beqz $t0, try	6							

2.28.4 [5] <2.11> Заполните таблицу значениями регистров для каждого заданного тактового цикла.

Упражнение 2.29

Три первые задачи этого упражнения ссылаются на критический участок кода, имеющий следующую форму:

```
lock(lk);
операция
unlock(lk);
```

где «операция» обновляет общую переменную shvar, используя локальную (не общую) переменную x, как показано ниже:

а	shvar=shvar+x;
б	shvar=min(shvar,x);

2.29.1 [10] <2.11> Напишите для этого критического участка код на языке ассемблера MIPS, при условии, что адрес переменной lk находится в регистре

адрес переменной shvar — находится в регистре $a1, а значение переменной — в регистре $a2. Ваш критический участок не должен содержать никаких вызовов функций, то есть в него нужно включить MIPS-инструкции для операций блокировки — lock(), разблокировки — unlock(), max() и min(). Для реализации операции lock()используйте инструкции ll и sc, а для операции unlock() — простую инструкцию сохранения.

2.29.2 [10] <2.11> Решите задачу 2.29.1 еще раз, но теперь для осуществления непосредственного атомарного обновления переменной shvar, без операций lock() и unlock(), воспользуйтесь инструкциями ll и sc. Обратите внимание на то, что при решении данной задачи переменная lk не используется.

2.29.3 [10] <2.11> Сравните производительность своего кода из задач 2.29.1 и 2.29.2 при наиболее благоприятных условиях, предполагая, что каждая инструкция тратит на свое выполнение один цикл. Примечание: наиболее благоприятные условия означают, что пара инструкций ll — sc, когда нужно выполнить операцию lock(), всегда выполняется успешно, блокировка всегда свободна, и, если осуществляется переход, то мы выбираем путь, завершающий операцию с наименьшим количеством выполняемых инструкций.

2.29.4 [10] <2.11> Используйте в качестве примера ваш код решения задачи 2.29.2 и объясните, что получится, когда два процессора начнут выполнять этот критический участок в одно и то же время, предполагая, что каждый процессор выполняет за один цикл только одну инструкцию.

2.29.5 [10] <2.11> Объясните, почему в вашем коде из задачи 2.29.2 регистр $a1 содержит адрес переменной shvar, а не значение этой переменной, и почему регистр $a2 содержит значение переменной x, а не ее адрес.

2.29.6 [10] <2.11> Если потребуется атомарно выполнить ту же операцию над двумя общими переменными (например, shvar1 и shvar2) на таком же критическом участке, это легко можно будет сделать с использованием подходов из задачи 2.29.1 (просто поместить оба обновления между операцией блокировки и соответствующей ей операцией разблокировки). Объясните, почему мы не можем сделать это, используя подход из задачи 2.29.2., то есть почему мы не можем использовать пару инструкций ll — sc для доступа к обеим общим переменным способом, гарантирующим, что оба обновления будут выполнены вместе, в рамках одной атомарной операции.

Упражнение 2.30

Ассемблерные псевдоинструкции не являются частью набора инструкций MIPS, но довольно часто попадаются в MIPS-программах. Таблица содержит некоторые псевдоинструкции MIPS, которые при работе программы ассемблера транслируются в другие инструкции ассемблера MIPS:

а	move $t1, $t2
б	beq $t1, small, LOOP

2.30.1 [5] <2.12> Для каждой из псевдоинструкций предоставьте минимально возможную последовательность реальных MIPS-инструкций, выполняющих то

же действие. При этом в некоторых случаях может понадобиться использование временных регистров. В таблице large ссылается на число, требующее для своего представления 32 разрядов, а small — на число, которое может поместиться в 16 разрядах.

В таблице содержатся некоторые псевдоинструкции MIPS, которые при работе программы ассемблера транслируются в другие инструкции ассемблера MIPS:

а	`la $s0, v`
б	`blt $a0, $v0, Loop`

2.30.2 [5] <2.12> Нуждаются ли инструкции, показанные в расположенной выше таблице, в редактировании на этапе компоновки? Почему?

Упражнение 2.31

В таблице содержатся подробности, касающиеся компоновки двух разных процедур. В данном упражнении нужно будет выступить в роли программы, занимающейся компоновкой.

а

Процедура А				Процедура Б			
Текстовый сегмент	Адрес	Инструкция		Текстовый сегмент	Адрес	Инструкция	
	0	`lw $a0, 0($gp)`			0	`sw $a1, 0($gp)`	
	4	`jal 0`			4	`jal 0`	
	...				...		
Сегмент данных	0	(X)		Сегмент данных	0	(Y)	
	...				...		
Настр. информация	Адрес	Тип инструкции	Зависим.	Настр. информация	Адрес	Тип инструкции	Зависим.
	0	`lw`	X		0		Y
	4	`jal`	B		4		A
Таблица имен	Адрес	Метка		Таблица имен	Адрес	Метка	
	–	X			–	Y	
	–	B			–	A	

б

Процедура А				Процедура Б			
Текстовый сегмент	Адрес	Инструкция		Текстовый сегмент	Адрес	Инструкция	
	0	`lui $at, 0`			0	`sw $a0, 0($gp)`	
	4	`ori $a0, $at, 0`			4	`jmp 0`	
	8	`jal 0`			...		
	...				0x180	`jr $ra`	
					...		

Процедура А				Процедура Б			
Сегмент данных	0	(X)		Сегмент данных	0	(Y)	
	...				...		
Настр. информация	Адрес	Тип инструкции	Зависим.	Настр. информация	Адрес	Тип инструкции	Зависим.
	0	lui	X		0	sw	Y
	4	ori	X		4	jmp	F00
	8	jal	B				
Таблица имен	Адрес	Метка		Таблица имен	Адрес	Метка	
	–	X			–	Y	
	–	B			0x180	F00	

2.31.1 [5] <2.12> Скомпонуйте показанные выше объектные файлы, чтобы сформировать заголовок исполняемого файла. Предположим, что процедура А имеет размер текстового сегмента 0x140, размер сегмента данных 0x40, а процедура Б имеет размер текстового сегмента 0x300, размер сегмента данных 0x50. Также предположим, что используется стратегия распределения памяти, показанная на рис. 2.7.

2.31.2 [5] <2.12> Какие ограничения, если таковые имеются, накладываются на размер исполняемого файла?

2.31.3 [5] <2.12> Изложите свое понимание ограничений, накладываемых на инструкции условного и безусловного перехода. Почему у ассемблера могут возникнуть проблемы при непосредственной реализации инструкций условного и безусловного перехода в объектном файле?

Упражнение 2.32

При решении первых трех задач этого упражнения предполагается, что функция swap, вместо кода, показанного в листинге 2.1, определена на языке C в следующем виде:

а
```
void swap(int v[], int k, int j){
  int temp;
  temp=v[k];
  v[k]=v[j];
  v[j]=temp;
}
```

б
```
void swap(int *p){
  int temp;
  temp=*p;
  *p=*(p+1);
  *(p+1)=*p;
}
```

2.32.1 [10] <2.13> Преобразуйте эту функцию в ассемблерный код MIPS.

2.32.2 [5] <2.13> Что нужно будет изменить в функции sort?

2.32.3 [5] <2.13> Каким изменениям должен быть подвергнут MIPS-код для swap в задаче 2.32.1, если бы велась сортировка 8-разрядных байтов, а не 32-разрядных слов?

Для остальных трех задач этого упражнения предположим, что функция sort из листинга 2.5 изменена следующим образом:

а	Вместо t-регистров используются s-регистры.
б	В коде с меткой for2tst вместо slt и bne используется инструкция bltz (branch on less than zero — переход, если меньше нуля).

2.32.4 [5] <2.13> Повлияет ли это изменение на код в листинге 2.5, предназначенный для сохранения и восстановления регистров?

2.32.5 [10] <2.13> Насколько больше (или меньше) инструкций будет выполнено в результате этого изменения при сортировке 10-элементного, уже отсортированного массива?

2.32.6 [10] <2.13> Насколько больше (или меньше) инструкций будет выполнено в результате этого изменения при сортировке 10-элементного, уже отсортированного в убывающем порядке (противоположном по сравнению с тем, который создается процедурой sort()) массива?

Упражнение 2.33

Задачи в этом упражнении ссылаются на следующую функцию, заданную в виде кода по обработке массива:

а

```
int find(int a[], int n, int x){
  int i;
  for(i=0;i!=n;i++)
    if(a[i]==x)
      return i;
  return -1;
}
```

б

```
int count(int a[], int n, int x){
  int res=0;
  int i;
  for(i=0;i!=n;i++)
    if(a[i]==x)
      res=res+1;
  return res;
}
```

2.33.1 [10] <2.14> Преобразуйте эту функцию в код на языке ассемблера MIPS.

2.33.2 [10] <2.14> Преобразуйте эту функцию в код на языке C, основанный на использовании указателей.

2.33.3 [10] <2.14> Преобразуйте полученный в результате решения задачи 2.33.2 код на языке C, основанный на использовании указателей, в код на языке ассемблера MIPS.

2.33.4 [5] <2.14> Сравните при самых неблагоприятных условиях количество выполняемых инструкций за итерацию цикла, не являющуюся последней, в созданном вами коде на основе использования массива из задачи 2.33.1 и в созданном вами коде на основе использования указателей из задачи 2.33.3. Примечание: самые неблагоприятные условия наступают, когда по условиям перехода выбираются наи-

длинные последовательности выполняемого кода, то есть если используется инструкция if, результат проверки условия выдается таким, что выбирается путь с большим количеством инструкций. Но если результат проверки условия приведет к завершению цикла, мы будем предполагать, что выбран путь, при котором цикл продолжается.

2.33.5 [5] <2.14> Сравните количество временных регистров (t-регистров), необходимое для кода, основанного на использовании массива из задачи 2.33.1, с количеством, необходимым для кода, основанного на использовании указателей из задачи 2.33.3.

2.33.6 [5] <2.14> Что изменилось бы в вашем ответе на вопрос задачи 2.33.4, если регистры `$t0`–`$t7` и `$a0`–`$a3` в соглашении о вызовах, используемом в MIPS, были бы сохраняемыми при осуществлении вызовов, по аналогии с регистрами `$s0`–`$s7`?

Упражнение 2.34

В показанной ниже таблице содержится ассемблерный код ARM. В следующих задачах нужно будет перевести код ARM в ассемблерный код MIPS.

а	`MOV r0, #10 ;присваивание счетчику цикла начального значения 10`
	`LOOP ADD r0, r1 ;добавление значения r1 к значению r0`
	`SUBS r0, 1 ;уменьшение значения счетчика на единицу`
	`BNE LOOP ;если Z=0, повторение цикла`

б	`ROR r1, r2, #4 ; r1 = r2, объединяется с r2`

2.34.1 [5] <2.16> Преобразуйте этот ассемблерный код ARM в ассемблерный код MIPS, при условии, что ARM-регистры `r0`, `r1` и `r2` содержат те же значения, что и MIPS-регистры `$s0`, `$s1` и `$s2` соответственно. Используйте при необходимости временные регистры (`$t0` и т. д.).

2.34.2 [5] <2.16> Покажите битовые поля, представляющие собой ARM-инструкции для ассемблерных инструкций ARM, показанных в расположенной выше таблице.

Следующая таблица содержит ассемблерный код MIPS. При решении следующих задач нужно будет преобразовать ассемблерный код MIPS в код ARM.

а	`slt $t0, $s0, $s1` `blt $t0, $0, FARAWAY`
б	`add $s0, $s1, $s2`

2.34.3 [5] <2.16> Подберите для содержимого показанной выше таблицы ассемблерный код ARM, соответствующий последовательности ассемблерного кода MIPS.

2.34.4 [5] <2.16> Покажите битовые поля, представляющие собой ассемблерный код ARM.

Упражнение 2.35

У процессора ARM имеется несколько разных режимов адресации, не поддерживаемых в MIPS. В следующих задачах исследуются эти новые режимы адресации.

а	`LDR  r0, [r1]  ; r0 = память[r1]`
б	`LDMIA r0, {r1, r2, r4}  ; r1 = память[r0], r2 = память[r0+4]` `; r4 = память[r0+8]`

2.35.1 [5] <2.16> Определите тип режима адресации ассемблерных инструкций ARM, показанных в расположенной выше таблице.

2.35.2 [5] <2.16> Напишите для показанной выше ассемблерной инструкции ARM последовательность ассемблерных инструкций MIPS для выполнения такого же переноса данных.

В следующих задачах нужно будет сравнить код, написанный с использованием наборов инструкций ARM и MIPS. В таблице показан код, написанный с использованием набора инструкций ARM:

а	`      LDR  r0, =Table1 ;загрузка базового адреса таблицы`
	`      LDR  r1, #100    ;инициализация счетчика цикла`
	`      EOR  r2, r2, r2  ;очистка r2`
	`ADDLP LDR  r4, [r0]    ;получение первого операнда сложения`
	`      ADD  r2, r2, r4  ;сложение с r2`
	`      ADD  r0, r0, #4  ;приращение для получения следующего элемента таблицы`
	`      SUBS r1, r1, #1  ;уменьшение счетчика цикла на единицу`
	`      BNE  ADDLP       ;если счетчик цикла != 0, переход на ADDLP`

б	`      ROR  r1, r2, #4  ; r1 = r2  объединяется с r2`

2.35.3 [10] <2.16> Напишите для показанного выше ассемблерного кода ARM эквивалентную ему процедуру на ассемблерном коде MIPS.

2.35.4 [5] <2.16> Каким будет общее количество ассемблерных инструкций ARM, необходимое для выполнения кода? Каким будет общее количество ассемблерных инструкций MIPS, необходимое для выполнения кода?

2.35.5 [5] <2.16> Предположим, что средний показатель CPI процедуры на ассемблере MIPS имеет значение, равное среднему показателю CPI процедуры на ассемблере ARM, а процессор MIPS имеет рабочую частоту в 1,5 раза выше, чем у процессора ARM, насколько быстрее окажется процессор ARM процессора MIPS?

Упражнение 2.36

У процессора ARM есть интересный способ поддержки непосредственных констант. В этом упражнении исследуются имеющиеся в нем особенности. В таблице содержатся инструкции ARM.

а	`ADD, r3, r2, r1, LSL #3 ;r3=r2+(r1 << 3)`
б	`ADD, r3, r2, r1, ROR #3 ;r3= r2+(r1, прокрученный вправо на 3 разряда)`

2.36.1 [5] <2.16> Напишите эквивалентный код MIPS для показанного выше ассемблерного кода ARM.

2.36.2 [5] <2.16> Перепишите ваш код на ассемблере MIPS, сведя к минимуму количество инструкций, с учетом того условия, что в регистре R1 содержится константа со значением 8.

2.36.3 [5] <2.16> Перепишите ваш код на ассемблере MIPS, сведя к минимуму количество инструкций, с учетом того условия, что в регистре R1 содержится константа со значением 0x06000000.

В следующей таблице содержатся MIPS-инструкции.

a	addi r3, r2, 0x1
б	addi r3, r2, 0x8000

2.36.4 [5] <2.16> Напишите для показанного выше ассемблерного кода MIPS его эквивалент на ассемблере ARM.

Упражнение 2.37

В этом упражнении исследуются различия в наборах инструкций MIPS и x86. В таблице содержится ассемблерный код x86.

а	mov edx, [esi+4*ebx]

б	START: mov ax, 00101100b
	mov cx, 00000011b
	mov bx, 11110000b
	and ax, bx
	or ax, cx

2.37.1 [10] <2.17> Напишите псевдокод для данной процедуры.

2.37.2 [10] <2.17> Каким будет MIPS-эквивалент для данной процедуры?

В таблице содержатся ассемблерные инструкции x86.

а	mov edx, [esi+4*ebx]
б	add eax, 0x12345678

2.37.3 [5] <2.17> Покажите для каждой ассемблерной инструкции размер каждого из битовых полей, представляющих инструкцию.

2.37.4 [10] <2.17> Напишите эквивалентный этим инструкциям код на языке ассемблера MIPS.

Упражнение 2.38

В наборе инструкций x86 имеется префикс REP, вызывающий повторение инструкции заданное количество раз или до тех пор, пока не будет удовлетворено

определенное условие. Первые три задачи этого упражнения ссылаются на следующие инструкции x86:

	Инструкция	Интерпретация
а	REP MOVSB	Повторять, пока ECX не имеет нулевого значения: Mem8[EDI]=Mem8[ESI], EDI=EDI+1, ESI=ESI+1, ECX=ECX-1
б	REP MOVSD	Повторять, пока ECX не имеет нулевого значения: Mem32[EDI]=Mem32[ESI], EDI=EDI+4, ESI=ESI+4, ECX=ECX-1

2.38.1 [5] <2.17> Для чего будет обычно применяться эта инструкция?

2.38.2 [5] <2.17> Напишите MIPS-код, выполняющий такую же операцию, при условии, что `$a0` соответствует `ECX`, `$a1` соответствует `EDI`, `$a2` соответствует `ESI`, а `$a3` соответствует `EAX`.

2.38.3 [5] <2.17> Если инструкция x86 требует один цикл для чтения из памяти, один цикл для записи в память и один цикл для обновления каждого регистра и если MIPS использует один цикл на инструкцию, каким будет выигрыш в скорости от использования этой инструкции x86 вместо эквивалентного ей MIPS-кода, когда `ECX` имеет очень большое значение? Предположим, что время тактового цикла для x86 и MIPS имеет одинаковое значение.

Остальные три задачи в этом упражнении ссылаются на следующую функцию, которая дана как на языке C, так и на языке ассемблера x86. Для каждой инструкции x86 показывается также ее длина в формате инструкции с переменной длиной, а также ее интерпретация (описание того, что она делает). Следует отметить, что в архитектуре x86 по сравнению с MIPS очень мало регистров, поэтому в имеющемся в x86 соглашении по вызовам предусматривается проталкивание всех аргументов в стек. Значение, возвращаемое функцией x86, передается назад вызывающему коду в регистре `EAX`.

	C-код	x86-код

а	`int f(int a, int b){`
	`  return a+b;`
	`}`
		`f:push %ebp  ;1Б. проталкивание %ebp в стек`
		`mov %esp,%ebp  ;2Б. перемещение %esp в %ebp`
		`mov 0xc(%ebp),%eax  ;3Б. загр.2-го арг. в %eax`
		`add 0x8(%ebp),%eax  ;3Б. слож. 1-го арг.с %eax`
		`pop %ebp  ;1Б. восстановление %ebp`
		`ret  ;1Б. возвращение`

б	`void f(int *a, int *b){`
	`  *a=*a+*b;`
	`  *b=*a;`
	`}`
		`f:push %ebp  ;1Б. проталкивание %ebp в стек`
		`mov %esp,%ebp  ;2Б. перемещение %esp в %ebp`
		`mov 8(%ebp),%eax  ;3Б. загр. 1-го арг. в %eax`
		`mov 12(%ebp),%ecx  ;3Б. загр. 2-го арг. в %ecx`
		`mov (%eax),%edx  ;2Б. загр. *a в %edx`
		`add (%ecx),%edx  ;2Б. сложение *b и %edx`
		`mov %edx,(%eax)  ;2Б. сохранение %edx в *a`
		`mov %edx,(%ecx)  ;2Б. сохранение %edx в *b`
		`pop %ebp  ;1Б. восстановление %ebp`
		`ret  ;1Б. возвращение`

2.38.4 [5] <2.17> Преобразуйте эту функцию в код на языке ассемблера MIPS. Сравните размер (сколько байтов памяти займут инструкции) для этого кода x86 и вашего MIPS-кода.

2.38.5 [5] <2.17> Если за один цикл процессор может выполнить две инструкции, то он способен по крайней мере прочитать за один цикл две последовательные инструкции. Объясните, как это было бы сделано в MIPS и в x86.

2.38.6 [5] <2.17> Если каждая MIPS-инструкция занимает один цикл, и если каждая инструкция x86 занимает один цикл плюс цикл для каждого необходимого чтения из памяти или записи в нее, то каким будет выигрыш во времени от использования x86 вместо MIPS? Предположим, что время тактового цикла одинаковое как для x86, так и для MIPS и выполнение функции занимает наикратчайший путь (то есть выход из каждого цикла происходит немедленно и каждая инструкция if выбирает направление, приводящее к выходу из функции). Следует заметить, что инструкция ret в x86 читает адрес возврата из стека.

Упражнение 2.39

В таблице приведены показатели CPI для инструкций разных типов.

	Арифметические	Загрузки-сохранения	Перехода
a	2	10	3
c	1	10	4

2.39.1 [5] <2.18> Предположим, что для выполнения заданных программ дано разбиение по инструкциям:

	Инструкций (в миллионах)
Арифметические	500
Загрузки-сохранения	300
Перехода	100

Каким будет время выполнения для процессора, если рабочая частота равна 3 ГГц?

2.39.2 [5] <2.18> Предположим, что набор инструкций был пополнен новыми, более мощными арифметическими инструкциями. В среднем за счет использования этих более мощных арифметических инструкций появилась возможность сократить на 25% количество арифметических инструкций, необходимых для выполнения программ, но при этом пришлось поплатиться всего лишь 10% увеличением времени тактового цикла. Можно ли считать удачным такой конструкторский выбор? Почему?

2.39.3 [5] <2.18> Предположим, что был найден способ удвоения производительности арифметических инструкций. Каков общий выигрыш в скорости для вашей машины? А что будет, если найдется способ увеличить производительность арифметических инструкций в 10 раз?

В таблице показаны пропорции выполнения инструкций разного типа.

	Арифметические	Загрузки-сохранения	Перехода
a	60%	20%	20%
б	80%	15%	5%

2.39.4 [5] <2.18> Найдите для показанного выше сочетания разных типов инструкций средний показатель CPI, если для арифметической инструкции требуется два цикла, для инструкции загрузки-сохранения 6 циклов, а для инструкции условного перехода 3 цикла.

2.39.5 [5] <2.18> Сколько в среднем может прийтись циклов на арифметические инструкции при 25% увеличении производительности, при условии, что производительность инструкций загрузки-сохранения и перехода увеличена не будет?

2.39.6 [5] <2.18> Сколько в среднем может прийтись циклов на арифметические инструкции при 50% увеличении производительности, при условии, что производительность инструкций загрузки-сохранения и перехода увеличена не будет?

Упражнение 2.40

Первые три задачи этого упражнения ссылаются на функцию, показанную на языке ассемблера MIPS. К сожалению, программист, создававший эту функцию, пребывал в заблуждении, полагая, что MIPS относится к машинам с пословной, а не с побайтовой адресацией.

а

```
. int f(int a[], int n, int x);
f: move $v0,$zero    ; ret=0
   move $t0,$zero    ; i=0
L  add  $t1,$t0,$a0  ; &(a[i])
   lw   $t1,0($t1)   ; чтение a[i]
   bne  $t1,$a2,S    ; если (a[i]==x)
   addi $v0,$v0,1    ;   ret++;
S: addi $t0,$t0,1    ; i++
   bne  $t0,$a1,L    ; повторение, если i!=n
   jr   $ra          ; возвращение ret
```

б

```
: void f(int *a, int *b, int n);
f  move $t0,$a0      ; p=a
   move $t1,$a1      ; q=b
   add  $t2,$a2,$a0  ; &(a[n])
L  lw   $t3,0($t0)   ; чтение *p
   lw   $t4,0($t1)   ; чтение *q
   add  $t3,$t3,$t4  ; *p+*q
   sw   $t3,0($t0)   ; *p=*p+*q
   addi $t0,$t0,1    ; p=p+1
   addi $t1,$t1,1    ; q=q+1
   bne  $t0,$t2,L    ; повторение, если p!=&(a[n])
   jr   $ra          ; возвращение
```

Учтите, что в ассемблере MIPS символ «;» показывает, что остальная часть строки является комментарием.

2.40.1 [5] <2.18> Архитектура MIPS требует, чтобы обращения, имеющие размер слова (`lw` и `sw`), были выровнены по границам слов, то есть два самых нижних разряда адреса должны иметь нулевые значения. Если адрес не выровнен по границе слова, процессор выдает исключение «ошибка шины» («bus error»). Объясните, как эти требования по выравниванию влияют на выполнение данной функции.

2.40.2 [5] <2.18> Будет ли эта функция работать, если «a» будет указателем на начало массива, состоящего из однобайтовых элементов, а инструкции `lw` и `sw` будут соответственно заменены инструкциями `lb` (загрузить байт) и `sb` (сохранить байт)?

Примечание: инструкция lb считывает байт из памяти, проводит расширение знака и помещает результат в регистр назначения, а инструкция sb сохраняет младший байт регистра в памяти.

2.40.3 [5] <2.18> Внесите в код изменения, позволяющие ему работать с 32-разрядными целыми числами.

Остальные три задачи в этом упражнении ссылаются на программу, которая выделяет память под массив, заполняет этот массив числами, вызывает функцию sort, показанную в листинге 2.5, а затем выводит содержимое массива. Функция main этой программы имеет следующий вид (на языке C и на языке ассемблера MIPS):

Код функции main на языке C	MIPS-версия функции main

main(){
  int *v;
  int n=5;
  v=my_alloc(5);
  my_init(v,n);
sort(v,n);
	main:
	li $s0,5
	move $a0,$s0
	jal my_alloc
	move $s1,$v0
	move $a0,$s1
	move $a1,$s0
	jal my_init
	move $a0,$s1
	move $a1,$s0
	jal sort

Функция my_alloc имеет следующее определение (на языке C и на языке ассемблера MIPS). Учтите, что программист, создававший эту функцию, пребывал в заблуждении относительно использования указателя на автоматическую переменную arr за пределами функции, в которой она была определена.

my_alloc на языке C	MIPS-код для my_alloc

int *my_alloc(int n){
  int arr[n];
  return arr;
}
	my_alloc:
	  addu $sp, $sp,-4 ; Проталкивание
	  sw $fp, 0($sp) ; $fp в стек
	  move $fp, $sp ; Сохр $sp в $fp
	  sll $t0, $a0,2 ; Нужны 4*n байт
	  sub $sp, $sp,$t0 ; Созд места для arr
	  move $v0, $sp ; Возвр адреса arr
	  move $sp, $fp ; Восст. $sp из $fp
	  lw $fp, 0(sp) ; Выталкивание $fp
	  addiu $sp, $sp,4 ; из стека
	  jr ra

Функция my_init имеет следующее определение (в MIPS-коде):

```
a       my_init:
            move   $t0,$zero       ; i=0
            move   $t1,$a0
        L:  sw     $zero,0($t1)    ; v[i]=0
            addiu  $t1,$t1,4
            addiu  $t0,$t0,1       ; i=i+1
            bne    $t0,$a1,L         пока не будет i==n
            jr     $ra
```

```
б       my_init:
            move    $t0,$zero       ; i=0
            move    $t1,$a0
        L:  sub     $t2,$a1,$t0
            sw      $t2,0($t1)      ; a[i]=n-i
            addiu   $t1,$t1,4
            addiu   $t0,$t0,1       ; i=i+1
            bne     $t0,$a1,L       ; пока не будет i==n
            jr      $ra
```

2.40.4 [5] <2.18> Каково содержимое (значения всех пяти элементов) массива v непосредственно перед инструкцией «jal sort» при выполнении кода функции main?

2.40.5 [15] <2.18, 2.13> Каково содержимое массива v непосредственно перед тем, как функция sort в первый раз войдет в свой внешний цикл? Предположим, что в начале кода функции main (непосредственно перед выполнением инструкции «li $s0, 5»), регистры $sp, $s0, $s1, $s2 и $s3 имеют соответственно значения 0x1000, 20, 40, 7 и 1.

2.40.6 [10] <2.18, 2.13> Каково содержимое пятиэлементного массива, на который указывает v, сразу же после возвращения управления функции main, следующего за выполнением инструкции «jal sort»?

Ответы на вопросы для самопроверки

Раздел 2.2: MIPS, C, Java

Раздел 2.3: 2) как очень низкие

Раздел 2.4: 3) -8_{10}

Раздел 2.5: 4) sub $s2, $s0, $s1

Раздел 2.6: Обе. Операция И, использующая маску из единиц, обнулит все разряды, кроме требуемого поля. Сдвиг влево на нужное количество разрядов удаляет разряды слева от поля. Сдвиг вправо на нужное количество разрядов смещает поле в самые правые разряды слова, оставляя нули в остальных разрядах слова. Учтите, что операция И оставляет поле там, где оно находилось изначально, а пара смещений перемещает поле в самую правую часть слова.

Раздел 2.7: I. Справедливы все утверждения. II. 1).

Раздел 2.8: Справедливы оба утверждения.

Раздел 2.9: I. 2). II. 3)

Раздел 2.10: I. 4) ±128K. II. 6) внутри блока из 256M адресов III. 4) sll.

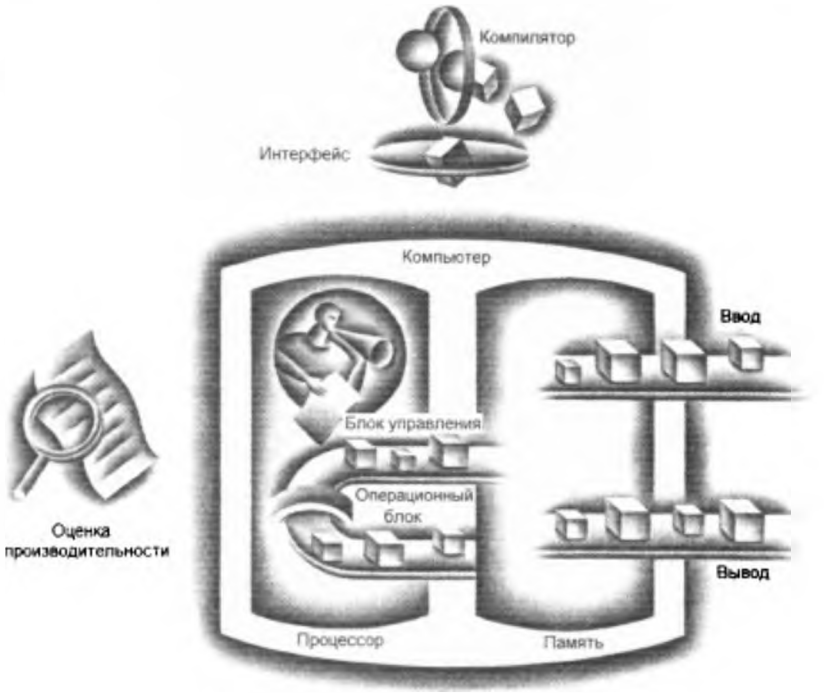
Раздел 2.11: В обоих случаях.

Раздел 2.12: 4) независимость от машины.

Глава 3

Арифметика для компьютеров

Истинной душой науки является точность чисел.
Сэр Дарси Вентворт Томпсон



3.1. Введение

Компьютерный мир состоит из битов, следовательно, слова могут быть представлены в виде двоичных чисел. В главе 2 показывается, что целые числа могут быть

представлены как в десятичной, так и в двоичной форме, а как насчет других часто встречающихся чисел? Например:

- Как работать с дробями и другими вещественными числами?
- Что происходит, когда в результате операции создается число больше того, которое может быть представлено?
- А в основе этих вопросов лежит еще одна тайна: как оборудование в действительности умножает и делит числа?

Целью этой главы является раскрытие всех этих тайн, включая представление вещественных чисел, арифметические алгоритмы, оборудование, следующее этим алгоритмам, и вовлечение во все это набора инструкций. Проникновение в суть этих вопросов может объяснить те странности, с которыми вы уже столкнулись при работе с компьютерами.

3.2. Сложение и вычитание

Вычитание — это хитрый приятель сложения.
Дэвид Литтерман

Сложение является именно тем процессом, выполнение которого можно ожидать от компьютера. Цифры складываются поразрядно справа налево, с переносом на следующую цифру слева, как это делается при счете вручную. Вычитание использует сложение: соответствующий операнд просто меняет свой знак перед сложением.

Упражнение

Двоичное сложение и вычитание
Попробуйте сложить 6_{10} и 7_{10} в двоичном виде, а затем в том же виде вычесть 6_{10} из 7_{10}.

Ответ

```
   0000 0000 0000 0000 0000 0000 0000 01112 = 7ten
 + 0000 0000 0000 0000 0000 0000 0000 01102 = 6ten
-------------------------------------------------------
 = 0000 0000 0000 0000 0000 0000 0000 11012 = 13ten
```

Все действие производится с четырьмя разрядами справа. На рис. 3.1 продемонстрированы суммирования и переносы. Переносы записаны в круглых скобках со стрелками, показывающими, как они передаются.

Вычитание 6_{10} из 7_{10} может быть произведено непосредственно:

```
   0000 0000 0000 0000 0000 0000 0000 01112 = 7ten
 - 0000 0000 0000 0000 0000 0000 0000 01102 = 6ten
-------------------------------------------------------
 = 0000 0000 0000 0000 0000 0000 0000 00012 = 1ten
```

или посредством сложения, используя представление числа −6 в виде дополнения до двух:

```
   0000 0000 0000 0000 0000 0000 0000 01112 = 7ten
 + 1111 1111 1111 1111 1111 1111 1111 10102 = -6ten
-------------------------------------------------------
 = 0000 0000 0000 0000 0000 0000 0000 00012 = 1ten
```

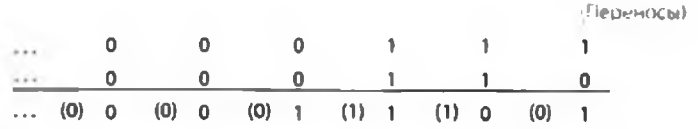


Рис. 3.1. Двоичное сложение, показывающее переносы справа налево. В самом правом разряде складываются 1 и 0, в результате чего получается сумма 1, и перенос из этого разряда будет 0. Следовательно, операция для второго разряда справа будет 0+1+1. В результате для этого разряда суммы получится 0 с переносом 1. Третьей цифрой станет сумма 1+1+1, дающая перенос 1 и значение разряда суммы 1. Четвертый разряд будет получен в результате операции 1+0+0, что даст в сумме 1 без переноса

Вспомним, что переполнение возникает в том случае, когда результат операции не может быть представлен на доступном оборудовании, в данном случае — в 32-разрядном слове. Когда может произойти переполнение при сложении? При сложении операндов с разными знаками переполнение произойти не может. Причина в том, что сумма не может быть больше одного из операндов. Например, $-10 + 4 = -6$. Поскольку операнды помещаются в 32 разряда и сумма не превышает размер операнда, сумма также должна помещаться в 32 разряда. Поэтому при сложении положительного и отрицательного операндов переполнение произойти не может.

Аналогичные ограничения на возникновение переполнения имеются и при вычитании, но по противоположному принципу: когда знаки у операндов *одинаковые*, переполнение произойти не может. Вспомним, что $x - y = x + (-y)$, поскольку вычитание ведется путем смены знака у второго операнда и последующего сложения. Стало быть, при вычитании с использованием операндов с одинаковыми знаками все сводится к *сложению* операндов с *разными* знаками. Из предыдущего абзаца известно, что переполнение в данном случае также не может произойти.

Это хорошо, что мы знаем о том, когда не может произойти переполнение при сложении и вычитании, но как определить, когда оно *может* произойти? Вполне очевидно, что сложение или вычитание двух 32-разрядных чисел может привести к результату, для полного выражения которого потребуется 33 разряда. Отсутствие 33-го разряда означает, что при возникновении переполнения знаковый разряд заполняется *значением* результата, а не соответствующим ему знаком. Поскольку нам нужен всего один дополнительный разряд, неверное значение будет иметь только знаковый разряд. Следовательно, переполнение происходит, когда при сложении двух положительных чисел сумма является отрицательным числом, или наоборот. Это означает, что происходит перенос в знаковый разряд.

Переполнение при вычитании происходит, когда отрицательное число вычитается из положительного и дает отрицательный результат или когда положительное число вычитается из отрицательного и дает положительный результат. Это означает, что происходит заем из знакового разряда. Комбинации операций, операндов и результатов, свидетельствующих о переполнении, показаны в табл. 3.1.

Таблица 3.1. Условия переполнения для сложения и вычитания

Операция	Операнд A	Операнд B	Результат, свидетельствующий о переполнении
A + B	≥ 0	≥ 0	< 0
A + B	< 0	< 0	≥ 0
A – B	≥ 0	< 0	< 0
A – B	< 0	≥ 0	≥ 0

Рассмотренные только что способы определения переполнения касались использования в компьютере чисел с дополнением до двух. А как быть с переполнением при использовании беззнаковых целых чисел? Эти числа обычно используются для адресации памяти, где переполнение игнорируется.

Поэтому разработчики компьютеров должны предоставить способ выборочного игнорирования и распознавания переполнения. В MIPS решение заключается в наличии двух видов арифметических инструкций, позволяющих распознать эти два варианта:

- Сложение (`add`) и непосредственное сложение (`addi`), а также вычитание (`sub`) вызывают исключения при переполнении.
- Беззнаковое сложение (`addu`), непосредственное беззнаковое сложение (`addiu`) и беззнаковое вычитание (`subu`) *не вызывают* исключений при переполнении.

Поскольку в C переполнения игнорируются, MIPS C-компиляторы будут всегда генерировать беззнаковые версии арифметических инструкций `addu`, `addiu` и `subu`, независимо от типа переменных. А вот MIPS Фортран-компиляторы выбирают соответствующие арифметические инструкции в зависимости от типа операндов.

Оборудование, выполняющее сложение и вычитание, называется **арифметико-логическим устройством (АЛУ)**.

Интерфейс аппаратного и программного обеспечения

Разработчики компьютеров должны решить, как обрабатывать арифметические переполнения. Хотя некоторые языки, к которым относятся C и Java, игнорируют целочисленное переполнение, другие языки, например Ада и Фортран, требуют, чтобы программа получила соответствующую информацию. Затем программист или программное окружение должны принять решение о том, что делать при возникновении переполнения.

MIPS определяет переполнение с помощью **исключения** (exception), также называемого на многих компьютерах **прерыванием** (interrupt). Исключение, или прерывание, по сути представляет собой незапланированный вызов процедуры.

Арифметико-логическое устройство (АЛУ)
Оборудование, которое выполняет сложение и вычитание, а также, обычно, логические операции, например И и ИЛИ.

Исключение
Также называется прерыванием. Незапланированное событие, прерывающее выполнение программы; используется для определения переполнения.

Прерывание
Исключение, возникающее за пределами процессора. (В описаниях некоторых архитектур для всех исключений используется термин «прерывание»).

Адрес инструкции, вызвавшей переполнение, сохраняется в регистре, а компьютер переходит на предопределенный адрес для вызова соответствующей подпрограммы для обработки этого исключения. (В разделе 4.9 исключения рассматриваются более подробно, а в главах 5 и 6 описываются другие ситуации, при которых происходят исключения и прерывания.)

Для хранения адреса инструкции, вызвавшей исключение, в MIPS имеется регистр, называемый *счетчиком команд исключения* — exception program counter (EPC). Чтобы у программного обеспечения MIPS была возможность возврата к проблемной инструкции с помощью перехода по регистру, для копирования EPC в регистр общего назначения используется инструкция *перемещения из системного управления* — move from system control (`mfc0`).

Арифметика для мультимедиа

Поскольку каждый микропроцессор настольной системы по определению имеет свой собственный графический дисплей, по мере увеличения возможностей по размещению транзисторов стало неизбежным добавление поддержки графических операций.

Многие графические системы изначально использовали восьмибитное представление каждого из трех основных цветов плюс 8 бит на указание места пиксела. Добавление громкоговорителей и микрофонов для телеконференций и видеоигр стало причиной поддержки звука. Аудиофрагментам нужна была более высокая точность, чем та, которую обеспечивали 8 бит, но 16 бит было вполне достаточно.

У каждого микропроцессора имеются специальные средства, обеспечивающие этим байтам и полусловам возможность занимать при хранении меньше места в памяти (см. раздел 2.9). Но из-за редкости арифметических операций с такими размерами данных в обычных целочисленных программах они, кроме переноса данных, имеют весьма незначительную поддержку. Конструкторы определили, что многие графические и аудиоприложения будут осуществлять подобные операции с векторами этих данных. Путем выделения удлиненных цепочек в 64-разрядном сумматоре процессор может выполнять одновременно операции над короткими векторами из восьми 8-разрядных операндов, из четырех 16-разрядных операндов или из двух 32-разрядных операндов. Стоимость таких разделенных сумматоров невелика. Эти расширения для выполнения одной инструкции над множественными данными были названы векторными, или SIMD-, расширениями (single instruction, multiple data) (см. раздел 2.16 и главу 7).

В микропроцессорах общего назначения обычно отсутствует одно свойство — операция *насыщения*. Насыщение означает, что при возникновении переполнения при вычислении результат устанавливается в виде наибольшего положительного числа или наименьшего отрицательного числа, а не в виде вычисления по модулю, как в арифметике с дополнением до двух. Насыщение наиболее подходит для медиа-операций. Например, ручка громкости в радиоприемнике будет работать неправильно, если при вращении звук будет некоторое время постепенно нарастать, а затем мгновенно станет очень тихим. Ручка с насыщением остановится

на крайнем верхнем значении, неважно, насколько градусов ее повернут дальше. В табл. 3.2 показаны арифметические и логические операции, имеющиеся во многих мультимедийных расширениях современных наборов инструкций.

Таблица 3.2. Сводные данные поддержки мультимедиа для настольных компьютеров

Категория инструкций	Операнды
Беззнаковое сложение и вычитание	Восемь 8-разрядных или четыре 16-разрядных
Насыщающее сложение и вычитание	Восемь 8-разрядных или четыре 16-разрядных
Нахождение максимума и минимума	Восемь 8-разрядных или четыре 16-разрядных
Нахождение среднего	Восемь 8-разрядных или четыре 16-разрядных
Сдвиг вправо и влево	Восемь 8-разрядных или четыре 16-разрядных

Уточнение. MIPS может перехватывать переполнение, но, в отличие от многих других компьютеров, в этой системе отсутствует условный переход, тестирующий переполнение. Обнаружить переполнение может последовательность MIPS-инструкций. Для сложения со знаком используется следующая последовательность (см. *Уточнение* для инструкции `xor` в разделе 2.6):

```
addu $t0, $t1, $t2           # $t0 = сумме, но перехвата не происходит2
xor  $t3, $t1, $t2           # Проверка на различие знаков
slt  $t3, $t3, $zero         # $t3 = 1, если знаки отличаются
bne  $t3, $zero, No_overflow # у $t1 и $t2 знаки ≠, значит, нет переполнения
xor  $t3, $t0, $t1           # знаки =; знак суммы также им соответствует?
                             # $t3 отрицательный, если знак суммы отличается
slt  $t3, $t3, $zero         # $t3 = 1, если знак суммы отличается
bne  $t3, $zero, Overflow    # все три знака =: переход к обработке переполнения
```

Для беззнакового сложения (`$t0 = $t1 + $t2`), тест будет следующим:

```
addu $t0, $t1, $t2          # $t0 = сумме
nor  $t3, $t1, $zero        # $t3 = НЕ $t1
                            # (дополнение до двух - 1  2^32 - $t1 - 1)
sltu $t3, $t3, $t2          # (2^32 - $t1 - 1) < $t2
                            # ⇒ 2^32 - 1 < $t1 + $t2
bne $t3,$zero, Overflow     # если (2^32-1<$t1+$t2), переход к обработке переполнения
```

Краткие выводы

Основное внимание в этом разделе уделялось вопросу о том, что, независимо от представления, небесконечный размер компьютерного слова означает, что результаты арифметических операций могут быть слишком большими для этого фиксированного размера. Обнаружить переполнение для беззнаковых чисел нетрудно, хотя оно практически всегда игнорируется, потому что программы не требуют его обнаружения для адресной арифметики, где чаще всего используются натуральные числа. Сложнее обстоят дела с числами с дополнением до двух, к тому же некоторые программные системы требуют обнаружения переполнения, поэтому сегодня все компьютеры имеют способы его обнаружения.

Рост популярности мультимедийных приложений приводит к появлению арифметических инструкций, поддерживающих более узкие по формату данных операции, которые довольно просто осуществлять в параллельном режиме.

Самопроверка

Некоторые языки программирования допускают арифметические действия с двоичным дополнением над переменными, объявленными в виде байта или полуслова. Какие MIPS-инструкции можно было бы для этого использовать?

1. Загрузку с помощью `lbu`, `lhu`; арифметику с помощью `add`, `sub`, `mult`, `div`; а затем сохранение с помощью `sb`, `sh`.
2. Загрузку с помощью `lb`, `lh`; арифметику с помощью `add`, `sub`, `mult`, `div`; а затем сохранение с помощью `sb`, `sh`.
3. Загрузку с помощью `lb`, `lh`; арифметику с помощью `add`, `sub`, `mult`, `div`, использование И (AND) для наложения маски на результат на 8 или 16 разрядов после каждой операции; затем сохранение с помощью `sb`, `sh`.

Уточнение. Ранее уже сообщалось, что `EPC` копируется в регистр с помощью инструкции `mfc0`, а затем управление возвращается прерванному коду с помощью перехода по регистру. В связи с этим возникает интересный вопрос: поскольку сначала нужно перенести `EPC` в регистр, используемый в переходе по регистру, как может этот переход по регистру вернуть управление прерванному коду *и* восстановить исходные значения всех регистров? Либо сначала нужно восстановить все старые значения регистров и, таким образом, уничтожить адрес возврата в `EPC`, который был помещен в регистр для использования при переходе по регистру, либо восстановить значения всех регистров, кроме одного, содержащего адрес возврата, чтобы можно было осуществить переход, а это значит, что исключение приведет к изменению этого одного регистра в любой момент выполнения программы! Ни один из этих вариантов нельзя признать удовлетворительным.

Чтобы избавить оборудование от такой дилеммы, MIPS-программисты согласились зарезервировать регистры `$k0` и `$k1` для нужд операционной системы; значения этих регистров при возникновении исключений *не* восстанавливаются. Точно так же как MIPS-компиляторы избегают использования регистра `$at`, чтобы ассемблер мог использовать его в качестве временного (см. подраздел «Интерфейс аппаратного и программного обеспечения» в разделе 2.10), они также воздерживаются от использования регистров `$k0` и `$k1`, чтобы они были доступны операционной системе. Подпрограммы обработки исключений помещают адрес возврата в один из этих регистров, а затем используют переход по регистру для восстановления адреса инструкции.

Уточнение. Скорость сложения увеличивается за счет более быстрого определения переноса в верхние разряды. Множество схем ожидания переноса в наихудшем случае представляется функцией $\log_2$ от числа разрядов в сумматоре. Эти сигналы ожидания проходят быстрее, поскольку их путь лежит через меньшее количество элементов в цепочке, но, чтобы предположить нужный перенос, задействуется намного больше элементов. Наиболее популярной стала схема *предвидения переноса* (carry lookahead).

3.3. Умножение

> Умножение меня раздражает, деление вообще не
> нранится, правило трех действий ставит в тупик
> а практика просто сводит с ума.
>
> *Неизвестный автор*

Завершив объяснение сложения и вычитания, мы готовы к построению такой еще более обременительной операции, как умножение.

Сначала посмотрим на обычное письменное умножение десятичных чисел, чтобы вспомнить все его этапы и названия операндов. По причинам, которые вскоре прояснятся, мы ограничиваем этот десятичный пример использованием цифр 0 и 1. Умножение 1000_{10} на 1001_{10}:

```
Множимое        1000₁₀
Множитель     × 1001₁₀
              ----------
                 1000
                0000
               0000
              1000
              ----------
Произведение  1001000₁₀
```

Первый операнд называется *множимым*, а второй — *множителем*. Окончательный результат называется *произведением*. Как вы, наверное, помните, алгоритм, изученный в начальной школе, предполагает поочередное использование цифр множителя справа налево, умножение множимого на одну из цифр множителя и смещение промежуточных произведений на одну цифру влево по отношению к ранее полученному промежуточному произведению.

Первое, на что нужно обратить внимание, — это то, что количество цифр в произведении существенно превышает их количество как в множимом, так и в множителе. Фактически, если проигнорировать знаковые разряды, в результате перемножения n-разрядного множимого и m-разрядного множителя получается произведение, имеющее $n + m$ разрядов. То есть для отображения всех возможных произведений потребуется $n + m$ разрядов. Следовательно, как и при сложении, при умножении нужно уметь справляться с переполнением, поскольку зачастую в качестве результата перемножения двух 32-разрядных чисел нужно получить 32-разрядное произведение.

В данном примере мы ограничились десятичными цифрами 0 и 1. При наличии только двух возможных вариантов каждый этап умножения сводится к простой задаче:

- простому помещению копии множимого (1 × множимое) в нужное место, если цифра множителя представлена единицей, или
- помещению 0 (0 × множимое) в нужное место, если цифра множителя представлена нулем.

Хотя в показанном выше десятичном примере намеренно используются только 0 и 1, при умножении двоичных чисел всегда должны использоваться только 0 и 1, и поэтому всегда предлагаются только эти два варианта.

Традиционно следующим шагом после рассмотрения основ умножения является предоставление высокооптимизированного оборудования для умножения. Мы не станем следовать этой традиции, полагая, что вы получите более глубокое понимание вопроса, рассматривая развитие умножающего оборудования и алгоритма в нескольких поколениях. А пока давайте предположим, что мы умножаем только положительные числа.

Последовательная версия алгоритма и оборудования умножения

Эта конструкция имитирует тот алгоритм, который изучался в школе; оборудование показано на рис. 3.2. Оборудование изображено таким образом, что поток данных направлен сверху вниз.

Предположим, что множитель находится в 32-разрядном регистре Множителя, а 64-разрядный регистр Произведения имеет нулевое начальное значение. Показанный выше пример операции умножения с помощью карандаша и бумаги дает понять, что нам нужно будет при каждом шаге перемещать множимое на одну цифру влево для получения промежуточных произведений. За 32 шага 32-разрядное множимое будет перемещено на 32 разряда влево. Следовательно, нам нужен 64-разрядный регистр Множимого, имеющий в качестве начального значения 32-разрядное множимое в правой половине и нули в левой половине. Затем этот регистр с каждым шагом будет смещаться на один разряд влево для выравнивания множимого с суммой, аккумулируемой в 64-разрядном регистре Произведения

Рис. 3.2. Первая версия умножающего оборудования. Регистр Множимого, Арифметико-логическое устройство (АЛУ) и регистр Произведения имеют величину 64 разряда, и только регистр Множителя состоит из 32 разрядов. 32-разрядное множимое изначально помещается в правую половину регистра Множимого и с каждым шагом сдвигается влево на один разряд. Множитель с каждым шагом сдвигается в обратном направлении. Алгоритм начинает работу с нулевым значением произведения. Блок управления решает, когда сдвигать содержимое регистров Множимого и Множителя и когда записывать новое значение в регистр Произведения

На рис. 3.3 показаны три основных шага, необходимых для каждого разряда. Наименьший значащий разряд множителя (Множитель0) определяет, добавлено ли множимое к регистру Произведения. Сдвиг влево во втором шаге вызывает перемещение промежуточных операндов влево, точно так же как при умножении с помощью карандаша и бумаги. Сдвиг вправо в третьем шаге дает нам следующий бит множителя для тестирования следующей итерации. Эти три шага повторяются 32 раза для получения произведения. Если для каждого шага требуется тактовый цикл, этот алгоритм для умножения двух 32-разрядных чисел потребует почти 100 тактовых циклов. Относительная важность таких арифметических операций, как умножение, варьируется от программы к программе, но сложение и вычитание могут быть в разных случаях от пяти до ста раз более востребованными, чем умножение. Соответственно во многих приложениях умножение может потребовать большого количества тактовых циклов без существенного влияния на производительность. Тем не менее закон Амдала (см. раздел 1.8) напоминает нам, что даже весьма умеренная частота использования медленных операций может ограничить производительность.

Этот алгоритм и оборудование довольно просто усовершенствовать, добившись одного тактового цикла на один шаг. Ускорение возникает за счет параллельного выполнения операций: Множитель и Множимое сдвигаются в тот самый момент, когда Множимое добавляется к Произведению, если разряд Множимого равен 1. Остается только убедиться в том, что отслежен сдвиг вправо Множителя, и получить предварительно прошедшую сдвиг версию Множимого. Обычно оборудование проходит еще более глубокую оптимизацию, нацеленную на то, чтобы поделить на два ширину сумматора и регистров, определяя, где находится неиспользуемая часть регистров и сумматоров. Видоизмененное оборудование показано на рис. 3.4.

Интерфейс аппаратного и программного обеспечения

Замена арифметических операций сдвигом может также использоваться при умножении с использованием констант. Некоторые компиляторы заменяют операции умножения короткими константами с серией сдвигов и сложений. Поскольку сдвиг на один разряд влево представляет вдвое большее число по основанию два, сдвиг разрядов влево имеет тот же эффект, что и умножение на два. Как уже упоминалось в главе 2, практически каждый компилятор будет оптимизировать код, упрощая его путем подстановки сдвига влево для умножения на два.

Упражнение

Алгоритм умножения

Используя для экономии пространства 4-разрядные числа, умножьте $2_{10} \times 3_{10}$, или $0010_2 \times 0011_2$.

Ответ

В рис. 3.5 показано значение каждого регистра для каждого шага, обозначенного в соответствии с рис. 3.3, с конечным значением 0000 0110_2 или 6_{10}. Чтобы показать значения регистра, изменившиеся при выполнении конкретного шага, используется жирный шрифт.

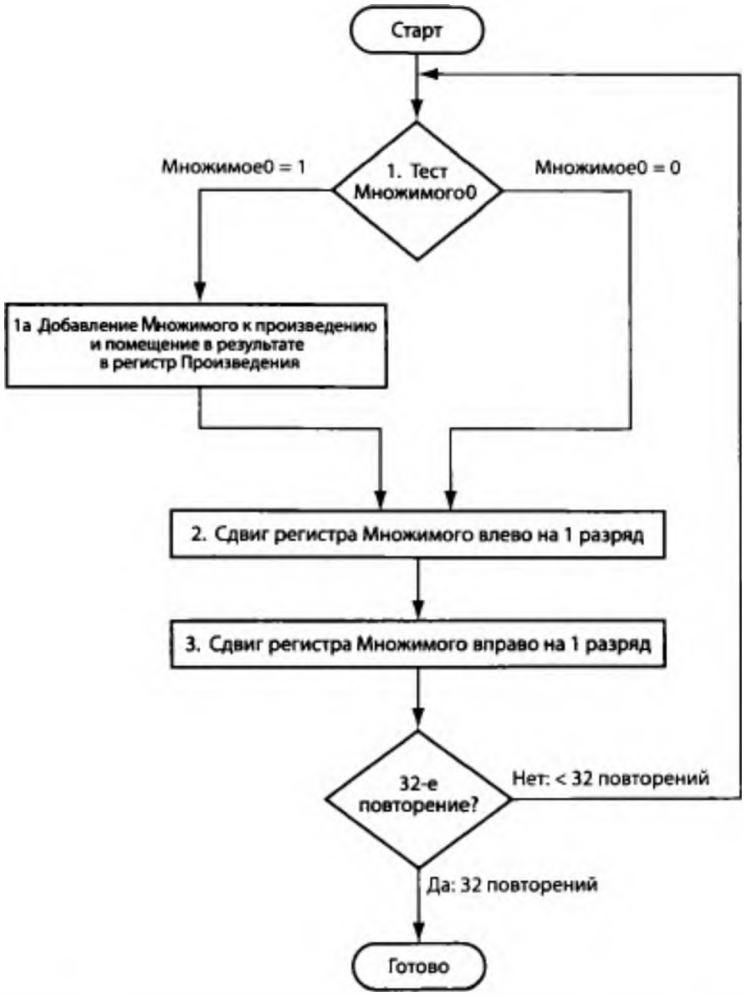


Рис. 3.3. Первый алгоритм умножения, использующий оборудование, показанное на рис. 3.2. Если наименьший значимый разряд Множителя равен единице, Множимое следует прибавить к Произведению. Если нет, следует перейти к следующему шагу. В следующих двух шагах осуществляется сдвиг Множимого влево и Множителя вправо. И эти три шага повторяются 32 раза

Рис. 3.4. Усовершенствованная версия умножающего оборудования. Сравните ее с первой версией на рис. 3.2. Регистр Множимого, АЛУ и регистр Множителя имеют величину 32 разряда, и только регистр Произведения остался 64-разрядным. Теперь Произведение сдвигается вправо. Также упразднен отдельный регистр Множителя. Множитель помещен на место правой половины регистра Произведения. Эти изменения на рисунке выделены серым цветом. (На самом деле регистр Произведения должен иметь 65 разрядов, чтобы содержать перенос сумматора, но здесь он показан 64-разрядным, чтобы выделить изменения по сравнению с рис. 3.2)

Итерация	Шаг	Множитель	Множимое	Произведение
0	Исходные значения	001①	0000 0010	0000 0000
1	1a: 1 => Произв. = Произв. + Множим.	0011	0000 0010	**0000 0010**
	2: Сдвиг Множимого влево	0011	**0000 0100**	0000 0010
	3: Сдвиг Множителя вправо	000①	0000 0100	0000 0010
2	1a: 1 => Произв. = Произв. + Множим.	0001	0000 0100	**0000 0110**
	2: Сдвиг Множимого влево	0001	**0000 1000**	0000 0110
	3: Сдвиг Множителя вправо	000⓪	0000 1000	0000 0110
3	1: 0 => Пустая операция	0000	0000 1000	0000 0110
	2: Сдвиг Множимого влево	0000	0001 0000	0000 0110
	3: Сдвиг Множителя вправо	000⓪	0001 0000	0000 0110
4	1: 0 => Пустая операция	0000	0001 0000	0000 0110
	2: Сдвиг Множимого влево	0000	**0010 0000**	0000 0110
	3: Сдвиг Множителя вправо	**0000**	0010 0000	0000 0110

Рис. 3.5. Пример умножения с использованием алгоритма, показанного на рис. 3.3. Бит, проверяемый для определения следующего шага, выделен жирным шрифтом

Умножение чисел со знаками

До сих пор мы имели дело с положительными числами. Самый простой способ понять, как нужно работать с числами со знаком, заключается в предварительном преобразовании их в положительные числа с последующим восстановлением исходных знаков. Затем алгоритм должен запустить 31 итерацию, выведя знаки за пределы вычислений. Из курса средней школы нам известно, что изменять знак произведения нужно только в том случае, когда у сомножителей разные знаки.

Получается, что последний алгоритм будет работать для чисел со знаками при условии, что мы помним, что имеем дело с числами, имеющими бесконечное количество цифр, и только лишь представляем их, используя 32-разрядный формат. Следовательно, для расширения знака произведения для чисел со знаками потребуются шаги сдвига. Когда действия согласно алгоритму будут завершены, слово в самых младших разрядах будет содержать 32-разрядное произведение.

Ускоренное умножение

Закон Мура предопределил настолько высокий рост ресурсов, что разработчики оборудования теперь могут создавать более быстрые аппаратные средства, используемые для умножения. Будет ли множимое складываться или нет, известно в начале умножения за счет изучения каждого из 32 разрядов. Ускоренное умножение возможно путем фактического предоставления одного 32-разрядного сумматора для каждого бита Множителя: одним входящим значением является Множимое, обработанное с помощью операции И с битом Множителя, а другим — выходное значение предыдущего сумматора.

Вполне очевидно, что выходы тех сумматоров, которые находятся справа, должны подключаться ко входам левых сумматоров, создавая последовательность из 32 сумматоров. Альтернативным способом организации этих 32 сумматоров является параллельное дерево, похожее на то, что показано на рис. 3.6. Вместо ожидания 32 сложений приходится ждать только $\log_2(32)$ или пяти 32-разрядных сложений. На рис. 3.6 показано, что это наиболее быстрый способ подключения сумматоров.

В действительности умножение может происходить еще быстрее, чем за пять сложений, за счет использования *сумматоров с сохранением переноса* и за счет простоты конвейеризации такой конструкции, способной поддержать множество одновременно проводимых операций умножения (см. главу 4).

Умножение в MIPS

В MIPS для 64-разрядного произведения предоставляется отдельная пара 32-разрядных регистров с именами `Hi` и `Lo`. Для получения произведения с нужным знаком или без знака в MIPS имеются две инструкции: умножения (`mult`) и беззнакового умножения (`multu`). Для извлечения целочисленного 32-разрядного произведения программисты используют инструкцию *перемещение из lo* — move from lo (`mflo`). В ассемблере MIPS для умножения имеется псевдоинструкция, при выполнении которой определяются три регистра общего назначения и генерируются инструкции `mflo` и `mfhi` для помещения произведения в регистры.

Краткие выводы

Умножающее оборудование занимается простым сдвигом и сложением, позаимствованным из методики умножения с помощью карандаша и бумаги, которой учат в школе. Компиляторы даже используют инструкции сдвига для умножения на 2^n.

Интерфейс аппаратного и программного обеспечения

Обе имеющиеся в MIPS инструкции умножения игнорируют переполнение, поэтому проверка, не превосходит ли произведение 32-разрядный размер, возлагается на программное обеспечение. Переполнения не произошло, если `Hi` имеет значение 0 для `multu` или повторяет знак `Lo` для `mult`. Для переноса из `Hi` в регистр общего назначения с целью проверки переполнения может быть применена инструкция *перемещение из hi* — move from hi (`mfhi`).

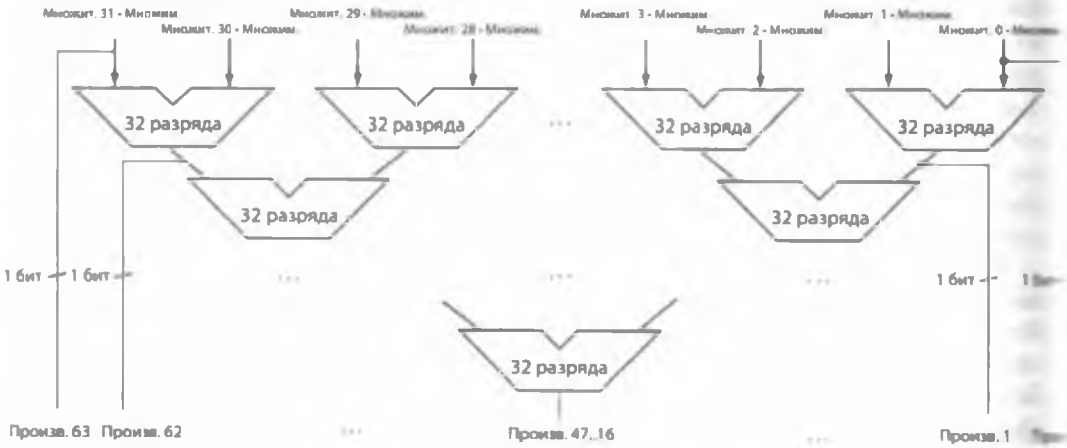


Рис. 3.6. Оборудование для ускоренного умножения. Вместо 31-кратного использования одного 32-разрядного сумматора это оборудование «разворачивает цикл» для использования 31 сумматора, а затем выстраивает их так, чтобы свести задержки к минимуму

3.4. Деление

Divide et impera[1].

Операцией, обратной умножению, является деление, которое встречается реже и требует для своей реализации несколько большей изобретательности. При ее ре-

[1] Разделяй и властвуй (*лат.*).

операции возникает даже возможность выполнения математически недопустимой операции: деления на нуль.

Чтобы вспомнить названия операндов и алгоритм деления, который преподается в школе, начнем с примера деления в столбик с использованием десятичных чисел. Из тех же соображений простоты, которые использовались и в предыдущем разделе, ограничимся десятичными цифрами 0 или 1. В используемом примере число $1\ 001\ 010_{10}$ будет разделено на 1000_{10}:

	1001_{10}	Частное
Делитель 1000_{10}	$\overline{)1001010_{10}}$	Делимое
	-1000	
	10	
	101	
	1010	
	-1000	
	10_{10}	Остаток

Два операнда деления, называемые **Делимым** и **Делителем**, и результат, называемый **Частным**, сопровождаются вторым результатом, который называется **Остатком**. Рассмотрим еще один способ выражения взаимоотношений между всеми составляющими:

Делимое = Частное × Делитель + Остаток,

где остаток меньше делителя. Изредка программы используют инструкцию деления для получения остатка, игнорируя при этом частное.

Алгоритм, которому учат в школе, представляет собой попытки определения, какое наибольшее число может быть вычтено, при этом при каждой попытке выбирается цифра частного. Наш тщательно подобранный десятичный пример использует только лишь числа 0 и 1, упрощающие определение того, сколько раз делитель составляет часть от делимого: это либо нуль, либо один раз. Двоичные числа состоят только из нулей или единиц, поэтому двоичное деление сводится только к этим двум вариантам, упрощая тем самым операцию деления.

Предположим, что как делимое, так и делитель являются положительными числами, а стало быть, частное и остаток также являются числами неотрицательными. Операнды деления и результаты являются 32-разрядными значениями, и, в данном случае, знаки чисел будут игнорироваться.

Делимое
Число, подвергаемое делению.

Делитель
Число, на которое делится делимое.

Частное
Первичный результат деления; число, которое при умножении на делитель и прибавлении к полученному произведению остатка превращается в делимое.

Остаток
Вторичный результат деления; число, которое при прибавлении к произведению частного и делителя дает делимое.

Алгоритм и оборудование, используемые для деления

На рис. 3.7 показано оборудование, предназначенное для имитации школьного алгоритма. Начнем с 32-разрядного регистра Частного, значение которого установлено в нуль. Каждая итерация алгоритма требует перемещения делителя вправо на одну цифру, поэтому мы начнем с делителя, расположенного в левой половине 64-разрядного регистра Делителя и с каждым шагом будем смещать его вправо на один разряд, выравнивая с Делимым. Регистр Остатка в качестве начального получает значение Делимого.

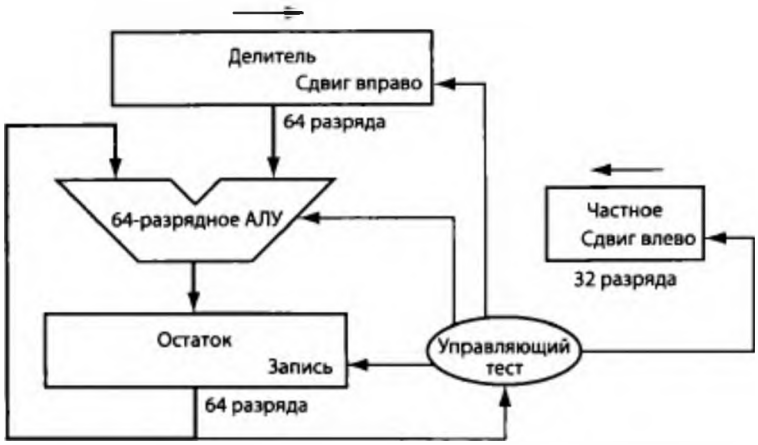


Рис. 3.7. Первая версия оборудования, предназначенного для деления. Регистр Делителя АЛУ и регистр Остатка имеют величину 64 разряда, и только регистр Частного имеет величину 32 разряда. 32-разрядный делитель изначально располагается в левой половине регистра Делителя, и с каждой итерацией он смещается на один разряд вправо. Остаток изначально получает значение делимого. Блок управления решает, когда нужно произвести сдвиг содержимого регистров Делимого и Частного и когда произвести запись нового значения в регистр Остатка

На рис. 3.8 показаны три шага первого алгоритма деления. В отличие от человека компьютер недостаточно умен, чтобы знать наперед, является ли делитель меньше делимого. Сначала он должен вычесть делитель в шаге 1; следует вспомнить, что точно так же проводится сравнение в инструкции «установка, меньше чем». Если результат положительный, значит, делитель был меньше или равен делимому, поэтому в частном генерируется единица (шаг 2а). Если результат отрицательный, следующий шаг заключается в восстановлении исходного значения путем обратного сложения делителя с остатком и генерации нуля в частном (шаг 2б). Делитель сдвигается вправо, а затем происходит новая итерация. После того как итерации завершатся, остаток и частное будут найдены в регистрах с соответствующими именами.

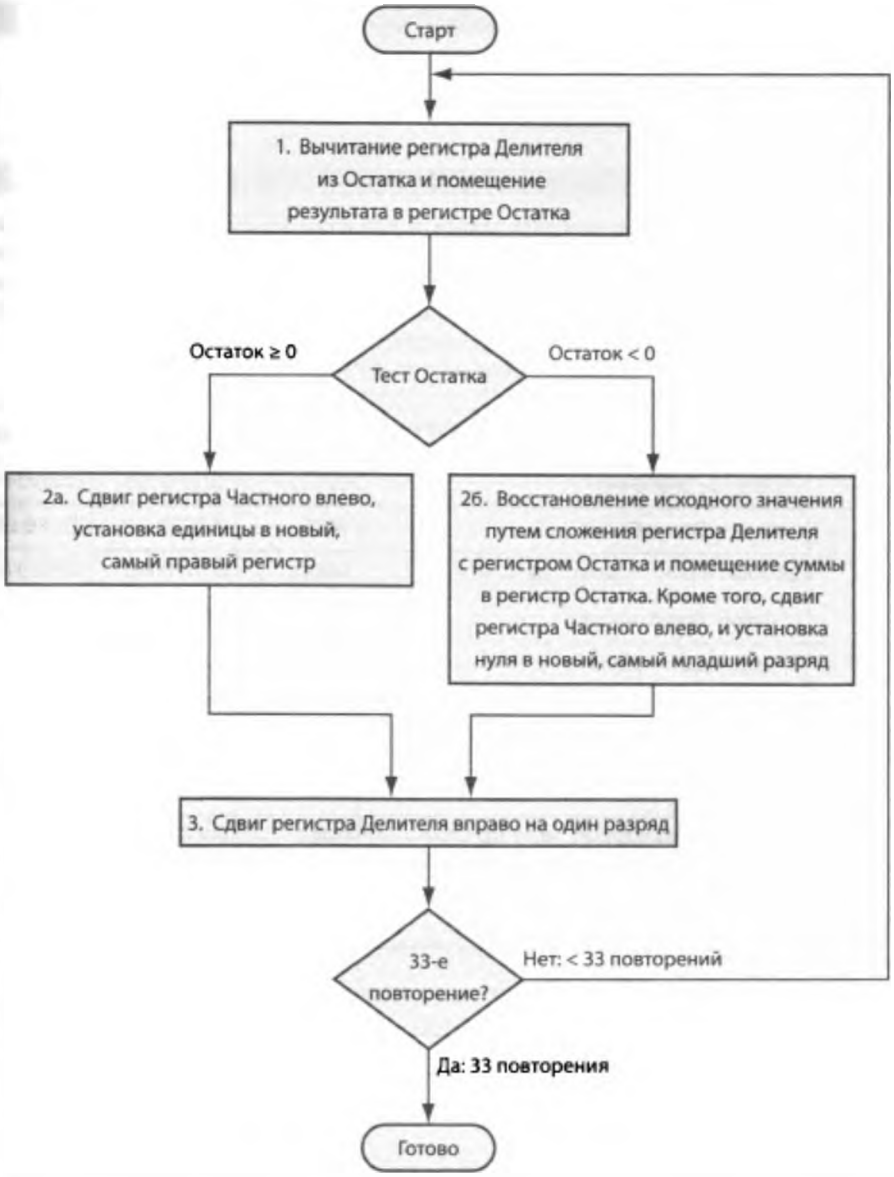


Рис. 3.8. Алгоритм деления, использующий оборудование, показанное на рис. 3.7. Если остаток имеет положительное значение, значит, делитель действительно являлся частью делимого, поэтому шаг 2а генерирует в частном 1. Отрицательный остаток после шага 1 означает, что делитель не входил в делимое, поэтому шаг 2б генерирует в частном 0 и складывает делитель с остатком, проводя, таким образом, действие, обратное вычитанию в шаге 1. Заключительный сдвиг в шаге 3 нужным образом выравнивает делитель по отношению к делимому для следующей итерации. Эти шаги повторяются 33 раза

Упражнение

Алгоритм деления

Используя для экономии пространства 4-разрядную версию алгоритма, попробуйте разделить 7_{10} на 2_{10}, или $0000\ 0111_2$ на 0010_2.

Ответ

На рис. 3.9 показано значение каждого регистра для каждого шага с частным, равным 3_{10}, и остатком, равным 1_{10}. Обратите внимание на то, что тест на шаге 2, проверяющий, является ли остаток положительным или отрицательным, просто выясняет, содержит ли знаковый разряд регистра Остатка 0 или 1. Неожиданным требованием этого алгоритма является использование $n + 1$ итераций для получения нужного частного и остатка.

Итерация	Шаг	Частное	Делитель	Остаток
0	Исходные значения	0000	0010 0000	0000 0111
1	1: Ост. = Ост. – Делит.	0000	0010 0000	**①110 0111**
	2б: Ост. < 0 => + Делит., sll Ч, Ч0 = 0	**0000**	0010 0000	0000 0111
	3: Сдвиг Делит. вправо	0000	**00010000**	0000 0111
2	1: Ост. = Ост. – Делит.	0000	0001 0000	**①111 0111**
	2b: Ост. < 0 => +Делит., sll Ч, Ч0 = 0	**0000**	0001 0000	0000 0111
	3: Сдвиг Делит. вправо	0000	**0000 1000**	0000 0111
3	1: Ост. = Ост. – Делит.	0000	0000 1000	**①111 1111**
	2b: Ост. < 0 => +Делит., sll Ч, Ч0 = 0	**0000**	0000 1000	0000 0111
	3: Сдвиг Делит. вправо	0000	**0000 0100**	0000 0111
4	1: Ост. = Ост. – Делит.	0000	0000 0100	**0000 0011**
	2a: Ост. > 0 => sll Ч, Ч0 = 1	0001	0000 0100	0000 0011
	3: Сдвиг Делит. вправо	0001	0000 0010	0000 0011
5	1: Ост. = Ост. – Делит.	0001	0000 0010	**0000 0001**
	2a: Ост. > 0 => sll Ч, Ч0 = 1	**0011**	0000 0010	0000 0001
	3: Сдвиг Делит. вправо	0011	**0000 0001**	0000 0001

Рис. 3.9. Пример деления, использующего алгоритм, показанный на рис. 3.8. Разряд, проверяемый для определения следующего шага, выделен жирным шрифтом

Чтобы ускорить и удешевить эту операцию, алгоритм и оборудование можно усовершенствовать. Ускорения можно добиться за счет того, что сдвиг операндов и частного будет осуществляться одновременно с вычитанием. Это усовершенствование вдвое уменьшает ширину сумматора и регистров за счет обнаружения неиспользуемых частей регистров и сумматоров. Модифицированное оборудование показано на рис. 3.10.

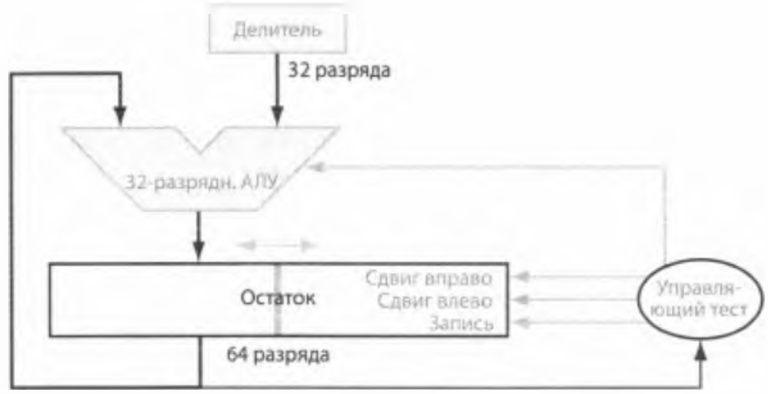


Рис. 3.10. Усовершенствованная версия оборудования, предназначенного для деления. Регистр Делителя, АЛУ и регистр Частного имеют величину 32 разряда, и только у регистра Остатка остается величина 64 разряда. По сравнению с оборудованием, показанным на рис. 3.7, АЛУ и регистр Делителя стали вдвое меньше, а остаток сдвигается влево. В этой версии регистр Частного также объединен с правой половиной регистра Остатка. (По аналогии с изображением на рис. 3.4, в действительности регистр Остатка должен иметь 65 разрядов, чтобы обеспечить сохранность переноса из сумматора.)

Деление чисел со знаком

До сих пор знак при делении чисел игнорировался. Наиболее простым решением для деления со знаком будет запоминание знаков делителя и делимого с последующей сменой знака у частного, если первые два знака отличаются друг от друга.

Уточнение. Одна из сложностей деления со знаком заключается в необходимости устанавливать также и знак остатка. Вспомним, что в любом случае должно выполняться следующее уравнение:

$$\text{Делимое} = \text{Частное} \times \text{Делитель} + \text{Остаток}$$

Чтобы понять, как устанавливается знак остатка, давайте рассмотрим пример деления всех комбинаций $\pm 7_{10}$ на $\pm 2_{10}$. Первый вариант самый простой:

$$+7 \times +2\text{: Частное} = +3, \text{Остаток} = +1.$$

Проверим результаты:

$$7 = 3 \times 2 + (+1) = 6 + 1.$$

Если изменить знак делимого, то должно также измениться и частное:

$$-7 \div +2\text{: Частное} = -3.$$

Перепишем основную формулу для вычисления остатка:

$$\text{Остаток} = (\text{Делимое} - \text{Частное} \times \text{Делитель}) = -7 - (-3 \times +2) = -7 - (-6) = -1$$

Итак,

$$-7 \div +2\text{: Частное} = -3, \text{Остаток} = -1.$$

Еще раз проверим результаты:

$$-7 = -3 \times 2 + (-1) = -6 - 1.$$

Ответ не может состоять из частного –4 и остатка +1, что также удовлетворяло бы данной формуле, потому что абсолютное значение частного тогда бы изменялось в зависимости от знака делимого и делителя! Если

$$-(x \div y) \neq (-x) \div y,$$

то программирование осложнилось бы еще больше. Такого аномального поведения можно избежать, если следовать правилу, что делимое и остаток должны иметь одинаковые знаки, независимо от того, какие знаки у делителя и частного.

Остальные комбинации будут вычислены согласно тому же правилу:

$$+7 \div -2\text{: Частное} = -3, \text{Остаток} = +1;$$

$$-7 \div -2\text{: Частное} = +3, \text{Остаток} = -1.$$

Такое деление с правильной расстановкой знаков меняет знак частного, если знаки операндов противоположны друг другу, и присваивает ненулевому остатку знак делимого.

Ускоренное деление

Для ускорения умножения использовалось множество сумматоров, но применить подобный прием для деления невозможно. Причина в том, что перед выполнением следующего шага алгоритма нужно знать о знаке разницы, а при умножении 32 частичных произведения могут быть вычислены немедленно.

Существуют технологии, позволяющие за один шаг получать более одного разряда частного. Технология *SRT-деления* пытается за один шаг выстроить предположение относительно нескольких разрядов частного, используя табличный поиск, основанный на старших разрядах делителя и остатка. Для исправления неверных предположений она полагается на следующие шаги. На сегодняшний день типичным значением являются 4 разряда. Ключевым в этой технологии выступает предположение о вычитаемом значении. В двоичном делении есть только один приемлемый вариант. Для индексирования таблицы, определяющей предположение для каждого шага, в этих алгоритмах используются 6 разрядов остатка и 4 разряда делителя.

Точность этого ускоренного метода зависит от наличия в таблице поиска подходящих значений. То, что может произойти, если в таблице содержатся неверные значения, показано в разделе 3.8, где рассматриваются все заблуждения и недоразумения.

Деление в MIPS

При изучении рис. 3.4 и 3.10 можно было заметить, что для умножения и деления подходит оборудование, состоящее из одних и тех же логических схем. Единственное, что для этого требуется, — наличие 64-разрядного регистра со сдвигом вправ

влево и 32-разрядного АЛУ, способного производить сложение и вычитание. В соответствии с этим как для умножения, так и для деления в MIPS используется 32-разрядный регистр Hi и 32-разрядный регистр Lo. При изучении показанного выше алгоритма можно было предположить, что после выполнения инструкции деления в Hi содержится остаток, а в Lo — частное.

Чтобы работать с целыми числами со знаком и без знака, в MIPS имеется две инструкции: *деления* (div) и *беззнакового деления* (divu). Язык ассемблера MIPS позволяет указывать инструкциям деления три регистра, генерируя инструкции mflo или mfhi для помещения желаемых результатов в регистры общего назначения.

Краткие выводы

Общая аппаратная поддержка умножения и деления позволяет MIPS предоставлять отдельную пару 32-разрядных регистров, которые используются как для умножения, так и для деления. В табл. 3.3 представлены все дополнительные инструкции MIPS-архитектуры, имеющие отношение к двум последним разделам.

Интерфейс аппаратного и программного обеспечения

Имеющиеся в MIPS инструкции деления игнорируют переполнение, поэтому определение превышения размеров частного возлагается на программное обеспечение. Кроме переполнения, при делении может также иметь место недопустимое вычисление: деление на нуль. Некоторые компьютеры отслеживают эти ненормальные случаи. Чтобы обнаружить и деление на нуль, и переполнение, программное обеспечение MIPS должно проверять делитель.

Уточнение. Еще более быстрый алгоритм не производит немедленного обратного сложения делителя при отрицательном остатке. Он просто *прибавляет* в следующем шаге делимое к сдвинутому остатку. Этот алгоритм деления *без восстановления* затрачивает один тактовый цикл на шаг и исследуется в последующих упражнениях. А рассмотренный здесь алгоритм называется делением *с восстановлением*. Третий алгоритм, не сохраняющий результат вычитания, если он отрицательный, называется *бездействующим* (nonperforming) алгоритмом деления. В среднем он позволяет сэкономить треть арифметических операций.

Таблица 3.3. Основная архитектура MIPS. В целях экономии места память и регистры MIPS-архитектуры не приводятся, но в этом разделе для поддержки умножения и деления добавлены регистры Hi и Lo

Язык ассемблера MIPS

Категория инструкций	Инструкции	Пример	Значение	Комментарии
Арифметические	add (сложение)	`add $s1,$s2,$s3`	`$s1 = $s2 + $s3`	Три операнда; переполнение определяется
	subtract (вычитание)	`sub $s1,$s2,$s3`	`$s1 = $s2 - $s3`	Три операнда; переполнение определяется
	add immediate (непосредственное сложение)	`addi $s1,$s2,20`	`$s1 = $s2 + 100`	+ константа; переполнение определяется
	add unsigned (беззнаковое сложение)	`addu $s1 $s2,$s3`	`$s1 = $s2 + $s3`	Три операнда; переполнение не определяется
	subtract unsigned (беззнаковое вычитание)	`subu $s1,$s2,$s3`	`$s1 = $s2 - $s3`	Три операнда; переполнение не определяется
	add immediate unsigned (беззнаковое непосредственное сложение)	`addiu $s1,$s2,20`	`$s1 = $s2 + 100`	+ константа; переполнение не определяется
	move from coprocessor register (перемещение из регистра сопроцессора)	`mfc0 $s1,$epc`	`$s1 = $epc`	Копирование счетчика команд исключения (EPC) и специальных регистров
	multiply (умножение)	`mult $s2,$s3`	`Hi, Lo = $s2 × $s3`	64-разрядное произведение со знаком в Hi, Lo
	multiply unsigned (беззнаковое умножение)	`multu $s2,$s3`	`Hi, Lo = $s2 × $s3`	64-разрядное произведение без знака в Hi, Lo
	divide (деление)	`div $s2,$s3`	`Lo = $s2 / $s3,` `Hi = $s2 mod $s3`	Lo = частное, Hi = остаток
	divide unsigned (беззнаковое деление)	`divu $s2,$s3`	`Lo = $s2 / $s3,` `Hi = $s2 mod $s3`	Беззнаковое частное и остаток

Категория инструкций	Инструкции	Пример	Значение	Комментарии
	move from Hi (перемещение из Hi)	mfhi $s1	$s1 = Hi	Используется для получения копии Hi
	move from Lo (перемещение из Lo)	mflo $s1	$s1 = Lo	Используется для получения копии Lo
Переноса данных	load word (загрузка слова)	lw $s1,20($s2)	$s1 = Память[$s2 + 20]	Слово из памяти в регистр
	store word (сохранение слова)	sw $s1,20($s2)	Память[$s2 + 20] = $s1	Слово из регистра в память
	load half unsigned (загрузка полуслова без знака)	lhu $s1,20($s2)	$s1 = Память[$s2 + 20]	Полуслово из памяти в регистр
	store half (сохранение полуслова)	sh $s1,20($s2)	Память[$s2 + 20] = $s1	Полуслово из регистра в память
	load byte unsigned (загрузка байта без знака)	lbu $s1,20($s2)	$s1 = Память[$s2 + 20]	Байт из памяти в регистр
	store byte (сохранение байта)	sb $s1,20($s2)	Память[$s2 + 20] = $s1	Байт из регистра в память
	load linked word (связанная загрузка слова)	ll $s1,20($s2)	$s1 = Память[$s2 + 20]	Загрузка слова при выполнении первой половины атомарного обмена
	store condition word (условное сохранение слова)	sc $s1,20($s2)	Память[$s2+20]=$s1; $s1=0 или 1	Сохранение слова при выполнении второй половины атомарного обмена
	load upper immediate (загрузка непосредственного значения в верхние разряды)	lui $s1,20	$s1 = 20 * 216	Загрузка константы в верхние 16 разрядов

продолжение ⇩

Таблица 3.3 (*продолжение*)

Категория инструкций	Инструкции	Пример	Значение	Комментарии
Логические	AND (И)	and $s1,$s2,$s3	$s1 = $s2 & $s3	Трехрегистровые операнды; поразрядное И
	OR (ИЛИ)	or $s1,$s2,$s3	$s1 = $s2 | $s3	Трехрегистровые операнды; поразрядное ИЛИ
	NOR (НЕ ИЛИ)	nor $s1,$s2,$s3	$s1 = ~ ($s2 | $s3)	Трехрегистровые операнды, поразрядное НЕ ИЛИ
	and immediate (непосредственное И)	andi $s1,$s2,100	$s1 = $s2 & 100	Поразрядное И с константой
	or immediate (непосредственное ИЛИ)	ori $s1,$s2,100	$s1 = $s2 | 100	Поразрядное ИЛИ с константой
	shift left logical (логический сдвиг влево)	sll $s1,$s2,10	$s1 = $s2 << 10	Сдвиг влево на константу
	shift right logical (логический сдвиг вправо)	srl $s1,$s2,10	$s1 = $s2 >> 10	Сдвиг вправо на константу
Условного перехода	branch on equal (переход при равенстве)	beq $s1,$s2,25	если ($s1 = = $s2), перейти на PC + 4 + 100	Тест на равенство; переход, связанный со счетчиком команд
	branch on not equal (переход при неравенстве)	bne $s1,$s2,25	если ($s1!= $s2), перейти на PC + 4 + 100	Тест на неравенство; переход, связанный со счетчиком команд
	set on less than (установка, если меньше чем)	slt $s1,$s2,$s3	если ($s2 < $s3), $s1 = 1; иначе $s1 = 0	Сравнение «меньше чем»; с дополнением до двух
	set less than immediate (установка, если меньше чем непосредственное значение)	slti $s1,$s2,20	если ($s2 < 20), $s1 = 1; иначе $s1 = 0	Сравнение «меньше чем»; с дополнением до двух

Категория инструкций	Инструкции	Пример	Значение	Комментарии
	set on less than unsigned (установка, если меньше, чем без знака)	sltu $s1,$s2,$s3	если ($s2 < $s3), $s1 = 1; иначе $s1 = 0	Сравнение «меньше чем»: натуральные числа
	set less than immediate unsigned (установка, если меньше, чем непосредственное значение без знака)	sltiu $s1,$s2,100	если ($s2 < 100), $s1 = 1; иначе $s1 = 0	Сравнение «меньше чем» с константой, натуральные числа
Безусловного перехода	jump (переход)	j 2500	перейти на 10000	Переход по указанному адресу
	jump register (переход по регистру)	jr $ra	перейти на адрес, указанный в $ra	Для переключения; возвращение из процедуры
	jump and link (переход и ссылка)	jal 2500	$ra = PC + 4; перейти на 10000	Для вызова процедуры

3.5. Числа с плавающей точкой

Скорость ничего не даст, если двигаться в неверном направлении.

Американская поговорка

Выходя за рамки целых чисел со знаком и без знака, языки программирования поддерживают дробные числа, которые в математике называются *действительными*. Рассмотрим несколько примеров действительных чисел:

$3{,}14159265..._{10}$ («пи»);

$2{,}71828..._{10}$ («е»);

$0{,}000000001_{10}$, или $1{,}0_{10} \times 10^{-9}$ (количество секунд в наносекунде);

$3\ 155\ 760\ 000_{10}$, или $3{,}15576_{10} \times 10^{9}$ (количество секунд в обычном столетии).

Обратите внимание на то, что в последнем примере число не представляло собой небольшую дробь, но было слишком большим для того, чтобы быть представленным 32-разрядным целым числом со знаком. Альтернативная форма записи для последних двух чисел, у которой слева от десятичного знака имеется всего одна цифра, называется **экспоненциальным представлением**. Число в экспоненциальном представлении, не имеющее лидирующих нулей, называется **нормализованным числом** и является обычной формой записи. Например, $1{,}0_{10} \times 10^{-9}$ относится к нормализованному экспоненциальному представлению, а $0{,}1_{10} \times 10^{-8}$ и $10{,}0_{10} \times 10^{-10}$ — нет.

В экспоненциальном представлении можно показывать не только десятичные, но и двоичные числа:

$$1.0_2 \times 2^{-1}$$

Чтобы сохранить двоичное число в нормализованной форме, требуется основание, которое можно будет увеличивать или уменьшать на конкретное количество разрядов, чтобы сдвинуть число и получить одну ненулевую цифру слева от используемого в нем разделителя целой и дробной части. Нашим потребностям могут удовлетворять только числа по основанию 2. Поскольку основание не десятичное, то вместо названия «десятичная точка» понадобится что-то другое, в качестве чего подойдет название *двоичная точка*.

Компьютерная арифметика, поддерживающая такие числа, называется арифметикой с **плавающей точкой** (floating point), потому что она представляет числа, в которых двоичная точка не фиксирована, как в целых числах. В языке программирования C для таких чисел используется имя *float*. Как и в экспоненциальном виде, числа представляются в виде одной ненулевой цифры слева от двоичной точки, для чего используется следующая форма:

$$1.xxxxxxxxx_2 \times 2^{yyyy}$$

(Хотя компьютер представляет экспоненту (показатель степени) по основанию 2, как и все остальное число, чтобы упростить запись, мы показываем порядок в виде десятичного числа.)

Стандартная экспоненциальная запись для вещественных чисел в нормализованной форме дает три преимущества. Она упрощает обмен данными, включающими числа с плавающей точкой, за счет однообразного представления алгоритмы арифметики чисел с плавающей точкой и повышает точность чисел, сохраняемых в слове, поскольку ненужные лидирующие нули заменяются реальными цифрами, расположенными справа от двоичной точки.

Экспоненциальное представление
Форма записи, представляющая числа с одной цифрой слева от десятичного знака.

Нормализованное число
Число в форме записи с плавающей точкой, не имеющее лидирующих нулей.

Плавающая точка (floating point)
Компьютерная арифметика, представляющая числа, в которых двоичная точка не зафиксирована.

Мантисса
Значение, которое обычно находится в диапазоне от нуля до единицы и помещается в поле мантиссы.

Экспонента
В системе числового представления арифметики чисел с плавающей точкой значение, помещаемое в поле экспоненты.

Представление с плавающей точкой

Разработчик представления с плавающей точкой должен найти компромисс между размером **мантиссы** (дробной части) и размером **экспоненты** (показателя порядка), поскольку фиксированный размер слова означает, что добавить разряд к одной из этих частей можно, только отняв его у другой части. Возникает компромисс между точностью и диапазоном: увеличение размера дробной части повышает точность, а увеличение размера порядка повы-

шает диапазон представляемых чисел. Конструкторские установки, изложенные во второй главе, напоминают о том, что хорошая конструкция требует удачных компромиссов.

Числа с плавающей точкой обычно кратны размеру слова. Ниже показано представление числа с плавающей точкой, принятое в MIPS, где s — это знак числа с плавающей точкой (1 означает, что оно отрицательное), *экспонента* — это значение 8-разрядного поля порядка (включая знак порядка), а *мантисса* — это 23-разрядное число. Это представление называется *знаком и абсолютной величиной*, поскольку знак является разрядом, отделенным от остального числа.

В общем виде числа с плавающей точкой имеют следующую форму:

$$(-1)^S \times F \times 2^E$$

F содержит значение поля мантиссы, а E содержит значение поля экспоненты; точные отношения между ними будут рассмотрены далее. (Вскоре будет показано, что в MIPS далеко не все так просто.)

Размеры, выбранные для экспоненты и мантиссы, предоставляют компьютерной арифметике MIPS весьма широкий диапазон чисел. Самые малые представляемые в компьютере числа могут иметь значение вплоть до $2.0_{10} \times 10^{-38}$, а самые большие — вплоть до $2.0_{10} \times 10^{38}$. Тем не менее это еще очень далеко от бесконечности, поэтому числа могут быть слишком большими. Поэтому, как и в целочисленной арифметике, в арифметике чисел с плавающей точкой могут происходить прерывания, связанные с переполнением. Учтите, что в данном случае **переполнение** означает, что экспонента числа слишком велика, чтобы быть представленной в поле экспоненты.

Числа с плавающей точкой (запятой) предлагают также новый вид исключительного события. Программистам нужно будет владеть информацией не только о том, что вычисляемое число слишком велико, чтобы быть представленным, но и о том, что вычисляемая ими дробь стала настолько маленькой (но не равной нулю), что она не может быть представлена. Любое из этих событий может привести к тому, что программа выдаст неверные ответы. Чтобы отличить второе событие от переполнения, мы назовем его **потерей значимости** (underflow). Связанная с ним ситуация складывается в том случае, когда отрицательная экспонента слишком велика, чтобы поместиться в поле экспоненты.

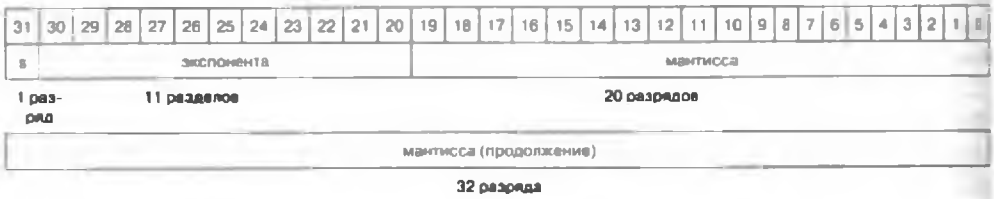
Переполнение (для чисел с плавающей точкой) Ситуация, в которой положительная экспонента становится слишком большой, чтобы поместиться в поле экспоненты.

Потеря значимости (при работе с числами с плавающей точкой) Ситуация, в которой отрицательная экспонента становится слишком большой, чтобы поместиться в поле экспоненты.

Одним из способов снижения шансов возникновения потери значимости или переполнения является предложение еще одного формата с более крупной экспонентой. В языке C такое число называют двойным — *double*, а опе-

рации над двойными числами называют арифметикой чисел с плавающей точкой **двойной точности**; действия с числами с плавающей точкой ранее рассмотренного формата называют арифметикой чисел с плавающей точкой **одинарной** точности.

Представление числа с плавающей точкой двойной точности занимает, как показано ниже, два MIPS-слова, где *s* по-прежнему является знаком числа, экспонента выражается значением 11-разрядного поля *экспоненты*, а мантисса — 52-разрядным полем *мантиссы*.



Двойная точность MIPS позволяет представлять малые числа вплоть до $2.0_{10} \times 10^{-308}$ и большие числа вплоть до $2.0_{10} \times 10^{308}$. Хотя двойная точность увеличивает диапазон экспоненты, ее главным преимуществом является более высокая точность числа благодаря значительно более длинной значащей части.

Эти форматы выходят за рамки MIPS. Они являются частью *стандарта чисел с плавающей точкой* IEEE 754, который действует практически для каждого компьютера, изобретенного начиная с 1980 года. Этот стандарт не только существенно улучшил переносимость программ, работающих с числами с плавающей точкой, но и повысил качество компьютерной арифметики.

Чтобы запаковать в значащую часть еще больше разрядов, в IEEE 754 лидирующий единичный бит нормализованных двоичных чисел сделан подразумеваемым. Поэтому при одинарной точности число на самом деле имеет длину 24 разряда (подразумеваемая единица и 23-разрядная мантисса), а при двойной точности — 53 разряда (1 + 52). Проявляя педантичность, мы будем использовать понятие *значащая часть* для представления 24- или 53-разрядных чисел, то есть единица плюс мантисса, и понятие *мантисса*, когда речь будет идти о 23- или 52-разрядном числе. Поскольку у нуля лидирующая единица отсутствует, он задается с помощью зарезервированного показателя порядка 0, не позволяющего оборудованию прикреплять к нему лидирующую единицу.

Таким образом, $00...00_2$ представляет нуль, а в представлении всех остальных чисел используется предыдущая форма с добавлением скрытой единицы:

$$(-1)^S \times (1 + \text{Мантисса}) \times 2^E,$$

где разряды мантиссы представляют собой число между нулем и единицей, а E определяет значение поля экспоненты, которое вскоре будет рассмотрено более подробно. Если пронумеровать разряды мантиссы *слева направо* s1, s2, s3, ... , то значение приобретет следующий вид:

Двойная точность
Значение с плавающей точкой, представленное двумя 32-разрядными словами.

Одинарная точность
Значение с плавающей точкой, представленное одним 32-разрядным словом.

$$(-1)^S \times (1 + (s1 \times 2^{-1}) + (s2 \times 2^{-2}) + (s3 \times 2^{-3}) + (s4 \times 2^{-4}) + \ldots) \times 2^E.$$

В табл. 3.4 показано кодирование стандарта чисел с плавающей точкой IEEE 754. Еще одной особенностью IEEE 754 являются специальные символы для представления неординарных событий. Например, вместо прерывания при делении на нуль программное обеспечение может установить результат битовому шаблону, представляющему $+\infty$ или $-\infty$; Для этих специальных символов зарезервирован самый большой показатель порядка. Когда программист выводит результат, программа выведет символ бесконечности. (Для искушенных в математике следует отметить, что цель использования бесконечности состоит в формировании топологического замыкания вещественных чисел.)

Таблица 3.4. Кодирование чисел с плавающей точкой IEEE 754. Отдельный разряд sign определяет знак числа. Ненормализованные числа описываются в «Уточнении» подраздела «Краткие выводы»

Одинарная точность		Двойная точность		Представляемый объект
Экспонента	Мантисса	Экспонента	Мантисса	
0	0	0	0	0
0	Не нуль	0	Не нуль	± ненормализованное число
1–254	Какая угодно	1–2046	Какая угодно	± число с плавающей точкой
255	0	2047	0	± бесконечность
255	Не нуль	2047	Не нуль	NaN (Не число)

В стандарте IEEE 754 предусмотрен даже символ для результата недопустимых операций, например 0/0 или вычитание бесконечности из бесконечности. Это символ NaN, означающий Not a Number — «не число». Цель использования символов NaN заключается в том, чтобы дать возможность программистам отложить некоторые проверки и решения на то время, когда их будет удобнее провести.

Разработчики IEEE 754 также захотели получить представление числа с плавающей точкой, которое могло бы свободно обрабатываться целочисленным сравнением, особенно в целях сортировки. Это желание предопределило местоположение знака в самом старшем разряде, что позволяет провести быстрый тест на «меньше чем», «больше чем» или «равно нулю». (Это несколько сложнее простой целочисленной сортировки, поскольку данная форма записи состоит, по сути, из знака и значения, а не из числа с дополнением до двух.)

Помещение экспоненты перед значащей частью также упрощает сортировку чисел с плавающей точкой с помощью инструкций целочисленного сравнения, поскольку числа, имеющие более крупную экспоненту, выглядят больше чисел, имеющих меньшую экспоненту, если обе экспоненты имеют одинаковые знаки.

Отрицательные экспоненты затрудняют упрощенную сортировку. При использовании записи с дополнением до двух или любой другой формы записи, в которой отрицательные экспоненты в самом старшем разряде поля экспоненты имеют

единицу, отрицательная экспонента будет похожа на большое число. Например, $1.0_2 \times 2^{-1}$ будет представлено как

(Не забудьте о подразумеваемой единице в значащей части.) Значение $1.0_2 \times 2^{-1}$ было бы похожим на меньшее двоичное число

Поэтому требуемая система записи должна представлять самую отрицательную экспоненту как $00...00_2$, а самую положительную как $11...11_2$. Это соглашение называется *смещенной нотацией* (biased notation), где смещение является числом, вычтенным из нормального, беззнакового, представления с целью определения реального значения.

Для чисел с одинарной точностью в IEEE 754 используется смещение 127, поэтому экспонента −1 представлена битовым шаблоном значения $-1 + 127_{10}$, или $126_{10} = 0111\ 1110_2$, а +1 представлена $1 + 127$, или $128_{10} = 1000\ 0000_2$. Смещением экспоненты для чисел с двойной точностью служит число 1023. Смещенная экспонента означает, что значение, представленное числом с плавающей точкой, на самом деле является следующим числом:

$$(-1)^S \times (1 + \text{Мантисса}) \times 2^{(\text{Экспонента} - \text{Смещение})}$$

В таком случае диапазон чисел с одинарной точностью простирается от самого малого числа

$$\pm 1.0000\ 0000\ 0000\ 0000\ 0000\ 000_2 \times 2^{-126}$$

до самого большого

$$\pm 1.1111\ 1111\ 1111\ 1111\ 1111\ 111_2 \times 2^{+127}.$$

Давайте покажем внешний вид этого представления.

Упражнение

Представление чисел с плавающей точкой

Покажите двоичное представление числа -0.75_{10} с одинарной и двойной точностью, соответствующее стандарту IEEE 754.

Ответ

Число -0.75_{10} можно также представить как

$$-3/4_{10} \text{ или } -3/2^2{}_{10}.$$

Оно также может быть представлено в виде двоичной дроби:

$$-11_2/2^2{}_{10} \text{ или } -0.11_2.$$

В экспоненциальной записи это значение имеет вид

$$-0.11_2 \times 2^0.$$

А в нормализованной экспоненциальной записи оно имеет вид

$$-1.1_2 \times 2^{-1}.$$

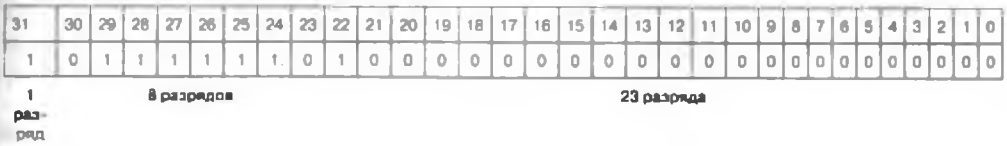
В общем виде числа с одинарной точностью представляются следующим образом:

$$(-1)^S \times (1 + \text{Мантисса}) \times 2^{(\text{Экспонента} - 127)}.$$

Вычитание смещения 127 из экспоненты числа $-1.1_2 \times 2^{-1}$ дает следующий результат:

$$(-1)^1 \times (1 + .1000\ 0000\ 0000\ 0000\ 0000\ 000_2) \times 2^{(126 - 127)}.$$

Тогда представление числа -0.75_{10} с одинарной точностью имеет следующий вид:



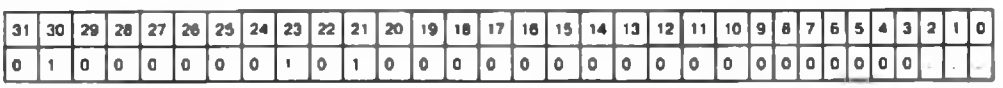
Представление этого же числа с двойной точностью имеет следующий вид:

$$(-1)^1 \times (1 + .1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_2) \times 2^{(1022 - 1023)}$$

Теперь попробуем проделать все это в обратном порядке

Упражнение

Преобразование двоичного числа с плавающей точкой в десятичное
Какое десятичное число представляет следующее число с плавающей точкой одинарной точности?



Ответ

Знаковый разряд имеет значение 1, поле экспоненты содержит число 129, а поле мантиссы содержит число $1 \times 2^{-2} = 1/4$, или 0.25. Используя базовую формулу,

$$(-1)^S \times (1 + \text{Мантисса}) \times 2^{(\text{Экспонента} - \text{Смещение})} = (-1)^1 \times (1 + 0.25) \times 2^{(129 - 127)} =$$

$$= -1 \times 1.25 \times 2^2 = -1.25 \times 4 = -5.0$$

В следующих подразделах будут показаны алгоритмы для сложения и умножения чисел с плавающей точкой. В своей основе они используют соответствующие целочисленные операции над значащими частями, но для обработки экспоненты и нормализации результата нужны дополнительные действия. Сначала будет дано наглядное представление алгоритмов в десятичных числах, а затем более подробно будет представлена двоичная версия.

Уточнение. В попытке расширения диапазона без перемещения разрядов из значащей части в некоторых компьютерах еще до принятия стандарта IEEE 754 использовалось основание, отличное от двух. Например, в универсальных компьютерах IBM 360 и 370 использовалось основание 16. Поскольку изменение экспоненты в машине IBM на единицу означало сдвиг значащей части на 4 разряда, «нормализованные» числа по основанию 16 могли иметь до 3 нулевых лидирующих разрядов! Следовательно, шестнадцатеричные цифры означают, что из значимой части пропущено до трех разрядов, что приводит к неожиданным проблемам, касающимся точности арифметики чисел с плавающей точкой. Универсальные машины IBM последних выпусков уже поддерживали не только шестнадцатеричный формат, но и стандарт IEEE 754.

Сложение чисел с плавающей точкой

Давайте сложим числа в экспоненциальной записи вручную, чтобы показать все сложности этой операции с числами с плавающей точкой: $9.999_{10} \times 10^1 + 1.610_{10} \times 10^{-1}$. Предположим, что можно сохранить только четыре десятичные цифры значимой части и две десятичные цифры экспоненты.

Шаг 1. Чтобы правильно сложить эти числа, нужно привести к общему виду десятичную точку числа с меньшей экспонентой. Следовательно, нам нужна форма наименьшего из чисел, $1.610_{10} \times 10^{-1}$, которая бы соответствовала большей экспоненте. Эту форму можно вывести, если принять во внимание, что у представления числа с плавающей точкой в экспоненциальной записи имеется множество ненормализованных форм:

$$1.610_{10} \times 10^{-1} = 0.1610_{10} \times 10^0 = 0.01610_{10} \times 10^1.$$

Требуемая версия показана справа, поскольку ее экспонента соответствует экспоненте наибольшего из двух чисел, $9.999_{10} \times 10^1$. Таким образом, первый шаг заключается в сдвиге значимой части наименьшего из двух чисел вправо до тех пор, пока экспонента не будет скорректирована до соответствия экспоненте наибольшего из двух чисел. Но мы можем использовать в представлении только четыре десятичные цифры, поэтому после сдвига число приобретет следующий вид:

$$0.016_{10} \times 10^1.$$

Шаг 2. Затем настанет черед сложения значащих частей:

$$\begin{array}{r} 9.999_{10} \\ +\ 0.016_{10} \\ \hline 10.015_{10} \end{array}$$

Получится сумма $10.015_{10} \times 10^1$.

Шаг 3. Эта сумма не соответствует нормализованной экспоненциальной форме записи, поэтому ее необходимо скорректировать:

$$10.015_{10} \times 10^1 = 1.0015_{10} \times 10^2.$$

Таким образом, после сложения может потребоваться сдвиг суммы для приведения ее к нормализованной форме, с соответствующей корректировкой экспоненты. В этом примере показан сдвиг вправо, но если одно число было бы положительным, а другое отрицательным, вполне возможно, что у суммы было бы много лидирующих нулей, требующих сдвигов влево. Независимо от увеличения или уменьшения экспоненты, нужно проводить проверку на переполнение или на потерю значимости, то есть нужно убедиться в том, что экспонента все еще помещается в своем поле.

Шаг 4. Поскольку было поставлено условие, что в значащей части должно быть всего четыре цифры (исключая знак), число следует округлить. Согласно правилу средней школы, если стоящая справа от нужной позиции цифра находится в диапазоне от 0 до 4, число просто усекается, а если эта цифра находится в диапазоне от 5 до 9, то после усечения к числу прибавляется единица. Число

$$1.0015_{10} \times 10^2$$

округляется до четырех цифр в значащей части и приобретает следующий вид:

$$1.002_{10} \times 10^2,$$

поскольку четвертая цифра справа от десятичной точки находилась в диапазоне между 5 и 9. Следует заметить, что при неблагоприятном стечении обстоятельств при округлении, например при прибавлении единицы к строке, состоящей из девяток, сумма может потерять нормализованную форму и потребуется выполнить шаг 3 еще раз.

На рис. 3.11 показан алгоритм сложения двоичных чисел с плавающей точкой, который соответствует приведенному примеру сложения десятичных чисел. Шаги 1 и 2 соответствуют шагам в только что рассмотренном примере: сначала к общему виду приводится значащая часть числа с наименьшей экспонентой, а затем складываются две значащие части. Проверка на переполнение или на потерю значимости в шаге 3 зависит от точности операндов. Вспомним, что нули во всех разрядах в экспоненте зарезервированы и используются в числах с плавающей точкой для представления нуля. Кроме того, все единицы в экспоненте зарезервированы в числах с плавающей точкой для отображения значений и ситуаций, выходящих за рамки нормальных чисел (см. «Уточнение» в подразделе «Краткие выводы»). Таким образом, для чисел с одинарной точностью максимальная экспонента равна 127, а минимальная — −126. Предельные значения для чисел с двойной точностью равны 1023 и −1022.

Рис. 3.11. Сложение чисел с плавающей точкой. При обычных обстоятельствах шаги 3 и 4 выполняются по одному разу, но если при округлении сумма получается ненормализованной, шаг 3 нужно повторить

Упражнение

Сложение двоичных чисел с плавающей точкой

Попробуйте сложить числа 0.5_{10} и -0.4375_{10} в двоичном представлении, используя алгоритм, показанный на рис. 3.11.

Ответ

Сначала рассмотрим двоичную версию этих двух чисел в нормализованном экспоненциальном представлении, предполагая, что точность ограничена 4 разрядами:

$$\begin{aligned} 0.5_{10} &= 1/2_{10} &&= 1/2^1_{10} \\ &= 0.1_2 &&= 0.1_2 \times 2^0 \quad = 1.000_2 \times 2^{-1} \\ -0.4375_{10} &= -7/16_{10} &&= -7/24_{10} \\ &= -0.0111_2 &&= -0.0111_2 \times 2^0 \quad = -1.110_2 \times 2-2 \end{aligned}$$

Теперь последуем алгоритму:

Шаг 1. Значащая часть числа с наименьшей экспонентой ($-1.11_2 \times 2^{-2}$) сдвигается вправо до тех пор, пока ее экспонента не сравняется с экспонентой наибольшего числа:

$$-1.110_2 \times 2^{-2} = -0.111_2 \times 2^{-1}.$$

Шаг 2. Сложение значащих частей:

$$1.000_2 \times 2^{-1} + (-0.111_2 \times 2^{-1}) = 0.001_2 \times 2^{-1}.$$

Шаг 3. Нормализация суммы, проверка на переполнение или на потерю значимости:

$$\begin{aligned} 0.0012 \times 2-1 \quad &= 0.0102 \times 2-2 = 0.1002 \times 2-3 \\ &= 1.0002 \times 2-4 \end{aligned}$$

Поскольку $127 \geqslant -4 \geqslant -126$, переполнение или потеря значимости не произошли. (Смещенная экспонента должна иметь значение $-4 + 127$, или 123, и это значение находится между 1 и 254, то есть между наименьшим и наибольшим значениями незарезервированных смещенных экспонент.)

Шаг 4. Округление суммы:

$$1.000_2 \times 2^{-4}$$

Сумма как раз помещается в 4 разряда, поэтому при округлении разряды не изменяются.

Тогда сумма равна

$$\begin{aligned} 1.0002 \times 2-4 &= 0.00010002 = 0.00012 \\ &= 1/2410 \quad = 1/1610 \quad = 0.062510 \end{aligned}$$

Эта сумма соответствует ожидаемой от сложения 0.5_{10} и -0.4375_{10}.

У многих компьютеров оборудование предназначено для выполнения операций над числами с плавающей точкой с наивысшей скоростью. На рис. 3.12 показана базисная структура оборудования для сложения чисел с плавающей точкой.

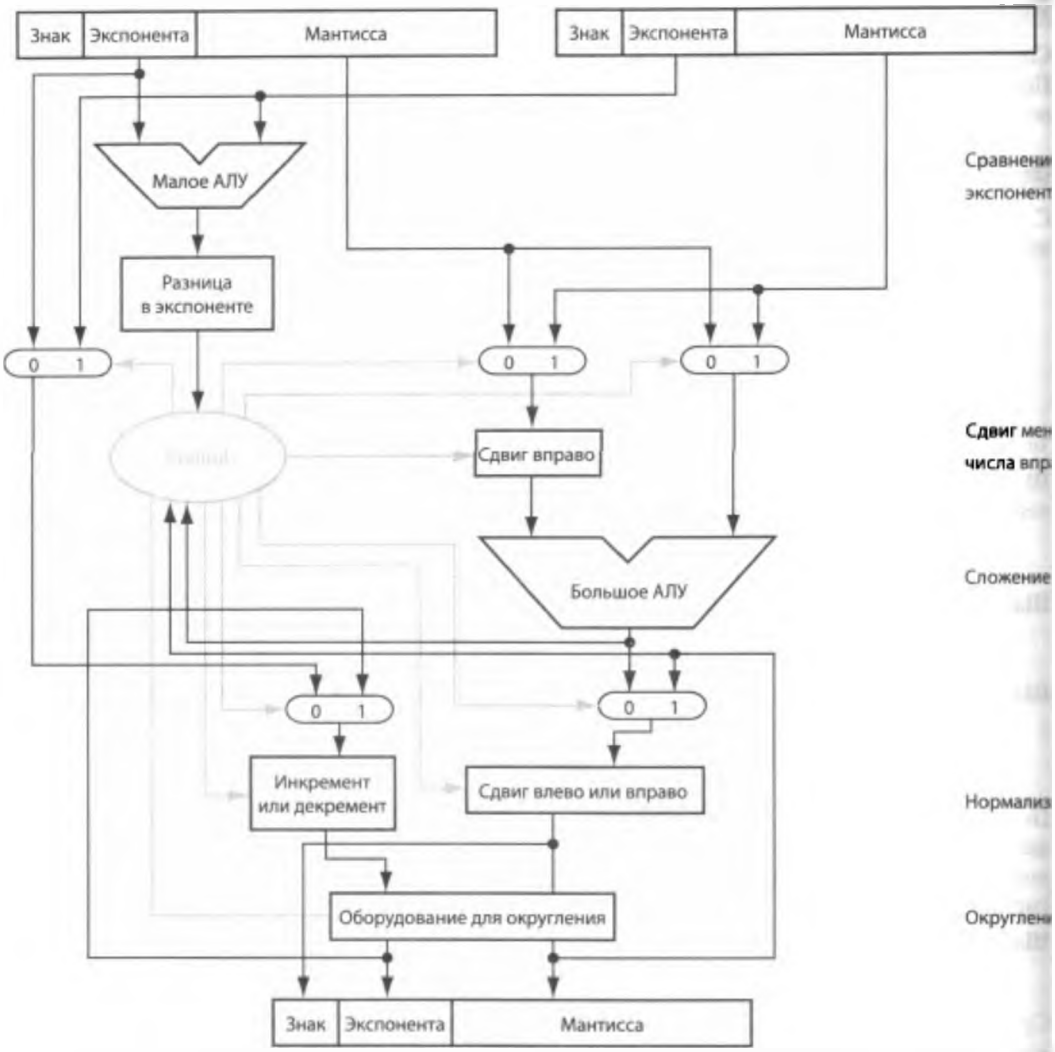


Рис. 3.12. Блок-схема арифметического устройства, предназначенного для сложения чисел с плавающей точкой. Каждому блоку сверху вниз соответствуют шаги, изображенные на рис. 3.11. Сначала с помощью малого АЛУ экспонента одного операнда вычитается из экспоненты второго операнда, чтобы определить, какая из них больше и на сколько. Эта разница управляет тремя коммутаторами; слева направо они выбирают наибольшую экспоненту, значащую часть наименьшего числа и значащую часть наибольшего числа. Значащая часть наименьшего числа сдвигается вправо, а затем значащие части складываются с помощью большого АЛУ. Затем на шаге нормализации сумма сдвигается влево или вправо, увеличивая или уменьшая экспоненту. Затем с помощью округления формируется конечный результат, которому может понадобиться повторная нормализация

Умножение чисел с плавающей точкой

После объяснения сложения чисел с плавающей точкой перейдем к их умножению. Начнем с умножения вручную двух десятичных чисел в экспоненциальном представлении: $1.110_{10} \times 10^{10} \times 9.200_{10} \times 10^{-5}$. Предположим, что в значащей части можно сохранять четыре цифры, а в экспоненте — две.

Шаг 1. В отличие от сложения экспонента произведения вычисляется простым сложением экспонент операндов:

$$\text{Новая экспонента} = 10 + (-5) = 5.$$

Проделаем то же самое со смещенными экспонентами, чтобы убедиться в получении такого же результата:

$$\text{Новая экспонента} = 137 + 122 = 259.$$

Этот результат слишком велик для восьмиразрядного поля экспоненты, значит, что-то тут не так! Проблема кроется в смещении, поскольку мы сложили не только экспоненты, но и их смещения:

$$\text{Новая экспонента} = (10 + 127) + (-5 + 127) = (5 + 2 \times 127) = 259.$$

Следовательно, для получения суммы с правильным смещением при сложении смещенных чисел нужно вычесть из суммы смещение:

$$\text{Новая экспонента} = 137 + 122 - 127 = 259 - 127 = 132 = (5 + 127)$$

и 5 является той самой экспонентой, вычисленной изначально.

Шаг 2. Теперь настал черед перемножения значащих частей:

$$\begin{array}{r} 1.110_{10} \\ \times\ 9.200_{10} \\ \hline 0000 \\ 0000 \\ 2220 \\ 9990 \\ \hline 10212000_{10} \end{array}$$

У каждого операнда справа от десятичной точки находится по три цифры, поэтому десятичная точка в значащей части произведения ставится с отступом шести цифр справа:

$$10.212000_{10}.$$

Если принять во внимание, что справа от десятичной точки могут храниться только три цифры, то получим произведение 10.212×10^5.

Шаг 3. Это произведение имеет ненормализованный вид, поэтому оно нуждается в нормализации:

$$10.212_{10} \times 10^5 = 1.0212_{10} \times 10^6.$$

Таким образом, после умножения произведение должно быть сдвинуто вправо на одну цифру, чтобы оно приобрело нормализованный вид, при этом к экспоненте прибавляется единица. Здесь нужно проверить, не возникло ли переполнение или потеря значимости. Потеря значимости может произойти, если оба операнда имеют небольшие значения, то есть если у обоих из них большие отрицательные экспоненты.

Шаг 4. У нас было условие, что значащие части имеют всего четыре цифры (исключая знак), поэтому число требует округления. Число

$$1.0212_{10} \times 10^6$$

округляется до четырех цифр в значащей части, и получается

$$1.021_{10} \times 10^6$$

Шаг 5. Знак произведения зависит от знаков исходных операндов. При совпадении знаков у операндов у произведения знак положительный, а при несовпадении — отрицательный.

Следовательно, произведение имеет значение

$$+1.021_{10} \times 10^6.$$

Знак суммы в алгоритме сложения определялся путем сложения значимых частей, но в умножении знак произведения определяется исходя из знаков операндов.

И опять, как показано на рис. 3.13, умножение двоичных чисел с плавающей точкой очень похоже на только что проделанные нами шаги. Сначала для произведения путем сложения смещенных экспонент и обязательного вычитания смещения вычисляется новая экспонента. Затем перемножаются значащие части, после чего следует не всегда обязательный шаг нормализации. Размер экспоненты проверяется на переполнение или на потерю значимости, а затем произведение округляется. Если округление приводит к необходимости повторной нормализации, размер экспоненты проверяется еще раз. И наконец, знаковый разряд устанавливается в единицу, если у операндов были разные знаки (произведение становится отрицательным) или в нуль, если знаки были одинаковыми (произведение становится положительным).

Упражнение

Умножение двоичных чисел с плавающей точкой

Попробуйте перемножить числа 0.5_{10} и -0.4375_{10}, используя шаги, показанные на рис. 3.13.

Ответ

В двоичном виде задача состоит в умножении числа $1.000_2 \times 2^{-1}$ на число $-1.110_2 \times 2^{-2}$.

Шаг 1. Сложение экспонент без смещения:

$$-1 + (-2) = -3$$

или с использованием представления со смещением:

$$(-1 + 127) + (-2 + 127) - 127 = (-1 - 2) + (127 + 127 - 127) = -3 + 127 = 124$$

Рис. 3.13. Умножение чисел с плавающей точкой. Обычно шаги 3 и 4 выполняются один раз, но если округление произведения приводит к его ненормализованному виду, шаг 3 необходимо повторить

Шаг 2. Перемножение значащих частей:

$$\begin{array}{r} 1.000_2 \\ \times\ 1.110_2 \\ \hline 0000 \\ 1000 \\ 1000 \\ 1000 \\ \hline 1110000_2 \end{array}$$

Произведение равно $1.110000_2 \times 2^{-3}$, но нам нужно уместить его в четырех разрядах, поэтому оно равно $1.110_2 \times 2^{-3}$.

Шаг 3. Теперь произведение приводится, если это нужно, к нормализованному виду, а затем экспонента проверяется на переполнение или на потерю значимости. Произведение уже имеет нормализованный вид, а поскольку $127 \geq -3 \geq -126$, переполнение или потеря значимости не наступили. (При использовании смещенного представления, $254 \geq 124 \geq 1$, значит, экспонента умещается в отведенном для нее пространстве.)

Шаг 4. Округление произведения оставляет число без изменений:

$$1.110_2 \times 2^{-3}$$

Шаг 5. Поскольку знаки исходных операндов отличались друг от друга, произведению присваивается отрицательный знак. Следовательно, произведение имеет значение

$$-1.110_2 \times 2^{-3}.$$

Для проверки результата преобразуем число в десятичное:

$$\begin{aligned} -1.1102 \times 2{-}3 &= -0.0011102 = -0.001112 \\ &= -7/2510 = -7/3210 = -0.2187510. \end{aligned}$$

Произведение, полученное при умножении 0.5_{10} на -0.4375_{10}, действительно равно -0.21875_{10}.

Инструкции для работы с числами с плавающей точкой

MIPS поддерживает форматы одинарной и двойной точности стандарта IEEE 754 с помощью следующих инструкций:

- Сложения чисел с плавающей точкой с одинарной (`add.s`) и двойной (`add.d`) точностью
- Вычитания чисел с плавающей точкой с одинарной (`sub.s`) и двойной (`sub.d`) точностью
- Умножения чисел с плавающей точкой с одинарной (`mul.s`) и двойной (`mul.d`) точностью
- Деления чисел с плавающей точкой с одинарной (`div.s`) и двойной (`div.d`) точностью
- Сравнения чисел с плавающей точкой с одинарной (`c.x.s`) и двойной (`c.x.d`) точностью, где `x` может выражать равенство — equal (`eq`), неравенство — not equal (`neq`), меньше чем — less than (`lt`), меньше чем или равно — less than or

equal (le), больше чем — greater than (gt) или больше чем или равно — greater than or equal (ge)

- Перехода, когда результат вычисления с числами с плавающей точкой истина — true (bc1t) или ложь — false (bc1f)

При сравнении чисел с плавающей точкой в зависимости от условия сравнения разряд устанавливается в true или false, а затем, в зависимости от условия, принимается решение, нужно или нет осуществлять условный переход по результатам операции над числами с плавающей точкой.

Разработчики MIPS решили добавить для работы с числами с плавающей точкой отдельные регистры, которые называются $f0, $f1, $f2, ... и используются для чисел как с одинарной, так и с двойной точностью. Соответственно они включили в набор отдельные инструкции для загрузки и сохранения регистров, предназначенных для чисел с плавающей точкой: lwc1 и swc1. Базовыми регистрами для переноса данных, имеющих формат с плавающей точкой, остались целочисленные регистры.

MIPS-код для загрузки из памяти двух чисел с одинарной точностью, их сложения, а затем сохранения суммы может иметь следующий вид:

```
lwc1    $f4,x($sp)  # Загрузка 32-разрядного числа с плавающей точкой в F4
lwc1    $f6,y($sp)  # Загрузка 32-разрядного числа с плавающей точкой в F6
add.s   $f2,$f4,$f6 # F2 = F4 + F6 (одинарная точность)
swc1    $f2,z($sp)  # Сохранение 32-разрядного числа с плавающей точкой из F2
```

На самом деле регистр для чисел с двойной точностью — это пара, составленная из регистров для чисел с одинарной точностью с четным и нечетным номерами, с присвоением этому регистру имени четного регистра. Таким образом, пара регистров для чисел с одинарной точностью $f2 и $f3, кроме всего прочего, формирует также и регистр для числа с двойной точностью с именем $f2.

Часть рассматриваемой в данной главе MIPS-архитектуры, предназначенной для работы с числами с плавающей точкой, показана в табл. 3.5. Дополнения, внесенные в архитектуру для работы с числами с плавающей точкой, выделены жирным шрифтом. На рис. 3.14 показана кодировка инструкций, подобная той, которая была показана на рис. 2.12 в главе 2.

Таблица 3.5. MIPS-архитектура, рассмотренная в предыдущих разделах

Операнды MIPS, используемые для работы с числами с **плавающей точкой**

Название	Пример	Комментарии
32-разрядные регистры для чисел с плавающей точкой	$f0, $f1, $f2, . . . , $f31	**MIPS-регистры для чисел с плавающей точкой, используемые для чисел с двойной точностью попарно**
2^{32} слов памяти	Память[0], Память[4], . . . , Память[4294967292]	Доступны только для инструкций по переносу данных. В MIPS используется байтовая адресация, поэтому адреса последовательных слов отличаются друг от друга на 4. В памяти хранятся структуры данных, например массивы, и регистры с «утечкой» содержимого, например регистры, сохраненные при вызовах процедур.

продолжение

Таблица 3.5 (*продолжение*)

Язык ассемблера MIPS для работы с числами
с **плавающей точкой — floating-point (FP)**

Категория	Инструкция	Пример	Значение	Комментарии
Арифметика	FP add single	add.s $f2,$f4,$f6	$f2 = $f4 + $f6	FP-сложение (одинарная точность)
	FP subtract single	sub.s $f2,$f4,$f6	$f2 = $f4 - $f6	FP-вычитание (одинарная точность)
	FP multiply single	mul.s $f2,$f4,$f6	$f2 = $f4 × $f6	FP-умножение (одинарная точность)
	FP divide single	div.s $f2,$f4,$f6	$f2 = $f4 / $f6	FP-деление (одинарная точность)
	FP add double	add.d $f2,$f4,$f6	$f2 = $f4 + $f6	FP-сложение (двойная точность)
	FP subtract double	sub.d $f2,$f4,$f6	$f2 = $f4 - $f6	FP-вычитание (двойная точность)
	FP multiply double	mul.d $f2,$f4,$f6	$f2 = $f4 × $f6	FP-умножение (двойная точность)
	FP divide double	div.d $f2,$f4,$f6	$f2 = $f4 / $f6	FP-деление (двойная точность)
Перенос данных	load word copr. 1	lwc1 $f1,100($s2)	$f1 = Память[$s2 + 100]	32-разрядные данные в FP-регистр
	store word copr. 1	swc1 $f1,100($s2)	Память[$s2 + 100] = $f1	32-разрядные данные в память
	branch on FP true	bc1t 25	если (условие == 1), переход на PC + 4 + 100	Переход при соблюдении условия работы с FP-числами, осуществляемый относительно PC (счетчика команд)
	branch on FP false	bc1f 25	если (условие == 0), переход на PC + 4 + 100	Переход при несоблюдении условия работы с FP-числами, осуществляемый относительно PC (счетчика команд)
	FP compare single (eq, ne, lt, le, gt, ge)	c.lt.s $f2,$f4	если ($f2 < $f4), условие = 1, иначе условие = 0	Сравнение «меньше чем» при работе с FP- числами одинарной точности
	FP compare double (eq, ne, lt, le, gt, ge)	c.lt.d $f2,$f4	если ($f2 < $f4), условие = 1; иначе условие = 0	Сравнение «меньше чем» при работе с FP- числами двойной точности

Машинный язык MIPS для работы с числами с **плавающей точкой**

Название	Формат	Пример						Комментарии
add.s	R	17	16	6	4	2	0	add.s $f2,$f4,$f6
sub.s	R	17	16	6	4	2	1	sub.s $f2,$f4,$f6
mul.s	R	17	16	6	4	2	2	mul.s $f2,$f4,$f6
div.s	R	17	16	6	4	2	3	div.s $f2,$f4,$f6
add.d	R	17	17	6	4	2	0	add.d $f2,$f4,$f6
sub.d	R	17	17	6	4	2	1	sub.d $f2,$f4,$f6
mul.d	R	17	17	6	4	2	2	mul.d $f2,$f4,$f6
div.d	R	17	17	6	4	2	3	div.d $f2,$f4,$f6
lwc1	I	49	20	2	100			lwc1 $f2,100($s4)
swc1	I	57	20	2	100			swc1 $f2,100($s4)
bc1t	I	17	8	1	25			bc1t 25
bc1f	I	17	8	0	25			bc1f 25
c.lt.s	R	17	16	4	2	0	60	c.lt.s $f2,$f4
c.lt.d	R	17	17	4	2	0	60	c.lt.d $f2,$f4
Размер поля		6 разр.	5 разр.	5 разр.	5 разр.	5 разр.	6 разр.	Все MIPS-инструкции 32-разрядные

Интерфейс аппаратного и программного обеспечения

Один из вопросов, с которым приходится сталкиваться разработчикам при поддержке арифметики чисел с плавающей точкой, касается того, нужно ли использовать те же регистры, что и для инструкций, работающих с целыми числами, или же нужно добавить специальный набор регистров для чисел с плавающей точкой. Поскольку программы обычно выполняют операции над целыми числами и операции над числами с плавающей точкой, используя разные данные, разделение регистров лишь незначительно увеличит число инструкций, необходимых для выполнения программ. Главное последствие введения отдельных регистров заключается в необходимости создания отдельного набора инструкций переноса данных для их перемещения между регистрами для чисел с плавающей точкой и памятью. Преимущество использования отдельных регистров для чисел с плавающей точкой — вдвое большее количество регистров без задействования большего количества разрядов в формате инструкции, удвоенная пропускная способность регистров за счет использования отдельных наборов регистров для работы с целыми числами и для работы с числами с плавающей точкой; например, некоторые компьютеры приводят операнды всех размерностей к единому внутреннему формату.

op(31:26):

28-26 31-29	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(000)	Rfmt	Bltz/aez	j	jal	beq	bne	blez	bgtz
1(001)	addi	addiu	slti	sltiu	ANDi	ORi	xORi	lui
2(010)	TLB	FlPt						
3(011)								
4(100)	lb	ln	lwl	lw	lbu	lhu	l wr	
5(101)	sb	sh	swl	sw			swr	
6(110)	lwc0	lwc1						
7(111)	swc0	swc1						

op(31:26) = 010001 (FlPt), (rt(16:ie) s 0 => c = f, rt(16:16) = 1 =:> c = t), rs(25:21):

23-21 25-24	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(00)	mfcl		cfcl		mtcl		ctcl	
1(01)	bcl.C							
2(10)	*f* = single (одинарн. точн.)	*f* = double (двойная точн.)						
3(11)								

op(31:26) = 010001 (FlPt), (f above: 10000 => f = s, 10001 => f = d), funct(5:0):

2-0 5-3	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(000)	add.*f*	sub. *f*	mul. *f*	div. *f*		abs. *f*	mov. *f*	neg. *f*
1(001)								
2(010)								
3(011)								
4(100)	cvt.s. *f*	cvt.d. *f*			cvt.w. *f*			
5(101)								
6(110)	c.f. *f*	c.un. *f*	c.eq. *f*	c.ueq. *f*	colt. *f*	cult. *f*	cole. *f*	cule. *f*
7(111)	c.sf. *f*	c.ngle. *f*	c.seq. *f*	c.ngl. *f*	c.lt. *f*	cnge. *f*	cle. *f*	cngt. *f*

Рис. 3.14. Кодировка MIPS-инструкций, предназначенных для работы с числами с плавающей точкой. Эта система записи дает значение поля по строкам и столбцам. Например, в верхней части рисунка код инструкции lw можно найти в строке номер 4 (100_2 для разрядов 31–29 инструкции) и в столбце номер 3 (011_2 для разрядов 28–26 инструкции), то есть соответствующее значение поля op (разряды 31–26) равно 100011_2. Подчеркивание означает, что поле используется где-то в другом месте. Например, FlPt в строке 2 и столбце 1 (op = 010001_2) определяется в нижней части рисунка. Соответственно sub. f в строке 0 и столбце 1 в нижней части означает, что поле funct (разряды 5–0 инструкции) имеет значение 000001_2, а поле op (разряды 31–26) имеет значение 010001_2. Учтите, что пятиразрядное поле rs, указанное в средней части рисунка, определяет, предназначена ли инструкция для работы с числами одинарной (f = s, следовательно, rs = 10000) или двойной точности (f = d, следовательно, rs = 10001). По аналогии с этим, разряд 16 инструкции определяет, проверяется инструкция bcl c на истину — true (разряд 16 = 1 =>bcl.t) или на ложь — false (разряд 16 = 0 =>bcl f). Выделенные инструкции описываются в главе 2 или в данной главе

Упражнение

Компиляция программы на C, работающей с числами с плавающей точкой, в ассемблерный код MIPS

Давайте переведем температуру по Фаренгейту в температуру по Цельсию:

```
float f2c (float fahr)
  {
    return ((5.0/9.0) * (fahr - 32.0));
  }
```

Предположим, что аргумент в формате числа с плавающей точкой `fahr` передан в регистре `$f12`, а результат должен попасть в регистр `$f0`. (В отличие от регистров для работы с целыми числами регистр 0 для числа с плавающей точкой может содержать число.) Каким будет ассемблерный код MIPS?

Ответ

Предположим, что компилятор поместил три константы в формате чисел с плавающей точкой в память в пределах досягаемости посредством глобального указателя `$gp`. Первые две инструкции загружают константы 5.0 и 9.0 в регистры для чисел с плавающей точкой:

```
f2c:
    lwc1 $f16,const5($gp)  # $f16 = 5.0 (значение 5.0 находится в памяти)
    lwc1 $f18,const9($gp)  # $f18 = 9.0 (значение 9.0 находится в памяти)
```

Затем производится деление для получения дробного значения 5.0/9.0:

```
    div.s $f16, $f16, $f18 # $f16 = 5.0 / 9.0
```

(Многие компиляторы разделят 5.0 на 9.0 в процессе компиляции и сохранят в памяти одну константу 5.0/9.0, исключив, таким образом, деление в процессе выполнения программы.) Следующим шагом загружается константа 32.0, а затем она вычитается из `fahr` (`$f12`):

```
    lwc1 $f18, const32($gp)  # $f18 = 32.0
    sub.s $f18, $f12, $f18   # $f18 = fahr - 32.0
```

И наконец, два промежуточных результата перемножаются, и произведение помещается в регистр `$f0` в качестве возвращаемого результата, после чего осуществляется возвращение из процедуры

```
    mul.s $f0,  $f16, $f18  # $f0 = (5/9)*(fahr - 32.0)
    jr  $ra                 # возврат
```

А теперь выполним операции для чисел с плавающей точкой над матрицами; код с такими операциями часто встречается в программах научных расчетов.

Упражнение

Компиляция в MIPS-код процедуры на языке C, работающей с числами с плавающей точкой, принадлежащими двумерным матрицам

Большинство вычислений, производимых над числами с плавающей точкой, осуществляется с двойной точностью. Давайте выполним перемножение матриц X = X + Y * Z. Предположим, что X, Y и Z — это прямоугольные матрицы, имеющие в каждом измерении по 32 элемента.

```
void mm (double x[][], double y[][], double z[][])
{
  int i, j, k;

  for (i = 0; i != 32; i = i + 1)
  for (j = 0; j != 32; j = j + 1)
```

```
    for (k = 0; k != 32; k = k + 1)
      x[i][j] = x[i][j] + y[i][k] * z[k][j];
}
```

Стартовые адреса массивов являются параметрами, поэтому они находятся соответственно в регистрах `$a0`, `$a1` и `$a2`. Предположим, что целочисленные переменные находятся соответственно в регистрах `$s0`, `$s1` и `$s2`. Каким будет код тела этой процедуры на языке ассемблера MIPS?

Ответ

Следует заметить, что элемент `x[i][j]` используется в самом внутреннем цикле показанного выше кода. Поскольку индексом цикла служит переменная `k`, индекс не влияет на `x[i][j]`, поэтому можно не загружать и не сохранять `x[i][j]` при каждой итерации. Вместо этого компилятор загружает `x[i][j]` в регистр за пределами цикла, аккумулирует сумму произведений `y[i][k]` и `z[k][j]` в том же самом регистре, а затем, после завершения самого внутреннего цикла, сохраняет сумму в элементе `x[i][j]`.

Мы упростим код за счет использования псевдоинструкций языка ассемблера `li` (которая загружает константу в регистр), а также `l.d` и `s.d` (которые ассемблер превратит в пару инструкций `lwc1` или `swc1`, переносящих данные в пару регистров, предназначенных для хранения чисел с плавающей точкой).

Тело процедуры начинается с сохранения во временном регистре значения 32, используемого для прекращения цикла, и продолжается инициализацией трех переменных цикла `for`:

```
mm...
        li $t1, 32  # $t1 = 32 (размер строки/конец цикла)
        li $s0, 0   # i = 0; инициализация 1-го цикла for
L1:     li $s1, 0   # j = 0; перезапуск 2-го цикла for
L2:     li $s2, 0   # k = 0; перезапуск 3-го цикла for
```

Чтобы вычислить адрес элемента `x[i][j]`, нужно знать, как хранится в памяти двумерный массив 32 × 32. Вполне разумно предположить, что его размещение аналогично размещению 32 одномерных массивов по 32 элемента в каждом. Поэтому первым шагом будет пропуск i «одномерных массивов», или строк, для получения нужной нам строки. Таким образом, мы умножаем индекс первой размерности на размер строки, 32. Поскольку 32 является степенью числа 2, вместо умножения можно воспользоваться сдвигом:

```
        sll  $t2, $s0, 5    # $t2 = i * 25 (размер строки массива x)
```

Теперь для выбора `j`-того элемента нужной строки прибавим второй индекс:

```
        addu  $t2, $t2, $s1 # $t2 = i * размер(строка) + j
```

Чтобы превратить эту сумму в байтовый индекс, мы умножаем ее на размер элемента матрицы в байтах. Поскольку каждый элемент для чисел с двойной точностью состоит из 8 байт, вместо этого можно осуществить тройной сдвиг влево:

```
        sll  $t2, $t2, 3    # $t2 = байтовое смещение для [i][j]
```

Затем мы добавим эту сумму к базовому адресу `x`, предоставляя адрес элемента `x[i][j]`, а затем загрузим число с двойной точностью, хранящееся в элементе `x[i][j]`, в регистр `$f4`:

```
        addu  $t2, $a0, $t2  # $t2 = байтовый адрес элемента x[i][j]
        l.d   $f4, 0($t2)    # $f4 = 8 байтов элемента x[i][j]
```

Следующие пять инструкций фактически идентичны последним пяти: они вычисляют адрес, а затем загружают число с двойной точностью, которое хранится в элементе `z[k][j]`.

```
L3:     sll  $t0, $s2, 5      # $t0 = k * 25 (размер строки массива z)
        addu $t0, $t0, $s1    # $t0 = k * размер(строка) + j
        sll  $t0, $t0, 3      # $t0 = байтовое смещение [k][j]
```

```
addu $t0, $a2, $t0     # $t0 = байтовый адрес элемента z[k][j]
l.d  $f16, 0($t0)      # $f16 = 8 байтов элемента z[k][j]
```

Следующие пять инструкций также похожи на последние пять: они вычисляют адрес, а затем загружают число с двойной точностью, которое хранится в элементе `y[i][k]`.

```
sll  $t0, $s0, 5       # $t0 = i * 25 (размер строки массива y)
addu $t0, $t0, $s2     # $t0 = i * размер(строка) + k
sll  $t0, $t0, 3       # $t0 = байтовое смещение [i][k]
addu $t0, $a1, $t0     # $t0 = байтовый адрес элемента y[i][k]
l.d  $f18, 0($t0)      # $f18 = 8 байтов элемента y[i][k]
```

Теперь, когда все данные загружены, мы наконец-то готовы для проведения ряда операций над числами с плавающей точкой! Мы перемножаем элементы `y` и `z`, размещенные в регистрах `$f18` и `$f16`, а затем накапливаем сумму в регистре `$f4`.

```
mul.d $f16, $f18, $f16  # $f16 = y[i][k] * z[k][j]
add.d $f4, $f4, $f16    # f4 = x[i][j]+ y[i][k] * z[k][j]
```

В завершающем блоке происходит приращение индекса `k` и циклический возврат назад, если индекс не равен 32. Если он равен 32, то есть если достигнут конец самого внутреннего цикла, нам нужно сохранить сумму, накопленную в `$f4`, в элементе `x[i][j]`.

```
addiu  $s2, $s2, 1     # $k = k + 1
bne    $s2, $t1, L3    # если (k != 32), переход на L3
s.d    $f4, 0($t2)     # x[i][j] = $f4
```

Точно так же последние четыре инструкции занимаются приращением индексных переменных среднего и самого внешнего циклов, осуществляя циклический возврат, если индекс не равен 32, и выход из цикла, если индекс равен 32.

```
addiu  $s1, $s1, 1     # $j = j + 1
bne    $s1, $t1, L2    # если (j != 32), переход на L2
addiu  $s0, $s0, 1     # $i = i + 1
bne    $s0, $t1, L1    # если (i != 32), переход на L1
```

Уточнение. Рассмотренное в примере размещение массива, называемое развертыванием по строкам, используется в языке C и многих других языках программирования. В Фортране вместо этого используется развертывание по столбцам, в соответствии с чем массив хранится по столбцам.

Уточнение. Изначально для операций с двойной точностью могут использоваться только 16 из 32 MIPS-регистров, предназначенных для чисел с плавающей точкой (FP): `$f0`, `$f2`, `$f4`, ..., `$f30`. Двойная точность в вычислениях достигается за счет использования пар этих регистров, способных хранить числа с одинарной точностью. FP-регистры с нечетными номерами используются только для загрузки и хранения правых половин 64-разрядных чисел с плавающей точкой. В MIPS-32 к набору инструкций добавлены инструкции `l.d` и `s.d`. В MIPS-32 также добавлены версии всех инструкций с плавающей точкой, где одна инструкция приводит к двум параллельным операциям с числами с плавающей точкой над двумя 32-разрядными операндами внутри 64-разрядных регистров. Например, `add.ps $f0, $f2, $f4` является эквивалентом инструкции `add.s $f0, $f2, $f4`, за которой следует инструкция `add.s $f1, $f3, $f5`.

Уточнение. Еще одна причина разделения регистров для работы с целыми числами и регистров для чисел с плавающей точкой состоит в том, что микропроцессоры в 1980-х годах не имели достаточного количества транзисторов, чтобы блок для работы с числами с плавающей точкой поместился на том же самом чипе, что и блок для

работы с целыми числами. Поэтому блок для работы с числами с плавающей точкой, включая и соответствующие регистры, был доступен в качестве второго чипа. Такие дополнительные ускорители назывались сопроцессорами, что объясняет происхождение акронима для загрузки в MIPS чисел с плавающей точкой: `lwc1` означает «загрузить слово в сопроцессор 1» — load word to coprocessor 1, то есть в блок арифметики с плавающей точкой. (Сопроцессор 0 относится к виртуальной памяти, которая рассматривается в главе 5.) С начала 1990-х годов у микропроцессоров появился встроенный блок арифметики с плавающей точкой (как, собственно, и все остальное), и поэтому термин *сопроцессор* образует одно целое с *сумматором* и *основной памятью*.

Уточнение. Как уже упоминалось в разделе 3.4, ускорение операции деления дается намного труднее, чем ускорение умножения. Кроме технологии SRT, есть еще одна технология, позволяющее ускорить работу устройства деления, так называемый метод *простых итераций Ньютона*, при котором деление превращено в нахождение нуля функции для определения величины, обратной 1/x, которая затем перемножается с другим операндом. Технология итерации *не позволяет* провести нужное округление без вычисления множества дополнительных разрядов. Эту проблему путем вычисления сверхточной обратной величины решает микросхема TI.

Уточнение. В языке Java стандарт IEEE 754 используется посредством упоминания в своих определениях типов данных и операций, касающихся чисел с плавающей точкой в Java. Поэтому код, приведенный в первом примере, возможно, неплохо бы подошел для создания метода класса, переводящего значения из шкалы по Фаренгейту в шкалу по Цельсию.

Во втором примере используются многомерные массивы, которые напрямую в Java не поддерживаются. Язык Java позволяет создавать массивы массивов, но каждый массив должен иметь свою собственную длину, в отличие от многомерных массивов в языке C. Как и в тех примерах, которые приводились в главе 2, Java-версия этого второго примера потребует немалого объема кода для проверки границ массивов, включая вычисление новой длины в конце доступа к строке. Кроме этого, потребуется проверка на то, что ссылка на объект не является пустой.

Точная арифметика

В отличие от целых чисел, которые могут представлять в точности каждое число между самым малым и самым большим числами, числа с плавающей точкой обычно являются приближенным значением того числа, которое они на самом деле представить не в состоянии. Причина в том, что, скажем, между нулем и единицей находится бесконечное множество вещественных чисел, в то время как числами с плавающей точкой двойной точности может быть представлено не более чем 2^{53} вещественных чисел. Не остается ничего лучшего, как получить такое представление в виде числа с плавающей точкой, которое будет ближе всего к реальному числу. Поэтому стандарт IEEE 754 предлагает несколько режимов округления, позволяя программисту выбрать нужное приближение.

Сам термин «округление» звучит довольно просто, но для точного округления требуется оборудование, включающее в процесс вычисления дополнительные

разряды. В предыдущих примерах мы не уточняли количество разрядов, которое могло занять промежуточное представление, но вполне очевидно, что если каждый промежуточный результат должен быть усечен до конкретного количества цифр, то возможности для округления просто не представится. Поэтому, согласно стандарту IEEE 754, при промежуточном сложении справа всегда приберегаются два дополнительных разряда, которые называются соответственно **защитой** (guard) и **округлением** (round). Чтобы проиллюстрировать их важность, рассмотрим пример с использованием десятичных чисел.

Упражнение

Округление с разрядами защиты

Сложите $2.56_{10} \times 10^0$ с $2.34_{10} \times 10^2$, при условии, что имеются три значащих десятичных разряда. Округлите до ближайшего десятичного числа с тремя значащими десятичными разрядами сначала с использованием разрядов защиты и округления, а затем без их использования.

Ответ

Сначала нужно произвести сдвиг вправо наименьшего числа для выравнивания экспонент, превратив $2.56_{10} \times 10^0$ в $0.0256_{10} \times 10^2$. При наличии разрядов защиты и округления появляется возможность при выравнивании экспонент представить два самых младших разряда. Разряд защиты содержит значение 5, а разряд округления содержит значение 6. Сумма вычисляется следующим образом:

$$\begin{array}{r} 2.3400_{10} \\ +0.0256_{10} \\ \hline 2.3656_{10} \end{array}$$

и представляется числом $2.3656_{10} \times 10^2$. Поскольку две цифры подлежат округлению, значения от 0 до 49 нужно округлять вниз, а значения от 51 до 99 — вверх, а число 50 нуждается в дополнительном рассмотрении. При округлении суммы вверх до трех значащих цифр получается результат $2.37_{10} \times 10^2$.

Выполнение округления *без* разрядов защиты и округления выводит за пределы вычислений два разряда. При этом новая сумма вычисляется следующим образом:

$$\begin{array}{r} 2.34_{10} \\ +0.02_{10} \\ \hline 2.36_{10} \end{array}$$

Ответ становится равным $2.36_{10} \times 10^2$, теряя при этом по сравнению с предыдущей суммой единицу из последней цифры.

Защита (Guard)
Первый из двух дополнительных разрядов, оставляемых справа при промежуточных вычислениях с числами с плавающей точкой; используется для повышения точности округления.

Округление (round)
Метод, заставляющий промежуточный результат помещаться в формат чисел с плавающей точкой; цель его применения состоит обычно в том, чтобы найти ближайшее число, которое может быть представлено в этом формате.

Поскольку наихудшей ситуацией для округления будет получение такого числа, которое находится ровно на полпути между двумя представлениями чисел с плавающей точкой, мерой точности в числах с плавающей точкой обычно является количество разрядов в ошибке в наи-

меньших разрядах мантиссы; эта мера называется количеством **единиц последнего разряда** — units in the last place, или **ulp**. Если число лишилось двух единиц в младших разрядах, то это будет называться потерей двух ulp. При условии, что не возникло переполнения, потери значимости или исключения в связи с недопустимой операцией, стандарт IEEE 754 гарантирует, что компьютер будет работать с числом, находящимся в пределах половины единицы последнего разряда, — 0.5 ulp.

Уточнение. Хотя для приведенного выше примера реально нужна была только одна дополнительная цифра, для умножения могут понадобиться и две цифры. У двоичного произведения может быть один лидирующий нулевой разряд, следовательно, при нормализации произведение должно быть смещено на один разряд влево. Тем самым разряд защиты будет смещен в самый младший разряд произведения, оставляя разряд округления для того, чтобы помочь точному округлению произведения.

В IEEE 754 имеется четыре режима округления: постоянное округление вверх (по направлению к $+\infty$), постоянное округление вниз (по направлению к $-\infty$), усечение и округление до ближайшего четного числа. Последний режим определяет, что делать, если число оказывается точно посредине. Налоговое управление США всегда округляет 0,50 доллара вверх, возможно, для того, чтобы получить более весомую выручку. Справедливее было бы округлять вверх в половине всех таких случаев и вниз в другой половине. Стандарт IEEE 754 гласит, что если самый младший разряд, оказавшийся на полпути, имеет нечетное значение, нужно прибавить единицу, а если он имеет четное значение, его нужно просто отбросить. Этот метод всегда дает нуль в самом младшем разряде в случае неоднозначной ситуации, откуда и происходит название этого режима округления. Это самый распространенный и единственный поддерживаемый в Java режим округления.

Предназначение дополнительных разрядов округления заключается в том, чтобы позволить компьютеру получить одинаковые результаты, как и в том случае, если бы промежуточные результаты были вычислены с бесконечной точностью, а затем округлены. Для достижения этой цели и округления числа до ближайшего четного значения кроме разрядов защиты и округления у стандарта есть еще и третий разряд, который устанавливается, когда справа от разряда округления есть ненулевые разряды. Этот **разряд напоминания** (sticky bit) позволяет компьютеру увидеть при округлении разницу между $0.50...00_{10}$ и $0.50...01_{10}$.

Разряд напоминания может быть установлен, к примеру, при сложении, когда число с меньшим значением сдвигается вправо. Предположим, что в приведенном выше примере складываются числа $5.01_{10} \times 10^{-1}$ и $2.34_{10} \times 10^{2}$. Даже при наличии разрядов защиты и округления нам пришлось бы складывать 0.0050 и 2.34, получая сумму 2.3450. Разряд напоминания будет установлен, поскольку справа имеются ненулевые разряды. Без использования этого разряда, позволяющего запомнить, что были сдвинуты какие-либо единицы, мы предположили бы, что число равно

Единицы последнего разряда (ulp)
Количество разрядов в ошибке в младшем разряде мантиссы между фактическим числом и числом, которое может быть представлено.

Разряд напоминания
Разряд, используемый при округлении в дополнение к разрядам защиты и округления, который устанавливается, если справа от разряда округления имеются ненулевые разряды.

2.345000...00, и округлили бы его до ближайшего четного числа 2.34. При наличии этого разряда, позволяющего запомнить, что число больше, чем 2.345000...00, оно округляется до 2.35.

Уточнение. Архитектуры PowerPC, SPARC64 и AMD SSE5 предоставляют одну инструкцию, выполняющую сразу умножение и сложение, работая с тремя регистрами: a = a + (b × c). Вполне очевидно, что эта инструкция для этой распространенной операции потенциально позволяет добиться более высокой производительности при работе с числами с плавающей точкой. Не менее важно, что вместо двух округлений: после умножения, а затем после сложения — что бы произошло при использовании отдельных инструкций, — инструкция умножения и сложения может ограничиться одним округлением после сложения. Единственное округление повышает точность умножения со сложением. Такая операция с единственным округлением называются **совмещенной операцией умножения-сложения** (fused multiply add). Она была добавлена к измененному стандарту IEEE 754.

Краткие выводы

Следующий далее подраздел «Общее представление» подкрепляет концепцию хранимой программы, изложенную в главе 2; это означает, что характер информации не может быть установлен простым изучением битов, поскольку одни и те же биты могут представлять множество объектов. В этом разделе показывается, что компьютерная арифметика имеет определенные ограничения и может иметь расхождения с обычной арифметикой. Например, представление чисел с плавающей точкой, соответствующее стандарту IEEE 754

$$(-1)^S \times (1 + \text{Мантисса}) \times 2^{(\text{Экспонента} - \text{Смещение})},$$

практически всегда является приближением к фактическому числу. Компьютерные системы должны брать на себя минимизацию этого разрыва между компьютерной арифметикой и арифметикой реального мира, и программисты порой должны знать о значениях этого приближения.

Общее представление

У битовых комбинаций нет никаких предопределенных значений. Они могут представлять целые числа одинарной точности, целые числа без знака, числа с плавающей точкой, инструкции и т. д. Что именно ими представляется — зависит от инструкции, которая оперирует битами в слове.

Основное отличие компьютерных чисел от чисел реального мира состоит в том, что компьютерные числа имеют ограниченный размер, а стало быть, и ограниченную точность; существует возможность получения в результате вычисления слишком большого или слишком малого числа, которое не может быть представлено в слове. Программисты должны помнить об этих ограничениях и составлять программы соответствующим образом.

Совмещенное умножение-сложение Инструкция для чисел с плавающей точкой, выполняющая сразу и умножение, и сложение, но производящая округление только один раз после сложения.

Типы в языке C	Типы в языке Java	Перенос данных	Операции
int	int	lw, sw, lui	add, addi, sub, mult, div, AND, ANDi, OR, ORi, NOR, slt, slti
unsigned int	—	lw, sw, lui	addu, addiu, subu, multu, divu, AND, ANDi, OR, ORi, NOR, sltu, sltiu
char	—	lb, sb, lui	add, addi, sub, mult, div, AND, ANDi, OR, ORi, NOR, slt, slti
—	char	lh, sh, lui	addu, addiu, subu, multu, divu, AND, ANDi, OR, ORi, NOR, sltu, sltiu
float	float	lwc1, swc1	add.s, sub.s, mult.s, div.s, c.eq.s, c.lt.s, c.le.s
double	double	l.d, s.d	add.d, sub.d, mult.d, div.d, c.eq.d, c.lt.d, c.le.d

Интерфейс аппаратного и программного обеспечения

В предыдущей главе были представлены классы памяти языка программирования C (см. подраздел «Интерфейс аппаратного и программного обеспечения» в разделе 2.8). В показанной выше таблице приведены некоторые типы данных, имеющиеся в C и Java, инструкции переноса данных MIPS и инструкции, работающие с этими типами, фигурирующие в главе 2 и в данной главе. Следует заметить, что в Java целые числа без знака не используются.

Самопроверка

Предположим, что используется 16-разрядный формат чисел с плавающей точкой, соответствующий стандарту IEEE 754 и имеющий пять разрядов экспоненты.

Каким, скорее всего, будет диапазон чисел, отображаемых с его помощью?

1. От $1.0000\ 0000\ 00 \times 2^{0}$ до $1.1111\ 1111\ 11 \times 2^{31}$, 0
2. От $\pm 1.0000\ 0000\ 0 \times 2^{-14}$ до $\pm 1.1111\ 1111\ 1 \times 2^{15}$, ± 0, $\pm\infty$, NaN
3. От $\pm 1.0000\ 0000\ 00 \times 2^{-14}$ до $\pm 1.1111\ 1111\ 11 \times 2^{15}$, ± 0, $\pm\infty$, NaN
4. От $\pm 1.0000\ 0000\ 00 \times 2^{-15}$ до $\pm 1.1111\ 1111\ 11 \times 2^{14}$, ± 0, $\pm\infty$, NaN

Уточнение. Чтобы приспособиться к сравнениям, которые могут включать значения NaN, в стандарт в качестве возможных вариантов сравнений включено упорядоченное — ordered и неупорядоченное — unordered сравнение. В связи с этим полный набор инструкций MIPS располагает множеством разновидностей сравнений для поддержки значений NaN. (Java не поддерживает неупорядоченные сравнения.)

С целью выжать каждый последний разряд точности из операции над числами с плавающей точкой стандарт допускает представление некоторых чисел в ненор-

лизованной форме. Вместо разрыва между нулем и наименьшим нормализованным числом IEEE допускает ненормализованные числа (также известные как нестандартные — denorms или субнормальные — subnormals). У них такая же экспонента, как и у нуля, но ненулевая значащая часть. Они позволяют числу терять свое значение до тех пор, пока оно не станет нулем, что называется постепенной потерей значимости. Например, наименьшее положительное нормализованное число одинарной точности равно

$$1.0000\ 0000\ 0000\ 0000\ 0000\ 000_2 \times 2^{-126},$$

а наименьшее денормализованное число одинарной точности равно

$$0.0000\ 0000\ 0000\ 0000\ 0000\ 001_2 \times 2^{-126}, \text{ или } 1.0_2 \times 2^{-149}.$$

Для чисел двойной точности разрыв между нормализованной и ненормализованной формой составляет от 1.0×2^{-1022} и до 1.0×2^{-1074}.

Возможность применения время от времени ненормализованных операндов стала головной болью для разработчиков, стремящихся создать быстродействующие блоки для работы с числами с плавающей точкой. Поэтому на многих компьютерах при встрече ненормализованного операнда возникают исключения, позволяющие программе завершить операцию.

Несмотря на полностью работоспособные реализации программ, их низкая производительность снизила популярность нестандартных чисел в переносимых программах, работающих с числами с плавающей точкой. Более того, если программисты не ожидают применения нестандартных чисел, то программы могут вызвать у них немалое удивление.

3.6. Параллелизм и компьютерная арифметика: ассоциативность

Программы, перед тем как они переписываются для параллельного запуска, сначала, как правило, пишутся для последовательного запуска, поэтому вполне естественно задать вопрос: «Получается ли в результате работы обеих версий один и тот же ответ?». Если ответ отрицательный, предполагается наличие ошибки в параллельной версии, которую нужно отследить.

Такой подход предполагает, что компьютерная арифметика при переходе от последовательного к параллельному вычислению на результаты не влияет. То есть если нужно сложить вместе миллион чисел, нужно получить одинаковые результаты, независимо от того, сколько процессоров используется, один или тысяча. Это предположение соблюдается для целых чисел с дополнением до двух, даже если вычисление приводит к переполнению. По-другому можно сказать, что целочисленное сложение является ассоциативным.

Увы, поскольку числа с плавающей точкой являются приближением вещественных чисел и поскольку компьютерная арифметика имеет ограниченную точность, ассоциативность для чисел с плавающей точкой не соблюдается. То есть сложение чисел с плавающей точкой не является ассоциативным.

Упражнение

Проверка ассоциативности сложения чисел с плавающей точкой
Посмотрите, соблюдается ли условие x + (y + z) = (x + y) + z. Предположим, например, что $x = -1.5_{10} \times 10^{38}$, $y = 1.5_{10} \times 10^{38}$, а z = 1.0 и все числа имеют одинарную точность.

Ответ

С учетом широкого диапазона чисел, которые могут быть представлены в формате с плавающей точкой, проблемы, как мы увидим из следующего примера, возникают при сложении двух больших чисел с разными знаками и небольшого числа:

$$\begin{aligned} x + (y + z) &= -1.5_{10} \times 10^{38} + (1.5_{10} \times 10^{38} + 1.0) \\ &= -1.5_{10} \times 10^{38} + (1.5_{10} \times 10^{38}) = 0.0 \\ (x + y) + z &= (-1.5_{10} \times 10^{38} + 1.5_{10} \times 10^{38}) + 1.0 \\ &= (0.0_{10}) + 1.0 \\ &= 1.0 \end{aligned}$$

Следовательно, x + (y + z) ≠ (x + y) + z, поэтому сложение чисел с плавающей точкой не является ассоциативным.

Поскольку числа с плавающей точкой имеют ограниченную точность и результат является приближением реального результата, $1.5_{10} \times 10^{38}$ настолько больше 1.0_{10}, что $1.5_{10} \times 10^{38} + 1.0$ все еще представляется числом $1.5_{10} \times 10^{38}$. Именно поэтому сумма x, y и z равна 0.0 или 1.0, в зависимости от порядка сложения чисел с плавающей точкой, и, следовательно, сложение таких чисел не является ассоциативным.

Более досадный случай подобного недоразумения происходит на параллельном компьютере, где планировщик операционной системы может использовать разное количество процессоров в зависимости от других программ, запущенных на параллельном компьютере. Недостаточно осведомленный программист, работающий на параллельной машине, может быть сбит с толку тем, что его программа будет получать несколько отличающиеся друг от друга результаты при каждом запуске одного и того же кода при одних и тех же входных данных, поскольку использование при каждом запуске разного количества процессоров будет приводить к тому, что суммы будут вычисляться в разном порядке.

С учетом подобных трудностей программисты, создающие параллельный код для работы с числами с плавающей точкой, должны проверить надежность результатов, даже если они не получают точно такой же ответ, как при использовании последовательного кода. Сфера деятельности, в которой занимаются такими проблемами, называется численным анализом. Этот предмет по праву занимает свое особое место в учебниках. Подобные проблемы стали причиной популярности таких библиотек численного анализа, как LAPACK и SCALAPAK, которые прошли проверку как в своей последовательной, так и в параллельной форме.

Уточнение. Проблема ассоциативности находит свое коварное проявление в том случае, когда два процессора проводят избыточное вычисление, которое выполняется в разном порядке, в результате чего получаются немного отличающиеся друг от друга ответы, хотя оба ответа считаются точными. Ошибка возникает в том случае, когда происходит сравнение чисел с плавающей точкой при условном переходе, и два процессора выбирают разные пути ветвления, в то время как здравый смысл подсказывает, что они должны выбирать один и тот же путь.

3.7. Реальное оборудование: вычисления чисел с плавающей точкой в x86

В x86 имеются обычные инструкции умножения и деления, которые работают исключительно с регистрами и не полагаются, как в MIPS, на использование регистров Hi и Lo. (Справедливости ради нужно отметить, что в самые последние версии набора инструкций MIPS добавлены точно такие же инструкции.)

Основное отличие становится заметным в инструкциях для работы с числами с плавающей точкой. Архитектура x86 для работы с этими числами отличается от архитектуры всех других, имеющихся в мире компьютеров.

Архитектура x86 для работы с числами с плавающей точкой

Сопроцессор Intel 8087, предназначенный для работы с числами с плавающей точкой, появился в 1980 году. Его архитектура расширяла архитектуру процессора 8086 примерно на 60 инструкций для работы с числами с плавающей точкой.

Для своих инструкций с плавающей точкой Intel предоставила стековую архитектуру: загрузка проталкивает числа в стек, операции обнаруживают операнды в двух верхних элементах стека, а сохранения могут убрать элементы из стека. Intel дополнила эту стековую архитектуру инструкциями и режимами адресации, позволяющими архитектуре получить некоторые преимущества над моделью «регистр–память». В дополнение к нахождению операндов в двух верхних элементах стека один операнд может быть в памяти или в одном из семи внутрипроцессорных регистров ниже вершины стека. Таким образом, полноценный набор стековых инструкций был дополнен ограниченным набором инструкций, использующих модель регистр-память.

Но все же этот гибрид является ограниченной моделью «регистр–память», поскольку инструкции загрузки всегда перемещают данные на вершину стека с увеличением значения указателя на эту вершину, а инструкции сохранения могут переместить в память лишь тот элемент, который находится на вершине стека. Для индикации вершины стека в Intel используется запись ST, а для представления i-того регистра ниже вершины стека используется запись ST(i).

Другим новшеством этой архитектуры являлось то, что операнды в регистровом стеке шире, чем при хранении в памяти, и все операции выполняются с этой расширенной внутренней точностью. В отличие от установленного в MIPS максимума в 64 разряда операнды с плавающей точкой в стеке x86 имеют ширину 80 разрядов. Числа при загрузке автоматически преобразуются во внутренний 80-разрядный формат, а при сохранении проходят обратное преобразование к приемлемому размеру. Эта *двойная расширенная точность* не поддерживается языками программирования, хотя она бы пригодилась программистам, разрабатывающим математические программы.

Данные в памяти могут быть 32-разрядными (одинарной точности) или 64-разрядными (двойной точности) числами с плавающей точкой. Версии этих инструкций, использующие модель «регистр–память» затем, перед выполнением операции,

будут осуществлять преобразование операндов, хранящихся в памяти, в этот используемый в Intel 80-разрядный формат. Инструкции переноса данных при загрузке и сохранении целых чисел будут также осуществлять автоматическое преобразование 16- и 32-разрядных целых чисел в числа с плавающей точкой, и наоборот.

Операции с плавающей точкой x86 могут быть разбиты на четыре основных класса:

1. Инструкции перемещения данных, включая загрузку (load), загрузку константы (load constant) и сохранение (store).
2. Арифметические инструкции, включая сложение (add), вычитание (subtract), умножение (multiply), деление (divide), извлечение квадратного корня (square root) и получение абсолютного значения (absolute value).
3. Инструкции сравнения, включая инструкции для отправки результата целочисленному процессору для осуществления условного перехода
4. Трансцендентные инструкции, включая получение синуса (sine), косинуса (cosine), логарифма (log) и возведения в степень (exponentiation).

Часть из 60 операций с плавающей точкой показана в табл. 3.6. Следует заметить, что при включении режимов извлечения операндов для этих операций будет получено еще больше комбинаций. Множество вариантов для сложения чисел с плавающей точкой показано в табл. 3.7.

Инструкции с плавающей точкой кодируются с использованием кода операции (opcode) ESC процессора 8086 и постбайтового адресного указателя (см. табл. 2.17). Для операций, работающих с памятью, резервируются два разряда для выбора операндов из 32- или 64-разрядных чисел с плавающей точкой или из 16- или 32-разрядных целых чисел. Те же два разряда используются в версиях без доступа к памяти, чтобы решить, нужно ли выталкивать значения из стека после операции и должен ли результат помещаться на вершину стека или в расположенный ниже регистр.

В прошлом производительность операций с плавающей точкой в семействе x86 оставалась далеко позади производительности других компьютеров. В результате этого, Intel создала более традиционную архитектуру инструкций с плавающей точкой, сделав ее частью SSE2.

Таблица 3.6. Инструкции с плавающей точкой x86. Фигурные скобки {} используются для показа дополнительных вариантов основных операций. {I} означает наличие целочисленной версии инструкции, {P} означает, что этот вариант будет после операции выталкивать один операнд из стека, а {R} означает изменение на противоположный порядок операндов в данной операции. В первом столбце показаны инструкции переноса данных, которые перемещают данные в память или в один из регистров, находящихся ниже вершины стека. Последние три операции первого столбца проталкивают константы в стек: pi, 1.0 и 0.0. Второй столбец содержит описанные выше арифметические операции. Обратите внимание на то, что последние три работают только с элементом стека, находящимся на его вершине. В третьем столбце показаны инструкции сравнения. Ввиду отсутствия специальных инструкций с плавающей точкой для условного перехода результат сравнения должен

быть перенесен в целочисленный центральный процессор с помощью инструкции FSTSW , либо в регистр AX, либо в память, за которой следует инструкция SAHF для установки кодов условий. После этого сравнение с плавающей точкой должно быть проверено с использованием целочисленных инструкций условного перехода. В последнем столбце представлены высокоуровневые операции с плавающей точкой. Здесь представлены не все комбинации, предлагаемые системой записи. Поэтому операции F{I}SUB{R}{P} представляют следующие инструкции из набора x86: FSUB, FISUB, FSUBR, FISUBR, FSUBP, FSUBRP. Для целочисленных инструкций вычитания отсутствуют варианты с выталкиванием элементов из стека (FISUBP) или с реверсным выталкиванием (FISUBRP)

Перенос данных	Арифметические операции	Операции сравнения	Трансцендентные операции
F{I}LD mem/ST(i)	F{I}ADD{P} mem/ST(i)	F{I}COM{P}	FPATAN
F{I}ST{P} mem/ ST(i)	F{I}SUB{R}{P} mem/ST(i)	F{I}UCOM{P}{P}	F2XM1
FLDPI	F{I}MUL{P} mem/ST(i)	FSTSW AX/mem	FCOS
FLD1	F{I}DIV{R}{P} mem/ST(i)		FPTAN
FLDZ	FSQRT		FPREM
	FABS		FSIN
	FRNDINT		FYL2X

Таблица 3.7. Варианты операндов для сложения с плавающей точкой в x86

Инструкция	Операнды	Комментарий
FADD		Оба операнда в стеке; результат заменяет содержимое вершины стека
FADD	ST(i)	Одним операндом-источником является *i*-тый регистр, расположенный ниже вершины стека; результат заменяет содержимое вершины стека
FADD	ST(i), ST	Один операнд находится на вершине стека; результат заменяет *i*-тый регистр, расположенный ниже вершины стека
FADD	mem32	Одним операндом-источником является 32-разрядная ячейка памяти; результат заменяет содержимое вершины стека
FADD	mem64	Одним операндом-источником является 64-разрядная ячейка памяти; результат заменяет содержимое вершины стека

Потоковые SIMD-расширения 2 (SSE2) компании Intel для архитектуры с плавающей точкой

В главе 2 отмечалось, что в 2001 году Intel добавила к своей архитектуре 144 инструкции, включая регистры и операции с плавающей точкой двойной точности. Это расширение включало восемь 64-разрядных регистров, используемых для операндов с плавающей точкой, предоставляя компьютеру другой, отличающийся

от уникальной стековой архитектуры объект. Компиляторы могли выбирать использование восьми SSE2-регистров в качестве регистров с плавающей точкой, подобных тем, которые имеются в других компьютерах. Компания AMD в качестве составляющей расширения AMD64 увеличила количество регистров до 16, а компания Intel, чтобы использовать это расширение у себя, изменила его название на EM64T. Краткая сводка инструкций SSE и SSE2 показана в табл. 3.8.

В дополнение к хранению в регистре чисел одинарной или двойной точности Intel позволяет нескольким операндам с плавающей точкой быть упакованными в один 128-разрядный регистр SSE2: четырем с одинарной или двум с двойной точностью. Таким образом, 16 регистров с плавающей точкой для SSE2 фактически имеют ширину 128 разрядов. Если операнды могут быть выровнены в памяти как 128-разрядные данные, то тогда путем переноса 128-разрядных данных можно загрузить и сохранить одной инструкцией сразу несколько операндов. Этот упакованный формат чисел с плавающей точкой поддерживается арифметическими операциями, которые могут работать одновременно над четырьмя числами с одинарной (PS) или двумя числами с двойной (PD) точностью. Производительность этой архитектуры в два с лишним раза превышает производительность стековой архитектуры.

Таблица 3.8. Инструкции SSE/SSE2, предназначенные для работы с числами с плавающей точкой, в семействе x86. Сокращение xmm означает, что один операнд является 128-разрядным SSE2-регистром, а mem/xmm означает, что другой операнд находится либо в памяти, либо в SSE2-регистре. Фигурные скобки {} используются для показа дополнительных вариантов основных операций: {SS} обозначает число с плавающей точкой с одинарной скалярной точностью — Scalar Single, или один 32-разрядный операнд в 128-разрядном регистре; {PS} обозначает упакованное число с плавающей точкой одинарной точности — Packed Single, или четыре 32-разрядных операнда в 128-разрядном регистре; {SD} обозначает число с плавающей точкой с двойной скалярной точностью — Scalar Double, или один 64-разрядный операнд в 128-разрядном регистре; {PD} обозначает упакованное число с плавающей точкой двойной точности — Packed Double, или два 64-разрядных операнда в 128-разряде; {A} обозначает 128-разрядный операнд, выровненный в памяти; {U} обозначает 128-разрядный операнд, невыровненный в памяти; {H} обозначает перемещение старшей половины разрядов 128-разрядного операнда; а {L} обозначает перемещение младшей половины разрядов 128-разрядного операнда

Перенос данных	Арифметические операции	Операции сравнения
MOV{A/U}{SS/PS/SD/PD} xmm, mem/xmm	ADD{SS/PS/SD/PD} xmm, mem/xmm	CMP{SS/PS/SD/PD}
	SUB{SS/PS/SD/PD} xmm, mem/xmm	
MOV {H/L} {PS/PD} xmm, mem/xmm	MUL{SS/PS/SD/PD} xmm, mem/xmm	
	DIV{SS/PS/SD/PD} xmm, mem/xmm	
	SQRT{SS/PS/SD/PD} mem/xmm	
	MAX {SS/PS/SD/PD} mem/xmm	
	MIN{SS/PS/SD/PD} mem/xmm	

3.8. Заблуждения и недоразумения

> Таким образом, математику можно определить как предмет, в котором мы никогда не знаем, о чем говорим, а также не знаем, когда сказанное нами является истиной.
>
> *Бертран Рассел*

Арифметические заблуждения и недоразумения возникают главным образом из-за разницы между ограниченной точностью компьютерной арифметики и неограниченной точностью обычной арифметики.

Заблуждение. *Точно так же как инструкция сдвига влево может заменить целочисленное умножение на два, сдвиг вправо равноценен целочисленному делению на два.*

Вспомним, что двоичное число x, где x*i* означает *i*-тый разряд, представляет число

$$\ldots + (x3 \times 2^3) + (x2 \times 2^2) + (x1 \times 2^1) + (x0 \times 2^0).$$

Сдвиг разрядов числа x вправо на *n* разрядов может показаться равноценным делению на 2^n. Для целых чисел без знака это утверждение является верным. Проблема возникает с целыми числами со знаком. Предположим, например, что нужно разделить -5_{10} на 4_{10}; частное должно быть равно -1_{10}. Представление числа -5_{10} с дополнением до двух имеет вид

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1011_2.$$

В соответствии с этим заблуждением сдвиг вправо на два разряда должен разделить его на 4_{10} (2^2):

$$0011\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_2.$$

С нулем в знаковом разряде этот результат заведомо неверен. Значение, созданное за счет сдвига вправо, на самом деле равно $1\ 073\ 741\ 822_{10}$, а не -1_{10}.

Решением могло бы стать наличие арифметического сдвига вправо, которое расширяло бы знаковый разряд вместо сдвига его в нуль. Двухразрядный арифметический сдвиг вправо числа -5_{10} производит число

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_2.$$

Результат равен -2_{10}, а не -1_{10}; почти то, что надо, но не совсем.

Заблуждение. *MIPS-инструкция сложения с непосредственным значением без знака — add immediate unsigned (addiu) не осуществляет знакового расширения своего 16-разрядного поля непосредственного значения.*

Несмотря на свое название, инструкция add immediate unsigned (addiu) используется для сложения констант с целыми числами со знаком, когда на переполнение не обращается внимание. В MIPS отсутствует инструкция вычитания непосредственного значения, и отрицательные числа нуждаются в расширении знака, поэтому разработчики MIPS-архитектуры решили осуществлять расширение знака поля непосредственного значения.

Заблуждение. *Точность в числах с плавающей точкой нужна только специалистам по теоретической математике.*

Заголовки газет в ноябре 1994 года гласили, что это утверждение является не более чем простым заблуждением (рис. 3.15). А история появления таких заголовков была следующей.

В Pentium используется стандартный алгоритм деления чисел с плавающей точкой, генерирующий несколько разрядов частного за один шаг. Он использует старшие разряды делителя и делимого, чтобы выстроить предположения насчет следующих двух разрядов частного. Это предположение возникало в результате просмотра таблицы, содержащей −2, −1, 0, +1 или +2. Предполагаемое число умножалось на делитель и вычиталось из остатка, чтобы сгенерировать новый остаток. Подобно делению без восстановления, если в результате предыдущего предположения получался слишком большой остаток, промежуточный остаток приводился к нужному значению при следующем проходе.

По-видимому, и специалисты Intel считали, что к пяти элементам таблицы из процессора 80486 никогда не будет обращений, и оптимизировали программируемую логическую матрицу, чтобы в ситуации с Pentium возвращался 0, а не 2. Но специалисты Intel ошиблись: обеспечив неизменно правильный результат в первых 11 разрядах, они допустили периодическое возникновение ошибки в разрядах с 12-й по 52-й, или с четвертой по пятнадцатую десятичную цифру.

Рис. 3.15. Примеры газетных и журнальных статей от ноября 1994 года, включая статьи из New York Times, San Jose Mercury News, San Francisco Chronicle и Infoworld. Ошибка Pentium при делении чисел с плавающей точкой даже вошла в список горячей десятки новостей — «Top 10 List» телевизионного шоу Дэвида Леттермана. В конечном итоге Intel потратил на замену содержащих ошибку чипов 300 миллионов долларов

События в этой поучительной истории с ошибкой в процессоре Pentium развивались в следующем порядке.

- *Июль 1994 года.* Intel обнаруживает ошибку в процессоре Pentium. Фактические затраты на исправление ошибки составили несколько сотен тысяч долларов. В соответствии с обычными процедурами исправления ошибок внесения изменений, повторные проверки и запуск в производство исправленного чипа могли занять несколько месяцев. Intel планировал запустить годный чип в производство в январе 1995 года, предполагая, что бракованных процессоров 3–5 млн.
- *Сентябрь 1994 года.* Профессор математики из колледжа в Линчбурге, штат Вирджиния, Томас Найсли (Thomas Nicely), обнаруживает ошибку. После звонка в службу технической поддержки компании Intel и отсутствия какой-либо официальной реакции, он опубликовал свое открытие в Интернете. Было замечено, что даже небольшие погрешности превращались в большие при умножении на большие числа: например, определение доли людей, страдающих редким заболеванием, по отношению к населению Европы, могло привести к неверной оценке количества больных.
- *7 ноября 1994 года.* Газета «Electronic Engineering Times» поместила эту историю на первой странице, и вскоре ее подхватили другие газеты.
- *22 ноября 1994 года.* Intel выпустила пресс-релиз, назвав все это «глюком». Процессор Pentium «может допускать ошибки в девятой цифре... Практически все инженеры и финансовые аналитики требуют точности только в четвертой или пятой цифре поле десятичной точки. Похоже, что пользователям электронных таблиц и текстовых процессоров вообще не о чем беспокоиться... Возможно, эта проблема могла коснуться лишь нескольких десятков человек. До сих пор мы услышали о ней только от одного человека... Эта проблема должна вызывать озабоченность [только] у математиков-теоретиков (которые приобрели компьютеры Pentium до лета этого года)». Многих раздражало то, что клиентам предлагалось написать заявление в Intel о том, в каком качестве используется их компьютер, а затем компания решит, нужен им для этого новый Pentium без ошибки в операции деления или нет.
- *5 декабря 1994 года.* Intel заявила, что этот дефект проявляется для обычного пользователя электронных таблиц раз в 27 000 лет. Intel предположил, что пользователь проводит в день 1000 операций деления, и умножил это число на частоту проявления ошибки при условии, что числа с плавающей точкой используются нечасто, получился один случай на 9 млрд, или одна ошибка за 27 000 лет. Пользователи стали успокаиваться, несмотря на то что Intel не посчитал нужным объяснить, почему обычный пользователь будет обращаться к числам с плавающей точкой довольно редко.
- *12 декабря 1994 года.* Научно-исследовательский отдел компании IBM поставил под сомнение частоту проявления ошибки, вычисленную компанией Intel (их статью можно найти по адресу www.mkp.com/books_catalog/cod/links.htm). Специалисты IBM заявили, что обычные электронные таблицы, занимающиеся пересчетом результатов 15 минут в день, могут выдавать ошибки, связанные

с дефектом процессора Pentium каждые 24 дня. Они предполагали, что 5000 делений в секунду за 15 минут дают 4,2 миллиона делений в день, и не согласились с предположением о случайном распределении чисел, вычислив возможность ошибки как одну на 100 млн. В результате этого IBM немедленно прекратила поставки своих персональных компьютеров на основе процессоров Pentium. Скандал вокруг компании Intel разгорелся с новой силой.

- *21 декабря 1994 года.* Intel выпускает следующий пресс-релиз, подписанный президентом компании, главным исполнительным директором и председателем правления:

  «Мы в компании желаем принести искренние извинения за наше отношение к ставшему недавно известным недостатку процессора Pentium. Эмблема Intel Inside означает, что ваш компьютер имеет непревзойденный по качеству и производительности микропроцессор. Над доказательством истинности данного утверждения упорно трудятся тысячи сотрудников компании Intel. Но идеальных микропроцессоров не существует. В компании Intel продолжают верить тому, что в данном случае совершенно незначительная проблема приобрела особое значение. Хотя компания Intel настаивает на высоком качестве текущей версии процессора Pentium, мы отдаем себе отчет в имеющейся у многих пользователей озабоченности. Мы хотим устранить все сомнения. Intel обменяет текущую версию процессора Pentium на обновленную, с устраненным дефектом деления чисел с плавающей точкой, любому владельцу, запросившему такой обмен, совершенно бесплатно в любое время на протяжении всего срока эксплуатации его компьютера».

Аналитики оценили, что этот отзыв продукции обойдется компании Intel в 500 млн долларов, и инженеры компании Intel не получили в тот год рождественских бонусов.

У этой истории есть несколько поучительных моментов, касающихся всех нас. Насколько дешевле было бы устранить дефект в июле 1994 года? Во сколько обошлось восстановление репутации компании Intel? И что такое корпоративная ответственность в выявлении ошибок в таком широко используемом и серьезном продукте, как микропроцессор?

В апреле 1997 года была обнаружена еще одна ошибка при работе с числами с плавающей точкой в микропроцессорах Pentium Pro и Pentium II. Когда инструкция сохранения числа с плавающей точкой в виде целого числа (`fist`, `fistp`) сталкивается с отрицательным числом с плавающей точкой, которое не может поместиться в 16- или 32-разрядное слово после преобразования в целое число, эти команды устанавливали неверный бит в слово состояния FPO (исключение по точности вместо исключения по недопустимой операции). К чести компании Intel, на этот раз она публично оповестила о существовании ошибки и предложила программное средство для ее обхода, то есть поступила совершенно иначе, чем в 1994 году.

3.9. Заключительные комментарии

Побочным эффектом компьютера с программой, хранящейся в памяти, является то, что у битовых комбинаций нет никаких предопределенных значений. Одни и те же комбинации битов могут представлять собой целое число со знаком, целое число без знака, число с плавающей точкой, инструкцию и т. д. Значение слова определяется инструкцией, которая с ним работает.

Компьютерная арифметика отличается от арифметики с карандашом и бумагой необходимым ограничением в точности. Это ограничение может привести к недопустимым операциям, если вычисленные значения больше или меньше предопределенных ограничений. Подобные аномалии, которые называются «переполнением» или «потерей значимости», могут приводить к исключениям или прерываниям, аварийным событиям, подобным незапланированным вызовам подпрограмм. Более подробно исключения рассматриваются в главе 4.

У арифметики с плавающей точкой есть дополнительные сложности с приближением к вещественным числам, требующие особых мер, обеспечивающих выбор представления компьютерного числа, наиболее близкого к фактическому числу. Проблемы погрешности и ограниченного представления являются частью области численного анализа. Недавний переход к параллельным вычислениям возродит интерес к численному анализу, поскольку решения, которые долгое время считались надежными на последовательных компьютерах, должны быть пересмотрены при поиске самых быстродействующих алгоритмов для параллельных компьютеров, которые продолжали бы выдавать правильный результат.

С годами компьютерная арифметика стала практически полностью подчинена определенным стандартам, существенно улучшив переносимость программ. В подавляющем большинстве продаваемых в настоящее время компьютеров используется целочисленная арифметика с дополнением до двух и двоичная арифметика с плавающей точкой, соответствующая стандарту IEEE 754. К примеру, каждый настольный компьютер, проданный со времени выхода первого издания этой книги, соответствовал именно этим соглашениям по представлению чисел.

Наряду с рассмотрением компьютерной арифметики в данной главе дается описание большого количества новых инструкций, входящих в набор MIPS. Неясным остается только момент сравнения инструкций, рассмотренных в данной главе, с инструкциями, выполняемыми микропроцессорами MIPS, а также с инструкциями, воспринимаемыми ассемблерными программами MIPS. Прояснить ситуацию помогут две следующие таблицы.

В табл. 3.9 приводится список инструкций MIPS, рассмотренных в данной главе и в главе 2. Инструкции, показанные в левой части таблицы, называются *основными инструкциями MIPS* (MIPS core). Инструкции, показанные в правой части, называются *основными арифметическими инструкциями MIPS* (MIPS arithmetic core). В левой части табл. 3.10 показаны инструкции процессора MIPS, отсутствующие в табл. 3.9. Полный набор инструкций оборудования называется *MIPS-32*. В правой части табл. 3.10 показаны инструкции, воспринимаемые программами ассемблера и не являющиеся частью набора MIPS-32. Этот набор инструкций называется *псевдо-MIPS*.

Таблица 3.9. Набор инструкций MIPS. В этой книге основное внимание уделяется инструкциям, показанным в левой части таблицы

Основные инструкции MIPS	Название	Формат	Основные арифметические инструкции MIPS	Название	Формат
add (сложение)	add	R	multiply (умножение)	mult	R
add immediate (непосредственное сложение)	addi	I	multiply unsigned (беззнаковое умножение)	multu	R
add unsigned (беззнаковое сложение)	addu	R	divide (деление)	div	R
add immediate unsigned (беззнаковое непосредственное сложение)	addiu	I	divide unsigned (беззнаковое деление)	divu	R
subtract (вычитание)	sub	R	move from Hi (перемещение из Hi)	mfhi	R
subtract unsigned (беззнаковое вычитание)	subu	R	move from Lo (перемещение из Lo)	mflo	R
AND (И)	AND	R	move from system control (EPC) (перемещение из системного управления)	mfc0	R
AND immediate (И с непосредственным значением)	ANDi	I	floating-point add single (FP-сложение с одинарной точностью)	add.s	R
OR (ИЛИ)	OR	R	floating-point add double (FP-сложение с двойной точностью)	add.d	R
OR immediate (ИЛИ с непосредственным значением)	Ori	I	floating-point subtract single (FP-вычитание с одинарной точностью)	sub.s	R
NOR (not or — НЕ ИЛИ)	NOR	R	floating-point subtract double (FP-вычитание с двойной точностью)	sub.d	R
shift left logical (логический сдвиг влево)	sll	R	floating-point multiply single (FP-умножение с одинарной точностью)	mul.s	R
shift right logical (логический сдвиг вправо)	srl	R	floating-point multiply double (FP-умножение с двойной точностью)	mul.d	R
load upper immediate (загрузка непосредственного значения в верхние разряды)	lui	I	floating-point divide single (FP-деление с одинарной точностью)	div.s	R
load word (загрузка слова)	lw	I	floating-point divide double (FP-деление с двойной точностью)	div.d	R
store word (сохранение слова)	sw	I	load word to floating-point single (загрузка слова с одинарной точностью в FP-регистр)	lwc1	I

Основные инструкции MIPS	Название	Формат	Основные арифметические инструкции MIPS	Название	Формат
load halfword unsigned (загрузка полуслова без знака)	lhu	I	store word to floating-point single (сохранение FP-слова с одинарной точностью)	swc1	I
store halfword (сохранение полуслова)	sh	I	load word to floating-point double (загрузка слова с двойной точностью в FP-регистр)	ldc1	I
load byte unsigned (загрузка байта без знака)	lbu	I	store word to floating-point double (сохранение FP-слова с двойной точностью)	sdc1	I
store byte (сохранение байта)	sb	I	branch on floating-point true (условный переход, если FP-сравнение истинно)	bc1t	I
load linked (atomic update) (связанная загрузка при атомарном обновлении)	ll	I	branch on floating-point false (условный переход, если FP-сравнение ложно)	bc1f	I
store cond. (atomic update) (условное сохранение при атомарном обновлении)	sc	I	floating-point compare single (*x* = eq, neq, lt, le, gt, ge) (FP-сравнение с одинарной точностью)	c.*x*.s	R
branch on equal (переход при равенстве)	beq	I	floating-point compare double (*x* = eq, neq, lt, le, gt, ge) (FP-сравнение с двойной точностью)	c.*x*.d	R
branch on not equal (переход при неравенстве)	bne	I			
jump (переход)	j	J			
jump and link (переход и ссылка)	jal	J			
jump register (переход по регистру)	jr	R			
set less than (установка, если меньше чем)	slt	R			
set less than immediate (установка, если меньше чем непосредственное значение)	slti	I			
set less than unsigned (установка, если меньше чем без знака)	sltu	R			
set less than immediate unsigned (установка, если меньше чем непосредственное значение, без знака)	sltiu	I			

Таблица 3.10. Наборы остальных инструкций MIPS-32 и псевдо-MIPS. *f* означает одинарную (s) или двойную (d) точность FP-инструкций, *s* означает версии со знаком — signed и без знака — unsigned (u). В MIPS-32 также имеются FP-инструкции для умножения и сложения-вычитания (madd.*f*/msub.*f*), округления в большую сторону — ceiling (ceil.*f*), округления с отбрасыванием —truncate (trunc.*f*), округления — round (round.*f*) и обратной величины — reciprocal (recip.*f*). Подчеркивание показывает букву, включаемую для представления указываемого типа данных

Остальные инструкции MIPS-32	Название	Формат	Псевдо-MIPS	Название	Формат
exclusive or (*rs* ⊕ *rt*) (исключающее ИЛИ)	xor	R	absolute value (абсолютное значение)	abs	rd, rs
exclusive or immediate (исключающее ИЛИ с непосредственным значением)	xori	I	negate (*signed или unsigned*) (смена знака для чисел со знаком или без знака)	negs	rd, rs
shift right arithmetic (арифметический сдвиг вправо)	sra	R	rotate left (вращение влево)	rol	rd, rs, rt
shift left logical variable (переменный логический сдвиг влево)	sllv	R	rotate right (вращение вправо)	ror	rd, rs, rt
shift right logical variable (переменный логический сдвиг вправо)	srlv	R	multiply and don't check oflw (*signed или uns.*) (умножение без проверки переполнения со знаком или без знака)	muls	rd, rs, rt
shift right arithmetic variable (переменный арифметический сдвиг вправо)	srav	R	multiply and check oflw (*signed или uns.*) (умножение с проверкой переполнения со знаком или без знака)	mulos	rd, rs, rt
move to Hi (перемещение в Hi)	mthi	R	divide and check overflow (деление с проверкой переполнения)	div	rd, rs, rt
move to Lo (перемещение в Lo)	mtlo	R	divide and don't check overflow (деление без проверки переполнения)	divu	rd, rs, rt
load halfword (загрузка полуслова)	lh	I	remainder (*signed или unsigned*) (получение остатка со знаком или без знака)	rems	rd, rs, rt
load byte (загрузка байта)	lb	I	load immediate (загрузка непосредственного значения)	li	rd, imm
load word left (*unaligned*) (загрузка слова влево, без выравнивания)	lwl	I	load address (загрузка адреса)	la	rd, addr

Остальные инструкции MIPS-32	Название	Формат	Псевдо-MIPS	Название	Формат
load word right (*unaligned*) (загрузка слова вправо, без выравнивания)	lwr	I	load double (загрузка двойного слова)	ld	rd, addr
store word left (*unaligned*) (сохранение слова влево, без выравнивания)	swl	I	store double (сохранение двойного слова)	sd	rd, addr
store word right (*unaligned*) (сохранение слова вправо, без выравнивания)	swr	I	unaligned load word (загрузка слова без выравнивания)	ulw	rd, addr
load linked (*atomic update*) (связанная загрузка при атомарном обновлении)	ll	I	unaligned store word (сохранение слова без выравнивания)	usw	rd, addr
store cond. (*atomic update*) (условное сохранение при атомарном обновлении)	sc	I	unaligned load halfword (*signed или uns.*) (загрузка полуслова без выравнивания)	ulhs	rd, addr
move if zero (перемещение при нулевом значении)	movz	R	unaligned store halfword (сохранение полуслова без выравнивания)	ush	rd, addr
move if not zero (перемещение при ненулевом значении)	movn	R	branch (условный переход)	b	Label
multiply and add (*S или uns.*) (умножение и сложение)	madds	R	branch on equal zero (условный переход, если равно нулю)	beqz	rs, L
multiply and subtract (*S или uns.*) (умножение и вычитание)	msubs	I	branch on compare (*signed или unsigned*) (x = lt, le, gt, ge) (условный переход по результату сравнения)	bxs	rs, rt, L
branch on ⩾ zero and link (условный переход по ⩾ нулю со ссылкой)	bgezal	I	set equal (установка, если равно)	seq	rd, rs, rt
branch on < zero and link (условный переход по < нуля со ссылкой)	bltzal	I	set not equal (установка, если не равно)	sne	rd, rs, rt
jump and link register (переход по регистру со ссылкой)	jalr	R	set on compare (*signed или unsigned*) (x = lt, le, gt, ge) (установка по результату сравнения)	sxs	rd, rs, rt
branch compare to zero (условный переход при сравнении с нулем)	bxz	I	load to floating point (*s или d*) (загрузка в FP-формате с одинарной или двойной точностью)	l.f	rd, addr

продолжение ⅋

Таблица 3.10 (*продолжение*)

Остальные инструкции MIPS-32	Название	Формат	Псевдо-MIPS	Название	Формат
branch compare to zero likely (возможный условный переход при сравнении с нулем)	bxzl	I	store from floating point (*s или d*) (сохранение из FP-формата с одинарной или двойной точностью)	s.*f*	rd, addr
(x = lt, le, gt, ge)					
branch compare reg likely (возможный условный переход при сравнении с регистром)	bxl	I			
trap if compare reg (прерывание при сравнении с регистром)	tx	R			
trap if compare immediate (прерывание при сравнении с непосредственным значением)	txi	I			
(x = lt, le, gt, ge)					
return from exception (возвращение из исключения)	rfe	R			
system call (системный вызов)	syscall	I			
break (*cause exception*) (выход с возникновением исключения)	break	I			
move from FP to integer (перемещение из FP в целое число)	mfc1	R			
move to FP from integer (перемещение в FP из целого числа)	mtc1	R			
FP move (*s или d*) (FP-перемещение с одинарной или двойной точностью)	mov.*f*	R			
FP move if zero (*s или d*) (FP-перемещение с одинарной или двойной точностью при нулевом значении)	movz.*f*	R			
FP move if not zero (*s или d*) (FP-перемещение с одинарной или двойной точностью при ненулевом значении)	movn.*f*	R			
FP square root (*s или d*) (извлечение квадратного корня из FP-числа с одинарной или двойной точностью)	sqrt.*f*	R			
FP absolute value (*s или d*) (получение абсолютного значения из FP-числа с одинарной или двойной точностью)	abs.*f*	R			

Остальные инструкции MIPS-32	Название	Формат	Псевдо-MIPS	Название	Формат
FP negate (*s* или *d*) (смена знака FP-числа с одинарной или двойной точностью)	neg.*f*	R			
FP convert (*w*, *s*, или *d*) (конвертирование FP-числа в слово с одинарной или двойной точностью)	cvt.*f*.*f*	R			
FP compare un (*s* или *d*) (FP-сравнение с одинарной или двойной точностью)	c.xn.*f*	R			

В табл. 3.11 представлены MIPS-инструкции, широко используемые в контрольных задачах SPEC2006 для работы с целыми числами и с числами с плавающей точкой. Перечислены все инструкции, отвечающие хотя бы за 0,3% всех выполняемых команд.

Таблица 3.11. Частота MIPS-инструкций для контрольных задач SPEC2006, работающих с целыми числами и с числами с плавающей точкой. В таблицу включены все инструкции, составляющие не менее 1% используемых инструкций. Псевдоинструкции перед выполнением преобразовываются в MIPS-32 и поэтому здесь не показаны

Основные инструкции MIPS	Название	Целочисл.	С плав. точкой	Основные арифм. инстр. + MIPS-32	Название	Целочисл.	С пл. точкой
add (сложение)	add	0,0%	0,0%	FP add double (FP-сложение с двойной точностью)	add.d	0,0%	10,6%
add immediate (непосредственное сложение)	addi	0,0%	0,0%	FP subtract double (FP-вычитание с двойной точностью)	sub.d	0,0%	4,9%
add unsigned (беззнаковое сложение)	addu	5,2%	3,5%	FP multiply double (FP-умножение с двойной точностью)	mul.d	0,0%	15,0%
add immediate unsigned (беззнаковое непосредственное сложение)	addiu	9,0%	7,2%	FP divide double (FP-деление с двойной точностью)	div.d	0,0%	0,2%
subtract unsigned (беззнаковое вычитание)	subu	2,2%	0,6%	FP add single (FP-сложение с одинарной точностью)	add.s	0,0%	1,5%

продолжение ☞

Таблица 3.11 (*продолжение*)

Основные инструкции MIPS	Название	Целочисл.	С плав. точкой	Основные арифм. инстр. + MIPS-32	Название	Целочисл.	С пл. точкой
AND (И)	AND	0,2%	0,1%	FP subtract single (FP-вычитание с одинарной точностью)	sub.s	0,0%	1,8%
AND immediate (И с непосредственным значением)	ANDi	0,7%	0,2%	FP multiply single (FP-умножение с одинарной точностью)	mul.s	0,0%	2,4%
OR (ИЛИ)	OR	4,0%	1,2%	FP divide single (FP-деление с одинарной точностью)	div.s	0,0%	0,2%
OR immediate (ИЛИ с непосредственным значением)	ORi	1,0%	0,2%	load word to FP double (загрузка слова в FP-формате с двойной точностью)	l.d	0,0%	17,5%
NOR (НЕ ИЛИ)	NOR	0,4%	0,2%	store word to FP double (сохранение слова в FP-формате с двойной точностью)	s.d	0,0%	4,9%
shift left logical (логический сдвиг влево)	sll	4,4%	1,9%	load word to FP single (загрузка слова в FP-формате с одинарной точностью)	l.s	0,0%	4,2%
shift right logical (логический сдвиг вправо)	srl	1,1%	0,5%	store word to FP single (сохранение слова в FP-формате с одинарной точностью)	s.s	0,0%	1,1%
load upper immediate (загрузка непосредственного значения в верхние разряды)	lui	3,3%	0,5%	branch on floating-point true (условный переход при истинности условия в FP-формате)	bc1t	0,0%	0,2%
load word (загрузка слова)	lw	18,6%	5,8%	branch on floating-point false (условный переход при ложности условия в FP-формате)	bc1f	0,0%	0,2%

Основные инструкции MIPS	Название	Целочисл.	С плав. точкой	Основные арифм. инстр. + MIPS-32	Название	Целочисл.	С пл. точкой
store word (сохранение слова)	sw	7,6%	2,0%	floating-point compare double (сравнение в FP-формате с двойной точностью)	c.x.d	0,0%	0,6%
load byte (загрузка байта)	lbu	3,7%	0,1%	multiply (умножение)	mul	0,0%	0,2%
store byte (сохранение байта)	sb	0,6%	0,0%	shift right arithmetic (арифметический сдвиг вправо)	sra	0,5%	0,3%
branch on equal (zero) (переход при равенстве)	beq	8,6%	2,2%	load half (загрузка полуслова)	lhu	1,3%	0,0%
branch on not equal (zero) (переход при неравенстве)	bne	8,4%	1,4%	store half (сохранение полуслова)	sh	0,1%	0,0%
jump and link (переход со ссылкой)	jal	0,7%	0,2%	jump register (переход по регистру)	jr	1,1%	0,2%
jump register (переход по регистру)	jr	1,1%	0,2%				
set less than (установка при меньше чем)	slt	9,9%	2,3%				
set less than immediate (установка при меньше чем непосредственное значение)	slti	3,1%	0,3%				
set less than unsigned (установка при меньше чем, без знака)	sltu	3,4%	0,8%				
set less than imm. uns. (установка при меньше чем непосредственное значение, без знака)	sltiu	1,1%	0,1%				

Судя по данным, показанным в следующей таблице, несмотря на то что программисты и создатели компиляторов могут расширить свой выбор за счет инструкций MIPS-32, следует заметить, что основные инструкции MIPS доминируют среди целочисленных инструкций, выполняемых в контрольных задачах SPEC2006, и те же основные инструкции и основные арифметические инструкции доминируют среди инструкций с плавающей точкой, выполняемых в контрольных задачах SPEC2006.

Поднабор инструкций	Целочисленные	С плавающей точкой
Основные инструкции MIPS	98%	31%
Основные арифметические инструкции MIPS	2%	66%
Остальные инструкции MIPS-32	0%	3%

Чтобы упростить рассмотрение вопросов конструирования компьютеров, во всей остальной части книги основное внимание будет уделено основным инструкциям MIPS — набору целочисленных инструкций, исключая умножение и деление. Как видите, основные инструкции MIPS относятся к наиболее популярным инструкциям; будьте уверены, что освоение принципов устройства компьютера, на котором запускаются основные инструкции MIPS, даст вам достаточные базовые знания для понимания принципов работы более сложных компьютеров.

3.10. Упражнения

> Никогда не сдавайтесь, никогда не сдавайтесь, никогда, никогда, никогда, ни в чем, ни в большом, ни в малом, ни в серьезных делах, ни в пустяках.
>
> *Уинстон Черчилль, школа Хэрроу*

Предоставлены Мэтью Фарренсом (Matthew Farrens), Калифорнийский университет в Дэвисе.

Упражнение 3.1

В книге показано, как складывать и вычитать двоичные и десятичные числа. Но при работе с компьютерами были весьма популярны и некоторые другие системы счисления. Среди них была и восьмеричная (по основанию 8) система счисления. В следующей таблице приведены пары восьмеричных чисел.

	A	B
а	5323	2275
б	0147	3457

3.1.1 [5] <3.2> Какова сумма A и B, если они представляют восьмеричными числами 12-разрядные двоичные числа без знака? Результат должен быть записан в восьмеричном виде. Покажите весь процесс решения.

3.1.2 [5] <3.2> Какова сумма A и B, если они представляют восьмеричными числами 12-разрядные двоичные числа, сохраненные в формате значений со знаком? Результат должен быть записан в восьмеричном виде. Покажите весь процесс решения.

3.1.3 [10] <3.2> Преобразуйте A в десятичное число, предположив, что это число без знака. Повторите упражнение, предположив, что это число хранится в формате значения со знаком. Покажите весь процесс решения.

В таблице также показаны пары восьмеричных чисел:

	A	B
а	2762	2032
б	2646	1066

3.1.4 [5] <3.2> Каким будет результат вычисления выражения A – B, если в нем представлены восьмеричными числами 12-разрядные двоичные числа без знака? Результат должен быть записан в восьмеричном виде. Покажите весь процесс решения.

3.1.5 [5] <3.2> Каким будет результат вычисления выражения A – B, если в нем представлены восьмеричными числами 12-разрядные двоичные числа, сохраненные в формате значений со знаком? Результат должен быть записан в восьмеричном виде. Покажите весь процесс решения.

3.1.6 [10] <3.2> Преобразуйте A в двоичное число. Что делает основание 8 (восьмеричное) привлекательным для системы счисления, представляющей значения в компьютере?

Упражнение 3.2

Для представления значений в компьютерах также часто используется шестнадцатеричная (по основанию 16) система счисления. Эта система приобрела более высокую популярность по сравнению с восьмеричной. В таблице показаны пары шестнадцатеричных чисел:

	A	B
а	0034	0017
б	BA1D	3617

3.2.1 [5] <3.2> Какова сумма A и B, если они представляют шестнадцатеричными числами 16-разрядные двоичные числа без знака? Результат должен быть записан в шестнадцатеричном виде. Покажите весь процесс решения.

3.2.2 [5] <3.2> Какова сумма A и B, если они представляют шестнадцатеричными числами 16-разрядные двоичные числа в формате значений со знаком? Результат должен быть записан в шестнадцатеричном виде. Покажите весь процесс решения.

3.2.3 [10] <3.2> Преобразуйте A в десятичное число, предположив, что это число без знака. Повторите упражнение, предположив, что это число хранится в формате значения со знаком. Покажите весь процесс решения.

В таблице также показаны пары шестнадцатеричных чисел:

	A	B
а	BA7C	241A
б	AADF	47BE

3.2.4 [5] <3.2> Каким будет результат вычисления выражения A – B, если в нем представлены шестнадцатеричными числами 16-разрядные двоичные числа без знака? Результат должен быть записан в шестнадцатеричном виде. Покажите весь процесс решения.

3.2.5 [5] <3.2> Каким будет результат вычисления выражения A – B, если в нем представлены шестнадцатеричными числами 16-разрядные двоичные числа, сохраненные в формате значений со знаком? Результат должен быть записан в шестнадцатеричном виде. Покажите весь процесс решения.

3.2.6 [10] <3.2> Преобразуйте A в двоичное число. Что делает основание 16 (шестнадцатеричное) привлекательным для системы счисления, представляющей значения в компьютере?

Упражнение 3.3

Переполнение возникает в том случае, если результат слишком велик, чтобы быть точно представленным с помощью заданного конечного размера слова. Потеря значимости возникает в том случае, когда число слишком мало, чтобы быть правильно представленным, например, при получении отрицательного результата при беззнаковых арифметических вычислениях. (Тот случай, когда положительный результат получается путем сложения двух отрицательных целых чисел, также считается многими потерей значимости, но в данной книге он рассматривается как переполнение). В следующей таблице показаны пары десятичных чисел.

	A	B
а	69	90
б	102	44

3.3.1 [5] <3.2> Предположим, что A и B являются десятичными числами, представленными 8-разрядными двоичными целыми числами без знака. Вычислите результат выражения A – B. Возникает при этом переполнение или потеря значимости либо этих событий удается избежать?

3.3.2 [5] <3.2> Предположим, что A и B являются десятичными числами, представленными 8-разрядными двоичными целыми числами, сохраненными в формате значений со знаком. Вычислите результат выражения A + B. Возникает при этом переполнение или потеря значимости либо этих событий удается избежать?

3.3.3 [5] <3.2> Предположим, что A и B являются десятичными числами, представленными 8-разрядными двоичным целыми числами, сохраненными в формате

значений со знаком. Вычислите результат выражения A – B. Возникает при этом переполнение или потеря значимости либо этих событий удается избежать?

В таблице также показаны пары десятичных чисел:

	A	B
а	200	103
б	247	237

3.3.4 [10] <3.2> Предположим, что A и B являются десятичными числами, представленными 8-разрядными двоичными целыми числами со знаком, сохраненными в формате дополнения до двух. Вычислите результат выражения A + B, используя арифметику с насыщением. Результат должен быть записан в десятичном виде. Покажите весь процесс решения.

3.3.5 [10] <3.2> Предположим, что A и B являются десятичными числами, представленными 8-разрядными двоичными целыми числами со знаком, сохраненными в формате дополнения до двух. Вычислите результат выражения A – B, используя арифметику с насыщением. Результат должен быть записан в десятичном виде. Покажите весь процесс решения.

3.3.6 [10] <3.2> Предположим, что A и B являются десятичными числами, представленными 8-разрядными двоичными целыми числами без знака. Вычислите результат выражения A + B, используя арифметику с насыщением. Результат должен быть записан в десятичном виде. Покажите весь процесс решения.

Упражнение 3.4

Давайте более пристально посмотрим на умножение. Воспользуемся числами из таблицы.

	A	B
а	50	23
б	66	04

3.4.1 [20] <3.3> На основе таблицы, подобной той, что показана на рис. 3.5, вычислите произведение восьмеричных чисел A и B, представленных в виде 6-разрядных целых чисел без знака, используя оборудование, показанное на рис. 3.2. Покажите содержимое каждого регистра при каждом шаге вычисления.

3.4.2 [20] <3.3> На основе таблицы, подобной той, что показана на рис. 3.5, вычислите произведение шестнадцатеричных чисел A и B, представленных в виде 8-разрядных целых чисел без знака, используя оборудование, показанное на рис. 3.4. Покажите содержимое каждого регистра при каждом шаге вычисления.

3.4.3 [60] <3.3> Напишите программу на языке ассемблера MIPS, вычисляющую произведение целых чисел A и B без знака и использующую подходы, показанные на рис. 3.2.

В таблице показаны пары восьмеричных чисел.

	A	B
а	54	67
б	30	07

3.4.4 [30] <3.3> При умножении чисел со знаком одним из способов получения правильного ответа является преобразование множителя и множимого в положительные числа, сохранение исходных знаков, а затем соответствующая корректировка окончательного значения. На базе таблицы, подобной той, что показана на рис. 3.5, вычислите произведение чисел A и B, используя оборудование, показанное на рис. 3.2. Покажите содержимое каждого регистра при каждом шаге вычисления и включите шаг, необходимый для получения результата с правильным знаком. Допустим, что A и B хранятся в двоичном 6-разрядном формате значений со знаком.

3.4.5 [30] <3.3> При сдвиге значения регистра на один разряд вправо есть несколько путей решения вопроса о том, каким должен быть новый вводимый разряд. Он может всегда быть нулем или всегда быть единицей, или вводимый разряд может быть тем самым разрядом, который был вытолкнут из правой стороны (при этом сдвиг превращается в циклический), или же просто может сохраниться значение самого левого разряда (при так называемом арифметическом сдвиге вправо, поскольку в этом разряде хранится знак сдвигаемого числа). Используя таблицу, показанную на рис. 3.5, вычислите произведение двоичных 6-разрядных чисел A и B с дополнением до двух, получаемое при использовании оборудования, показанного на рис. 3.4. Сдвиг вправо должен быть осуществлен с использованием операции арифметического сдвига вправо. Учтите, что для этого описанный в тексте алгоритм нужно немного скорректировать, в частности, при отрицательном множителе все может происходить иначе. Подробности можно найти в Интернете. Покажите содержимое каждого регистра при каждом шаге вычисления.

3.4.6 [60] <3.3> Напишите программу на языке ассемблера MIPS, которая вычисляет произведение целых чисел A и B со знаком. Сообщите, используются для этого подходы, заданные в упражнении 3.4.4 или в упражнении 3.4.5.

Упражнение 3.5

По многим причинам появляется желание создать более быстродействующее устройство для умножения. Для достижения этой цели использовалось множество различных подходов. В следующей таблице столбец A представляет ширину целого числа в двоичных разрядах, а столбец B — количество единиц времени (time units — tu), затрачиваемых на шаг операции.

	A (ширина в двоичных разрядах)	B (количество единиц времени)
а	4	3
б	32	7

3.5.1 [10] <3.3> Вычислите время, необходимое для выполнения умножения с использованием подходов, показанных на рис. 3.2 и 3.3, если целое число имеет

ширину A разрядов и каждый шаг операции требует B единиц времени. Учтите, что на шаге 1, *a* всегда производится сложение — либо с множимым, либо с нулем. Также учтите, что регистры уже инициализированы (вы подсчитываете только продолжительность самого цикла умножения). Если это делается аппаратным способом, то сдвиг множимого и множителя может быть произведен одновременно. Если это делается программным способом, эти сдвиги должны производиться один за другим. Сделайте расчет для обоих случаев.

3.5.2 [10] <3.3> Вычислите время, необходимое для выполнения умножения с использованием подхода, описанного в тексте (последовательность из 31 сумматора), если целое число имеет ширину A разрядов, а сумматор затрачивает B единиц времени.

3.5.3 [20] <3.3> Вычислите время, необходимое для выполнения умножения с использованием подхода, показанного на рис. 3.6, если целое число имеет ширину A разрядов, а сумматор затрачивает B единиц времени.

Упражнение 3.6

В этом упражнении будут рассмотрены два других способа повышения производительности умножения, основанные в первую очередь на осуществлении большего количества сдвигов и меньшего количества арифметических операций. В таблице показана пара шестнадцатеричных чисел:

	A	B
а	24	c97
б	41	18

3.6.1 [20] <3.3> Как рассматривалось в тексте, один из способов повышения производительности заключается в сдвиге и сложении вместо реального умножения. Поскольку 9 × 6, к примеру, можно переписать как (2 × 2 × 2 + 1) × 6, можно вычислить 9 × 6, сдвигая 6 влево три раза, а затем прибавляя 6 к полученному результату. Покажите наилучший способ вычисления выражения A × B с использованием сдвигов и сложений-вычитаний, при условии, что A и B представлены 8-разрядными двоичными целыми числами без знака.

3.6.2 [20] <3.3> Покажите наилучший способ вычисления выражения A × B с использованием сдвигов и сложений, если A и B представлены 8-разрядными двоичными целыми числами, сохраненными в формате значений со знаком.

3.6.3 [60] <3.3> Напишите программу на языке ассемблера MIPS, выполняющую умножение целых чисел со знаком и использующую сдвиг и сложение, как описывалось в упражнении 3.6.1.

В следующей таблице показаны еще две пары шестнадцатеричных чисел:

	A	B
а	42	36
б	9F	8E

3.6.4 [30] <3.3> Еще одним способом снижения количества арифметических операций, необходимых для выполнения умножения, является применение алгоритма Бута. Этот алгоритм существует уже много лет, и подробности его работы можно найти в Интернете. В основном в нем предполагается, что сдвиг требует меньше времени, чем сложение или вычитание, и этот факт используется для сокращения количества арифметических операций, необходимых для выполнения умножения. Он работает за счет обнаружения череды единиц или нулей и выполнения сдвигов, пока не закончится эта череда. Найдите описание алгоритма и объясните подробно принцип его работы.

3.6.5 [30] <3.3> Покажите пошаговые результаты перемножения чисел A и B, использующего алгоритм Бута, при условии, что A и B являются 8-разрядными двоичными целыми числами с дополнением до двух, хранящимися в шестнадцатеричном формате.

3.6.6 [60] <3.3> Напишите программу на языке ассемблера для выполнения умножения чисел A и B с использованием алгоритма Бута.

Упражнение 3.7

Давайте более подробно рассмотрим деление. В таблице будут использоваться восьмеричные числа:

	A	B
а	50	23
б	25	44

3.7.1 [20] <3.4> Используя таблицу, подобную той, что показана на рис. 3.9, вычислите результат деления A на B с использованием оборудования, показанного на рис. 3.7. Покажите содержимое каждого регистра при каждом шаге, при условии, что A и B представлены 6-разрядными двоичными целыми числами без знака.

3.7.2 [30] <3.4> Используя таблицу, подобную той, что показана на рис. 3.9, вычислите результат деления A на B с использованием оборудования, показанного на рис. 3.10. Покажите содержимое каждого регистра при каждом шаге, при условии, что A и B представлены 6-разрядными двоичными целыми числами без знака. Этот алгоритм требует несколько иного подхода, чем тот, что показан на рис. 3.8. Для выполнения этого упражнения нужно как следует подумать, провести один-два эксперимента или же выйти в Интернет, чтобы определить, как правильно справиться с поставленной задачей. (Подсказка: одно из возможных решений требует использования того факта, что оборудование, показанное на рис. 3.10, подразумевает, что регистр остатка может быть сдвинут в любом направлении).

3.7.3 [60] <3.4> Напишите программу на языке ассемблера для вычисления результата деления A на B с использованием подхода, показанного на рис. 3.7, при условии, что A и B представлены 6-разрядными двоичными целыми числами без знака.

В следующей таблице показаны еще две пары восьмеричных чисел.

	A	B
в	55	24
г	36	51

3.7.4 [30] <3.4> Используя таблицу, подобную той, что показана на рис. 3.9, вычислите результат деления A на B с использованием оборудования, показанного на рис. 3.7. Покажите содержимое каждого регистра при каждом шаге, при условии, что A и B представлены 6-разрядными двоичными целыми числами в формате значений со знаком. Не забудьте включить способ вычисления знака частного и остатка.

3.7.5 [30] <3.4> Используя таблицу, подобную той, что показана на рис. 3.9, вычислите результат деления A на B с использованием оборудования, показанного на рис. 3.10. Покажите содержимое каждого регистра при каждом шаге, при условии, что A и B представлены 6-разрядными двоичными целыми числами в формате значений со знаком. Не забудьте включить способ вычисления знака частного и остатка.

3.7.6 [60] <3.4> Напишите программу на языке ассемблера MIPS для вычисления результата деления A на B с использованием подхода, показанного на рис. 3.10, при условии, что A и B являются целыми числами со знаком.

Упражнение 3.8

На рис. 3.8 показан алгоритм деления с восстановлением остатка, поскольку когда при вычитании делителя из остатка получается отрицательный результат, делитель снова складывается с остатком (восстанавливая, таким образом, свое значение). Но разработаны и другие алгоритмы, исключающие дополнительное сложение. Многочисленные ссылки на эти алгоритмы можно легко найти в Интернете. Исследование этих алгоритмов мы проведем с использованием пар восьмеричных чисел, приведенных в таблице:

	A	B
а	75	12
б	52	37

3.8.1 [30] <3.4> Используя таблицу, подобную той, которая показана на рис. 3.9, вычислите результат деления A на B, используя деление без восстановления остатка. Покажите содержимое каждого регистра при каждом шаге, при условии, что A и B представлены 6-разрядными двоичными целыми числами без знака.

3.8.2 [60] <3.4> Напишите программу на языке ассемблера MIPS, вычисляющую результат деления A на B с использованием деления без восстановления остатка, при условии, что A и B представлены 6-разрядными двоичными целыми числами со знаком (с дополнением до двух).

3.8.3 [60] <3.4> Как можно сравнить производительность деления с восстановлением и без восстановления остатка? Продемонстрируйте такое сравнение, показав количество шагов, необходимых для вычисления результата деления A на B с использованием каждого метода, при условии, что A и B представлены 6-разрядными двоичными целыми числами (в формате значения со знаком). При этом допустимо написание программы для выполнения деления с восстановлением и без восстановления остатка.

В следующей таблице приведены еще две пары восьмеричных чисел.

	A	B
а	17	14
б	70	23

3.8.4 [30] <3.4> Используя таблицу, подобную той, которая показана на рис. 3.5, вычислите результат деления A на B, используя бездействующее (nonperforming) деление. Покажите содержимое каждого регистра при каждом шаге, при условии, что A и B представлены 6-разрядными двоичными целыми числами без знака.

3.8.5 [60] <3.4> Напишите программу на языке ассемблера MIPS для вычисления результата деления A на B с использованием бездействующего (nonperforming) деления, при условии, что A и B представлены 6-разрядными двоичными целыми числами со знаком с дополнением до двух.

3.8.6 [60] <3.4> Как можно сравнить производительность деления без восстановления остатка и бездействующего деления? Продемонстрируйте такое сравнение, показав количество шагов, необходимых для вычисления результата деления A на B с использованием каждого метода, при условии, что A и B представлены 6-разрядными двоичными целыми числами (в формате значения со знаком). При этом допустимо написание программы для выполнения бездействующего деления и деления с восстановлением остатка.

Упражнение 3.9

На деление приходится тратить столько времени и сил, что в руководстве по оптимизации для языка Фортран на компьютере CRAY T3E утверждалось, что «лучшей стратегией для деления является его обход при любой предоставляющейся для этого возможности». В данном упражнении рассматриваются следующие разные стратегии для выполнения операций деления.

а	восстанавливающее деление
б	SRT-деление

3.9.1 [30] <3.4> Дайте подробное описание алгоритма.

3.9.2 [60] <3.4> Используйте для описания алгоритма блок-схему (или фрагмент кода на языке высокого уровня).

3.9.3 [60] <3.4> Напишите программу на языке ассемблера MIPS, выполняющую деление с использованием алгоритма.

Упражнение 3.10

В архитектуре фон Неймана группы битов не имеют сами по себе никакого предопределенного значения. То, что представляет собой комбинация разрядов, целиком зависит от того, как она используется. В таблице показана комбинация разрядов, отображенная в шестнадцатеричной записи.

а	0x24A60004
б	0xAFBF0000

3.10.1 [5] <3.5> Какое десятичное число представляет эта комбинация разрядов, если она является целым числом с дополнением до двух? Или если она является целым числом без знака?

3.10.2 [10] <3.5> Если эта комбинация разрядов будет помещена в регистр команд (Instruction Register), то какая MIPS-инструкция будет выполнена?

3.10.3 [10] <3.5> Какое десятичное число представляет эта комбинация разрядов, если она является числом с плавающей точкой? Используйте стандарт IEEE 754.

В следующей таблице приведены десятичные числа:

а	-1609,5
б	-938.8125

3.10.4 [10] <3.5> Запишите двоичное представление десятичного числа, предполагая использование формата IEEE 754 с одинарной точностью.

3.10.5 [10] <3.5> Запишите двоичное представление десятичного числа, предполагая использование формата IEEE 754 с двойной точностью.

3.10.6 [10] <3.5> Запишите двоичное представление десятичного числа, предполагая, что оно сохранено с использованием формата IBM с одинарной точностью (по основанию 16 вместо основания 2, с семью разрядами на экспоненту).

Упражнение 3.11

В стандарте чисел с плавающей точкой IEEE 754 экспонента хранится в формате со «смещением» (которое еще называют «Excess-N», то есть «с избытком N»). Такой подход был избран потому, что хотелось, чтобы набор из всех нулей был по возможности как можно ближе к нулю. Благодаря использованию скрытой единицы, если представлять экспоненту в формате с дополнением до двух, то набор из всех нулей на самом деле будет числом 1! (Вспомним, что любое число в степени нуль равно единице, то есть $1.0^0 = 1$.) Существует множество других аспектов стандарта IEEE 754, призванных повысить скорость функционирования аппаратных блоков, предназначенных для работы с числами с плавающей точкой. Но на многих старых машинах вычисления с плавающей точкой осуществлялись программным способом, и поэтому для них использовались другие форматы. В таблице показаны десятичные числа:

а	$5,00736125 \times 10^5$
б	$-2,691650390625 \times 10^{-2}$

3.11.1 [20] <3.5> Запишите комбинацию двоичных разрядов, используя формат, подобный использовавшемуся в машине DEC PDP-8 (самые левые 12 разрядов являются экспонентой, сохраненной в виде числа с дополнением до двух, а самые правые 24 разряда являются мантиссой, сохраненной в виде числа с дополнением до двух). Скрытые единицы не используются. Прокомментируйте, как отличаются диапазон и точность чисел этой 36-разрядной комбинации от чисел стандарта IEEE 754 с двойной точностью.

3.11.2 [20] <3.5> В изделиях компании NVIDIA имеется «половинный» формат, похожий на стандарт IEEE 754, отличающийся только тем, что в нем используется всего лишь 16 разрядов. Самый левый разряд по-прежнему является знаковым разрядом, экспонента имеет ширину 5 разрядов и хранится в формате excess-15, а мантисса имеет ширину 10 разрядов. Предполагается наличие скрытой единицы. Запишите комбинацию двоичных разрядов, предполагая использование этого формата. Прокомментируйте, как отличаются диапазон и точность чисел этой 16-разрядной комбинации от чисел стандарта IEEE 754 с одинарной точностью.

3.11.3 [20] <3.5> В машинах Hewlett-Packard 2114, 2115 и 2116 используется формат, в котором 16 самых левых разрядов являются мантиссой, хранящейся в формате с дополнением до двух, а за ними следует еще одно 16-разрядное поле, в котором 8 самых левых разрядов являются расширением мантиссы (формируя мантиссу длиной 24 разряда), а самые правые 8 разрядов представляют экспоненту. Но здесь все так интересно было закручено, что экспонента хранилась в формате значения со знаком, где знаковый разряд был в самом правом разряде! Запишите комбинацию двоичных разрядов, предполагая использование этого формата. Скрытые единицы не используются. Прокомментируйте, как отличаются диапазон и точность чисел этой 32-разрядной комбинации от чисел стандарта IEEE 754 с одинарной точностью.

В таблице приведены пары десятичных чисел:

	A	B
а	-1278×10^{3}	$-3{,}90625 \times 10^{-2}$
б	$2{,}3109375 \times 10^{-1}$	$6.391601562 \times 10^{1}$

3.11.4 [20] <3.5> Вычислите на бумаге сумму чисел A и B, при условии, что A и B представлены в 16-разрядном формате NVIDIA, описанном в упражнении 3.11.2 (а также в тексте). Используйте по одному разряду защиты, округления и напоминания и произведите округление до ближайшего четного числа. Покажите все шаги вычисления.

3.11.5 [60] <3.5> Напишите программу на ассемблере MIPS, вычисляющую сумму чисел A и B, при условии, что они хранятся в 16-разрядном формате NVIDIA, описанном в упражнении 3.11.2 (а также в тексте). Используйте по одному разряду защиты, округления и напоминания и произведите округление до ближайшего четного числа.

3.11.6 [60] <3.5> Напишите программу на ассемблере MIPS, вычисляющую сумму чисел A и B, при условии, что они хранятся в формате, описанном в упра-

нии 3.11.1. Теперь измените программу, приспособив ее для вычисления суммы с использование формата, описанного в упражнении 3.11.3. С каким форматом проще работать программисту? Дайте сравнительную характеристику каждого из этих форматов относительно формата IEEE 754. (При этом использование разрядов напоминаний учитывать не нужно.)

Упражнение 3.12

Умножение чисел с плавающей точкой является куда более сложным делом, чем сложение таких чисел, но трудности этих двух операций меркнут в сравнении с трудностями деления.

	A	B
а	$5,66015625 \times 10^{}$	$8,59375 \times 10^{}$
б	$6,18 \times 10^{}$	$5,796875 \times 10^{}$

3.12.1 [30] <3.5> Перемножьте на бумаге числа A и B, при условии, что A и B хранятся в 16-разрядном формате NVIDIA, описанном в упражнении 3.11.2 (а также в тексте). Используйте по одному разряду защиты, округления и напоминания и произведите округление до ближайшего четного числа. Покажите все шаги вычисления; тем не менее умножение можно выполнить в формате, удобном для чтения, как это сделано в примере в тексте, вместо использования технологии, описанной в упражнениях 3.4–3.6. Покажите, возникает ли при вычислениях переполнение или потеря значимости. Запишите ответ в виде 16-разрядной двоичной комбинации, а также в виде десятичного числа. Настолько точен полученный результат? Каким он представляется в сравнении с числом, полученным при выполнении умножения на калькуляторе?

3.12.2 [60] <3.5> Напишите программу на языке ассемблера MIPS, перемножающую числа A и B, при условии, что они хранятся в формате IEEE 754. Покажите, возникает ли при вычислениях переполнение или потеря значимости. (Не забудьте, что формат IEEE 754 предполагает использование по одному разряду защиты, округления и напоминания и округление чисел до ближайшего четного числа.)

3.12.3 [60] <3.5> Напишите программу на языке ассемблера MIPS, перемножающую числа A и B, при условии, что они хранятся в формате, описанном в упражнении 3.11.1. Как изменить программу, чтобы вычислить произведение, при условии использования формата, описанного в упражнении 3.11.3? С каким форматом проще работать программисту? Дайте сравнительную характеристику каждого из этих форматов относительно формата IEEE 754. (При этом использование разрядов напоминаний учитывать не нужно.)

В таблице показаны еще две пары десятичных чисел:

	A	B
а	$3,264 \times 10^{3}$	$6,52 \times 10^{2}$
б	$-2,27734375 \times 10^{6}$	$1,154375 \times 10^{2}$

3.12.4 [30] <3.5> Разделите на бумаге число A на число B. Покажите все шаги, необходимые для получения ответа. Предположите наличие разрядов защиты, округления и напоминания и воспользуйтесь ими в случае необходимости. Запишите итоговый ответ в 16-разрядном двоичном формате с плавающей точкой и в десятичном формате и сравните десятичный результат с тем, который будет получен на калькуляторе.

Тест производительности Livermore Loops является набором базовых компонентов характерных интенсивных вычислений, взятых из научных программ Ливерморской национальной лаборатории им. Э. Лоуренса. В таблице показаны два базовых компонента из этого набора, их описание можно найти по адресу http://www.netlib.org/benchmark/livermore:

а	Livermore Loop 1
б	Livermore Loop 7

3.12.5 [60] <3.5> Запишите цикл на языке ассемблера MIPS.

3.12.6 [60] <3.5> Дайте подробное описание одной из методик выполнения деления чисел с плавающей точкой на цифровом компьютере. Не забудьте привести ссылки на используемые источники информации.

Упражнение 3.13

Операции, выполняемые над целыми числами с фиксированной точкой, ведут себя предсказуемо — для них соблюдаются переместительные, ассоциативные и распределительные законы. Но все эти законы не всегда работают с числами с плавающей точкой. Сначала посмотрим на ассоциативный закон. В таблице показан набор десятичных чисел:

	A	B	C
а	$-1{,}6360 \times 10^{4}$	$1{,}6360 \times 10^{4}$	$1{,}0 \times 10^{-}$
б	$2{,}865625 \times 10^{}$	$4{,}140625 \times 10^{}$	$1{,}2140625 \times 10^{}$

3.13.1 [20] <3.2, 3.5, 3.6> Вычислите на бумаге (A + B) + C, при условии что A, B и C хранятся в 16-разрядном формате NVIDIA, описанном в упражнении 3.11.2 (а также в тексте). Используйте по одному разряду защиты, округления и напоминания и произведите округление до ближайшего четного числа. Покажите все шаги и запишите ответ в 16-разрядном двоичном формате для чисел с плавающей точкой и в десятичном формате.

3.13.2 [20] <3.2, 3.5, 3.6> Вычислите на бумаге A + (B + C), при условии, что A, B и C хранятся в 16-разрядном формате NVIDIA, описанном в упражнении 3.11.2 (а также в тексте). Используйте по одному разряду защиты, округления и напоминания и произведите округление до ближайшего четного числа. Покажите все шаги и запишите ответ в 16-разрядном двоичном формате для чисел с плавающей точкой и в десятичном формате.

3.13.3 [10] <3.2, 3.5, 3.6> Основываясь на ответах, полученных при выполнении упражнений 3.13.1 и 3.13.2, ответьте на вопрос, справедливо ли равенство (A + B) + C = A + (B + C)?

В следующей таблице показан еще один набор десятичных чисел.

	A	B	C
а	$4,8828125 \times 10^{-4}$	$1,768 \times 10^{2}$	$2,50125 \times 10^{2}$
б	$4,721875 \times 10^{1}$	$2,809375 \times 10$	$3,575 \times 10^{1}$

3.13.4 [30] <3.3, 3.5, 3.6> Вычислите на бумаге (A × B) × C, при условии, что A, B и C хранятся в 16-разрядном формате NVIDIA, описанном в упражнении 3.11.2 (а также в тексте). Используйте по одному разряду защиты, округления и напоминания и произведите округление до ближайшего четного числа. Покажите все шаги и запишите ответ в 16-разрядном двоичном формате для чисел с плавающей точкой и в десятичном формате.

3.13.5 [30] <3.3, 3.5, 3.6> Вычислите на бумаге A × (B × C), при условии, что A, B и C хранятся в 16-разрядном формате NVIDIA, описанном в упражнении 3.11.2 (а также в тексте). Используйте по одному разряду защиты, округления и напоминания и произведите округление до ближайшего четного числа. Покажите все шаги и запишите ответ в 16-разрядном двоичном формате для чисел с плавающей точкой и в десятичном формате.

3.13.6 [10] <3.3, 3.5, 3.6> Основываясь на ответах, полученных при выполнении упражнений 3.13.4 и 3.13.5, ответьте на вопрос, справедливо ли равенство (A × B) × C = A × (B × C)?

Упражнение 3.14

Ассоциативный закон не единственный из несоблюдаемых в отношении чисел с плавающей точкой. Существуют и другие странности. В таблице показан набор десятичных чисел:

	A	B	C
а	$1,5234375 \times 10^{-1}$	$2,0703125 \times 10^{-1}$	$9,96875 \times 10^{1}$
б	$-2,7890625 \times 10^{1}$	$-8,088 \times 103$	$1,0216 \times 10^{4}$

3.14.1 [30] <3.2, 3.3, 3.5, 3.6> Вычислите на бумаге A × (B + C), при условии, что A, B и C хранятся в 16-разрядном формате NVIDIA, описанном в упражнении 3.11.2 (а также в тексте). Используйте по одному разряду защиты, округления и напоминания и произведите округление до ближайшего четного числа. Покажите все шаги и запишите ответ в 16-разрядном двоичном формате для чисел с плавающей точкой и в десятичном формате.

3.14.2 [30] <3.2, 3.3, 3.5, 3.6> Вычислите на бумаге (A × B) + (A × C), при условии, что A, B и C хранятся в 16-разрядном формате NVIDIA, описанном в упражнении 3.11.2 (а также в тексте). Используйте по одному разряду защиты, округления

и напоминания и произведите округление до ближайшего четного числа. Покажите все шаги и запишите ответ в 16-разрядном двоичном формате для чисел с плавающей точкой и в десятичном формате.

3.14.3 [10] <3.2, 3.3, 3.5, 3.6> Основываясь на ответах, полученных при выполнении упражнений 3.14.1 и 3.14.2, ответьте на вопрос, справедливо ли равенство $(A \times B) + (A \times C) = A \times (B + C)$?

В следующей таблице показан еще один набор десятичных чисел:

	A	B
а	1/3	3
б	-1/7	7

3.14.4 [10] <3.5> Используя формат для чисел с плавающей точкой IEEE 754, запишите комбинацию двоичных разрядов, представляющую число A. Можно ли точно представить число A?

3.14.5 [10] <3.2, 3.3, 3.5, 3.6> Какой результат будет получен при сложении числа A с самим собой B раз? Чему будет равно $A \times B$? Будут ли два полученных значения одинаковыми? Какими они должны быть?

3.14.6 [60] <3.2, 3.3, 3.4, 3.5, 3.6> Что получится, если извлечь квадратный корень из B, а затем умножить это значение само на себя? Что вы должны получить? Выполните это упражнение для чисел как с одинарной, так и с двойной точностью. (Напишите программу для выполнения этих вычислений.)

Упражнение 3.15

Использование в мантиссе двоичных чисел не является обязательным. В компьютерах IBM в некоторых форматах чисел с плавающей точкой используются, к примеру, числа по основанию 16. Возможны также и другие подходы, у каждого из которых свои сильные и слабые стороны. В таблице показаны дроби для представления их в разных форматах чисел с плавающей точкой:

а	1/2
б	1/9

3.15.1 [10] <3.5, 3.6> Запишите комбинацию двоичных разрядов в мантиссе, при условии, что в ней используется формат двоичных чисел с плавающей точкой (именно тот, с которым вы имели дело в данной главе). Используйте 24 двоичных разряда и не придерживайтесь нормализации. Будет ли это представление абсолютно точным?

3.15.2 [10] <3.5, 3.6> Запишите комбинацию двоичных разрядов в мантиссе, при условии, что в ней используется формат чисел с плавающей точкой в двоично-десятичном коде — Binary Coded Decimal (с основанием 10), вместо чисел по основанию 2. Используйте 24 двоичных разряда и не придерживайтесь нормализации. Будет ли это представление абсолютно точным?

3.15.3 [10] <3.5, 3.6> Запишите комбинацию двоичных разрядов в мантиссе, при условии, что в ней вместо чисел по основанию 2 используются числа по основанию 15. (В числах по основанию 16 используются символы 0–9 и A–F, а в числах по основанию 15 используются символы 0–9 и A–E.) Используйте 24 двоичных разряда и не придерживайтесь нормализации. Будет ли это представление абсолютно точным?

3.15.4 [20] <3.5, 3.6> Запишите комбинацию двоичных разрядов в мантиссе, при условии, что в ней вместо чисел по основанию 2 используются числа по основанию 30. (В числах по основанию 16 используются символы 0–9 и A–F, а в числах по основанию 30 используются символы 0–9 и A–T.) Используйте 20 двоичных разрядов и не придерживайтесь нормализации. Будет ли это представление абсолютно точным? Видите ли вы какие-либо преимущества от использования такого подхода?

Ответы на вопросы для самопроверки

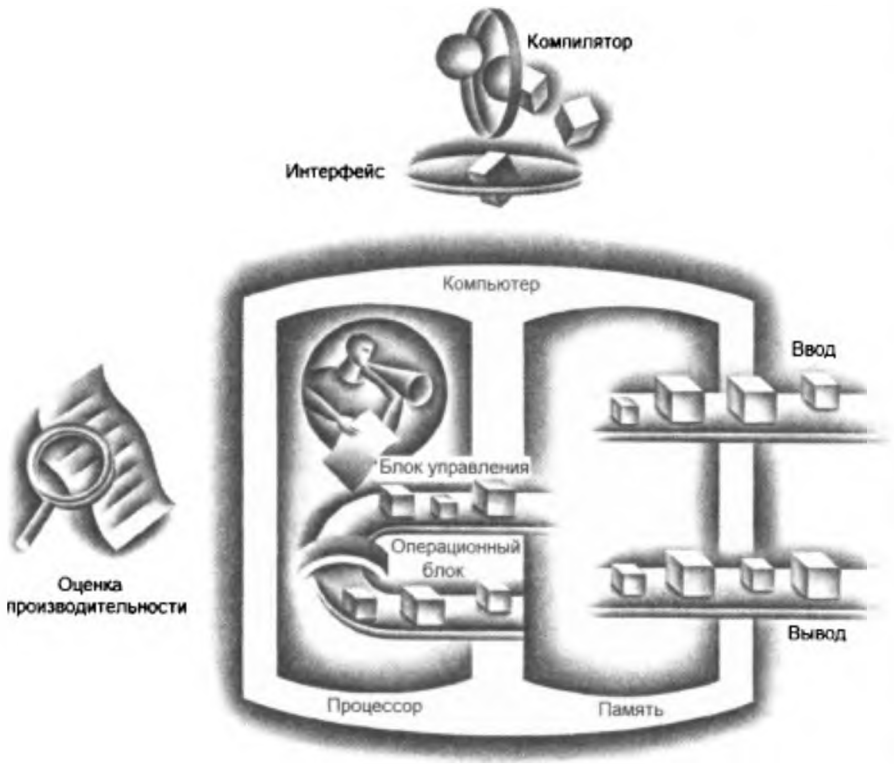
Раздел 3.2: 3.
Раздел 3.5: 3.

Глава 4

Процессор

По большому счету, несущественных деталей не существует.

Французская поговорка



4.1. Введение

В главе 1 говорилось о том, что производительность компьютера определяется тремя ключевыми факторами: количеством инструкций, продолжительностью тактового цикла и количеством тактовых циклов на инструкцию (CPI). В главе 2 говорилось о том, что количество инструкций, требуемое для той или иной программы, определяется компилятором и архитектурой набора инструкций

А вот продолжительность тактового цикла и количество тактовых циклов на инструкцию определяются реализацией процессора. В данной главе будет рассмотрена разработка операционного блока и блока управления для двух разных реализаций набора инструкций MIPS.

В главе будут рассмотрены принципы и технологии, используемые при создании процессора, начиная с сильно абстрагированного и упрощенного обзора, представляемого в данном разделе. Затем последует раздел, дополняющий создаваемую картину рассмотрением операционного блока и конструированием простой версии процессора, достаточного для реализации такого набора инструкций, как MIPS. В основной части главы будет рассмотрена более реалистичная реализация MIPS, использующая конвейеризацию, после чего последует раздел, в котором будут раскрыты понятия, необходимые для реализации более сложного набора инструкций, подобного x86.

Для читателей, желающих разобраться с высокоуровневой интерпретацией инструкций и ее влиянием на производительность программ, в этом начальном разделе, а также в разделе 4.5 представляются основные понятия конвейеризации. Самые последние тенденции рассматриваются в разделе 4.10, а в разделе 4.11 описывается недавно созданный микропроцессор AMD Opteron X4 (Barcelona). В данном разделе дается достаточный объем начальных сведений, позволяющий усвоить на высоком уровне понятие конвейеризации.

Для читателей, желающих глубже разобраться в устройстве процессора и его производительности, полезно будет изучить разделы 4.3, 4.4 и 4.6. Читатели, интересующиеся разработкой процессора, должны также изучить разделы 4.2, 4.7, 4.8 и 4.9.

Реализация элементарных инструкций MIPS

Рассмотрим реализацию, включающую поднабор основного набора инструкций MIPS:

- инструкции обращения к памяти, осуществляющие загрузку слова, — load word (`lw`) и сохранение слова — store word (`sw`);
- арифметико-логические инструкции `add`, `sub`, `AND`, `OR` и `slt`;
- инструкции условного перехода при равенстве — branch equal (`beq`) и безусловного перехода — jump (`j`), которые будут добавлены в последнюю очередь.

Этот поднабор не включает все инструкции для работы с целыми числами (к примеру, в нем отсутствуют инструкции сдвига, умножения и деления), а также не включает ни одной инструкции для работы с числами с плавающей точкой. Тем не менее здесь дается иллюстрация ключевых принципов, используемых при создании операционного блока и разработке блока управления. Реализация всех остальных инструкций осуществляется по сходным принципам.

При изучении реализации представится возможность разобраться в том, как в архитектуре набора инструкций определяются многие аспекты реализации и как выбор из множества стратегий реализации оказывает влияние на тактовую частоту и показатель CPI компьютера. При рассмотрении вопросов реализации могут быть

проиллюстрированы многие ключевые принципы конструирования, представленные в главе 2, например такие принципы, как «*часто встречающиеся задачи должны выполняться быстрее*» и «*простота предпочитает постоянство*». Кроме этого, большинство понятий, используемых при реализации поднабора инструкций MIPS в этой главе, представляют собой те же базовые идеи, которые использовались для конструирования широкого спектра компьютеров, от высокопроизводительных серверов до универсальных микропроцессоров для встраиваемых систем.

Общее представление о реализации

В главе 2 рассматривались основные инструкции MIPS, включая арифметико-логические инструкции для работы с целыми числами, инструкции обращения к памяти и инструкции условных переходов. Основная часть того, что должно быть сделано для реализации этих инструкций, имеет общие черты независимо от конкретного класса инструкции. Первые два шага для каждой инструкции совпадают:

1. Поместить в счетчик команд (PC) значение адреса ячейки памяти, содержащей код, и извлечь инструкцию из этой ячейки.
2. Прочитать значение одного или двух регистров, используя поля инструкции для выбора считываемых регистров. Для инструкции загрузки слова (load word) нужно прочитать только один регистр, но для большинства других инструкций требуется чтение двух регистров.

После этих двух шагов действия, требуемые для завершения инструкции, зависят от класса инструкции. В силу удачного стечения обстоятельств для каждого из трех классов инструкций (обращения к памяти, арифметико-логических и условного перехода) действия, независимо от конкретной инструкции, во многом схожи. Простота и постоянство, присущие набору инструкций MIPS, упрощают реализацию за счет схожести выполнения инструкций многих классов.

Например, все классы инструкций, за исключением инструкции перехода (jump), используют после чтения регистров арифметико-логическое устройство (АЛУ). Инструкции обращения к памяти используют АЛУ для вычисления адреса, арифметико-логические инструкции — для выполнения операций, а инструкции условных переходов — для сравнения. После использования АЛУ действия, необходимые для завершения инструкций разных классов, отличаются друг от друга. Инструкции обращения к памяти будут нуждаться в доступе к памяти либо для чтения данных с целью загрузки, либо для записи данных с целью их сохранения. Арифметико-логическая инструкция или инструкция загрузки должна записать данные из АЛУ или из памяти обратно в регистр. И наконец, для инструкции условного перехода может понадобиться изменить адрес следующей инструкции на основе проведенного сравнения; в противном случае для получения адреса следующей инструкции счетчик команд будет увеличен на 4.

На рис. 4.1 показано высокоуровневое представление реализации MIPS, основное внимание в котором уделено различным функциональным устройствам и их взаимосвязанности. Хотя на этом рисунке показано большинство потоков данных, проходящих через процессор, в нем опущены два важных аспекта выполнения инструкции.

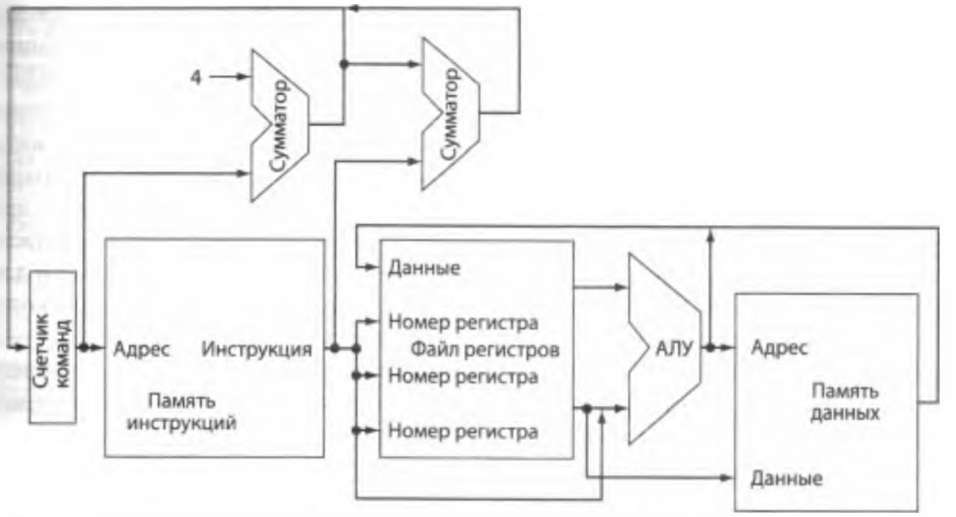


Рис. 4.1. Абстрактное представление реализации поднабора инструкций MIPS, показывающее главные функциональные блоки и основные связи между ними. Выполнение всех инструкций начинается с использования счетчика команд (PC) для получения адреса инструкции в памяти, хранящей инструкции. После извлечения инструкции поля этой инструкции определяют используемые ею операнды, находящиеся в регистрах. После извлечения операндов из регистров они могут быть обработаны для вычисления адреса памяти (для загрузки или сохранения), для вычисления арифметического результата (для целочисленных арифметико-логических инструкций) или для сравнения (для инструкций условного перехода). Если инструкция относится к классу арифметико-логических, результат из АЛУ должен быть записан в регистр. Если инструкция выполняет операцию загрузки или сохранения, результат из АЛУ используется в качестве адреса для сохранения значения из регистров либо для загрузки значения из памяти в регистры. Результат из АЛУ или из памяти записывается обратно в файл регистров[1]. Условные переходы требуют использования выхода АЛУ для определения адреса следующей инструкции, который поступает либо из АЛУ (когда PC и смещение условного перехода складываются), либо из сумматора, который увеличивает текущее значение PC на 4. Линии, связывающие функциональные блоки, представляют собой шины, по которым передаются сразу несколько сигналов. Стрелки используются для демонстрации направления информационных потоков. Поскольку сигнальные магистрали могут пересекаться, наличие соединений в местах их пересечения отмечено точками

Во-первых, в нескольких местах на рис. 4.1 показано, что данные, направляемые на конкретный блок, поступают из двух разных источников. Например, значение, записываемое в счетчик команд (PC), может поступать из одного из двух сумматоров, данные, записываемые в файл регистров, могут поступать либо из АЛУ, либо из памяти, хранящей данные, а второй вход АЛУ может получать данные из регистра или из поля непосредственных данных инструкции. На практике эти магистрали данных не могут быть так вот просто электрически связаны друг с другом; здесь необходимо добавить логический элемент, который бы осуществлял выбор из нескольких источников и направлял сигнал из одного из этих источников к месту его назначения. Обычно такой выбор производится устройством, которое называется

[1] «Файл регистров» в отечественной литературе также обозначается как «массив регистров» или «регистровый файл». — *Примеч. ред.*

мультиплексором, хотя лучше было бы назвать это устройство *селектором данных*. Сигнал на шине управления устанавливается прежде всего на основе информации, извлеченной из выполняемой инструкции.

Во-вторых, на рис. 4.1 отсутствует система, которая должна управлять некоторыми блоками в зависимости от типа выполняемой инструкции. Например, память, хранящая данные, должна подвергаться чтению при загрузке и записи при сохранении данных. Файл регистров должен подвергаться записи при загрузке и при выполнении арифметико-логической инструкции. И разумеется, АЛУ должно выполнять одну из нескольких операций, как было показано в главе 2. Как и для мультиплексоров, эти операции задаются по шинам управления, сигналы на которых выставляются на основе содержимого различных полей инструкции.

На рис. 4.2 показан операционный блок, изображенный на рис. 4.1, с добавленными к нему тремя необходимыми мультиплексорами, а также с шинами управления для главных функциональных блоков. Блок управления, на вход которого поступает инструкция, используется для определения сигналов, выставляемых на шинах управления для функциональных блоков и двух мультиплексоров. Третий мультиплексор, определяющий, какой адрес будет записан в счетчик команд, PC + 4 или адрес условного перехода, устанавливается на основе выхода АЛУ «Нуль» (Zero), который используется для осуществления сравнения в инструкции beq. «Постоянство и простота» инструкций MIPS означает, что для выставления сигналов на шинах управления может использоваться простой процесс декодирования.

В остальной части главы этот обзор будет уточнен для создания более подробной картины, позволяющей добавить следующие функциональные блоки, увеличить количество соединений между блоками и, разумеется, усовершенствовать блок управления, чтобы регулировать действия, предпринимаемые для различных классов инструкций. В разделах 4.3 и 4.4 рассматриваются простые реализации, использующие один длинный тактовый цикл для каждой инструкции, которые следуют общему представлению, показанному на рис. 4.1 и 4.2. В этой первой конструкции каждая инструкция начинает выполняться на одном фронте тактового импульса и заканчивает выполняться на другом его фронте.

При всей простоте восприятия этот подход не отличается практичностью, поскольку период тактового цикла должен быть удлинен, чтобы приспособиться под самую длинную инструкцию. После разработки системы управления для этого простого компьютера мы рассмотрим реализацию, использующую конвейер, со всеми ее сложностями, в том числе с исключениями.

Самопроверка

Сколько из пяти классических компонентов компьютера, показанных в начале главы, включают в себя схемы, представленные на рис. 4.1 и 4.2?

4.2. Соглашения по логическому проектированию

Чтобы рассматривать проектирование компьютера, нужно принять решение о том, как будет работать его логическая реализация и как будет синхронизироваться его работа. В этом разделе рассматриваются несколько ключевых идей цифровой логики, которые найдут широкое применение в данной главе.

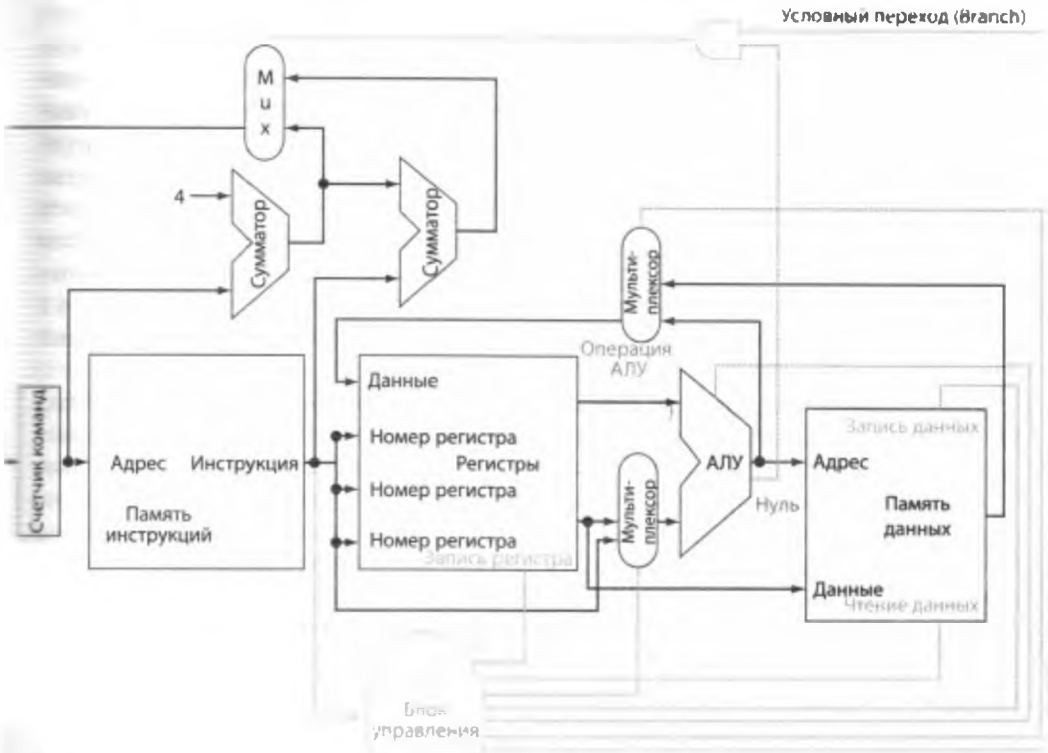


Рис. 4.2. Основная реализация поднабора инструкций MIPS, включающая все необходимые мультиплексоры и шины управления. Верхний мультиплексор («Mux») управляет значением, заменяющим значение PC (PC + 4 или адрес условного перехода); мультиплексор управляется логическим элементом, который осуществляет операцию «И» над выходом АЛУ «Нуль», и сигналом управления, свидетельствующим о том, что выполняется инструкция условного перехода. Средний мультиплексор, чей выход возвращается к файлу регистров, используется для управления выходом АЛУ (в случае выполнения арифметико-логической инструкции) или выходом памяти данных (в случае загрузки) с целью записи в файл регистров. И наконец, самый нижний мультиплексор используется для определения того, откуда АЛУ получит второй ввод: из регистров (для арифметико-логической инструкции или инструкции условного перехода) или из поля смещения (offset) инструкции (для инструкции загрузки или сохранения). Добавленные шины управления имеют вполне понятное назначение и определяют операцию, выполняемую АЛУ, а также необходимость чтения (или записи) данных и выполнения операции записи в регистры. Чтобы шины управления были заметнее, они выделены серым цветом

Устройства операционного блока в реализации MIPS состоят из двух разных типов логических элементов: тех, которые работают со значениями данных, и тех, которые хранят состояние. Все элементы, работающие со значениями данных, являются **комбинационными**, что означает, что их выходные сигналы зависят только от текущих входных сигналов. При одних и тех же входных сигналах комбинационный элемент всегда выставляет одни и те же выходные сигналы. АЛУ, показанное на рис. 4.1, является примером комбинационного элемента. При заданном наборе входных сигналов оно всегда выдает один и тот же выходной сигнал, потому что не имеет внутреннего хранилища данных.

Другие элементы в конструкции не относятся к комбинационным, но вместо этого хранят *состояние*. Элемент хранит состояние, если в нем имеется какое-нибудь внутреннее хранилище данных. Такие устройства называются **запоминающими элементами**, потому что если мы обесточим компьютер, мы сможем перезапустить его путем загрузки в запоминающие элементы тех значений, которые в них содержались до выдергивания вилки из розетки. Более того, если сохранить и восстановить значения запоминающих элементов, все будет находиться в таком состоянии, как будто компьютер и не обесточивался. Таким образом, эти запоминающие элементы дают полное описание состояния компьютера. На рис. 4.1 в качестве примеров запоминающих элементов можно привести память инструкций и данных, а также регистры.

Запоминающий элемент имеет как минимум два входа и один выход. На вход требуется подать значения данных, записываемых в элемент, и тактовый импульс, определяющий момент записи значения данных. На выходе запоминающего элемента выставляется значение, записанное в одном из предыдущих тактовых циклов. Например, одним из логически самых простых записывающих элементов является D-триггер, у которого как раз имеются эти два входа (значения и тактового импульса) и один выход. Кроме триггеров, в нашей реализации MIPS также используются два других типа запоминающих элементов: ячейки памяти и регистры, оба из которых присутствуют на рис. 4.1. Тактовый импульс определяет, когда в запоминающий элемент должна быть произведена запись; значение этого элемента может быть считано в любое время.

Логические компоненты, содержащие состояние, также называются *последовательными* (sequential), поскольку их выходы зависят от обоих входов и содержимого внутреннего состояния. Например, выход из функционального блока, представляющего регистры, зависит и от предоставленного номера регистра, и от того, что было предварительно записано в эти регистры.

Комбинационный элемент
Операционный элемент, такой как логический элемент И или АЛУ.

Запоминающий элемент
Элемент памяти, например регистр или оперативная память.

Выставленный
Сигнал, имеющий логически высокий уровень, или сигнал «истина» (true).

Невыставленный
Сигнал, имеющий логически низкий уровень, или сигнал «ложь» (false).

Для обозначения сигнала, имеющего высокий логический уровень, будет использоваться термин **выставленный** (asserted), для указания того, что сигнал должен управляться логически высоким уровнем, будет использоваться термин *выставлять*, а для представления логически низкого уровня будут использоваться термины **невыставленный** и *не выставлять*.

Метод синхронизации

Метод синхронизации определяет, когда сигналы должны считываться, а когда записываться. Определение моментов чтения и записи играет важную роль, поскольку если сигнал записывается в то же время, когда и считывается, считываемое значение может соответствовать старому значению, только что записанному значению или даже комбинации этих двух значений! Компьютерные конструкции не могут мириться с такой непредсказуемостью. Метод синхронизации как раз и разработан для обеспечения предсказуемости.

Чтобы упростить ситуацию, давайте допустим, что мы пользуемся методом синхронизации **по фронтам импульсов** (edge-triggered clocking). Этот метод означает, что любые значения, сохраненные в элементе последовательной логики, обновляются только при прохождении фронта синхроимпульса. Поскольку хранить значения данных могут только запоминающие элементы, на входы любого набора комбинационной логики должны поступать сигналы от набора запоминающих элементов, а выходные сигналы должны записываться в набор запоминающих элементов.

Вводом являются значения, которые были записаны во время предыдущего тактового цикла, а выводом — значения, которые могут использоваться во время следующего тактового цикла.

На рис. 4.3 показаны два запоминающих элемента, работающие в одном тактовом цикле: все сигналы должны передаваться от запоминающего элемента 1 через блок комбинационной логики к запоминающему элементу 2 за время одного тактового цикла. Продолжительность тактового цикла определяется временем, необходимым для того, чтобы сигнал достиг запоминающего элемента 2.

Чтобы не усложнять картину, здесь не показан **сигнал управления**, инициирующий запись, тогда как запись в запоминающий элемент ведется при каждом активном фронте синхроимпульса. В противоположность этому, если запоминающий элемент не обновляется по каждому синхроимпульсу, требуется конкретный сигнал управления, инициирующий запись. Синхроимпульс и управляющий сигнал записи являются входными сигналами, и содержимое запоминающего элемента изменяется только тогда, когда выставлен управляющий сигнал записи и прошел фронт синхроимпульса.

Метод синхронизации по фронту импульса позволяет читать содержимое регистра, перенаправлять значение через комбинационную логику и записывать значение в этот же регистр за один тактовый цикл. Обобщенный пример показан на рис. 4.4. Неважно, на каком именно фронте синхроимпульса, нарастающем или ниспадающем, мы предположим проведение записи, поскольку входы комбинационного логического блока не могут изменяться кроме как по выбранному фронту синхроимпульса. При методе синхронизации по фронту импульса

Метод синхронизации
Подход, используемый для определения момента достоверности и стабильности данных относительно синхроимпульсов.

Синхронизация по фронтам импульсов
Схема синхронизации, в которой все состояния изменяются по фронту синхроимпульса.

Сигнал управления
Сигнал, используемый для выбора, осуществляемого мультиплексором, или для инициирования операции функционального блока; является противоположностью «сигналу данных», содержащему информацию, с которой работает функциональный блок.

внутри одиночного тактового импульса ответной реакции *не возникает* и логика, показанная на рис. 4.4 работает правильно.

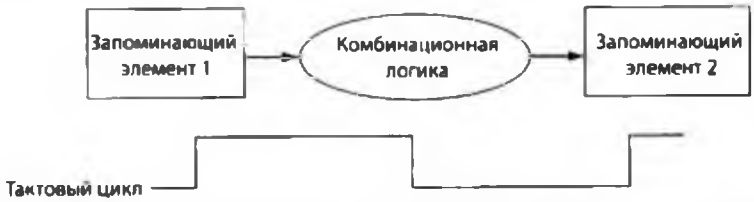


Рис. 4.3. Комбинационная логика, запоминающие элементы и синхронизация тесно связаны друг с другом. В синхронной цифровой системе синхроимпульс определяет, когда элементы, сохраняющие состояние, запишут значения во внутреннее хранилище. Любые входные данные запоминающего элемента должны получить стабильное значение (то есть должно быть получено значение, не изменяющееся до следующего фронта синхроимпульса), пока активный фронт синхроимпульса не вызовет обновление состояния. Все запоминающие элементы, рассматриваемые в данной главе, включая память, считаются запускаемыми по фронту синхроимпульса

Рис. 4.4. Метод синхронизации по фронту импульса позволяет вести чтение запоминающего элемента и запись в него за один и тот же тактовый цикл, не создавая при этом соревнования, которое может привести к неопределенности в значениях данных. Разумеется, тактовый цикл должен быть достаточно продолжительным, чтобы входные значения стабилизировались до поступления активного фронта синхроимпульса. Внутри самого синхроимпульса никакой ответной реакции не происходит, поскольку обновление содержимого запоминающего элемента происходит по его фронту. Если бы была возможна обратная реакция, эта конструкция не смогла бы работать правильно. Наши конструкции в этой и в следующей главах основаны на методе синхронизации по фронту импульса, и на структурах, похожих на показанную на данном рисунке

Для 32-разрядной архитектуры MIPS почти все эти запоминающие и логические элементы будут иметь входы и выходы шириной 32 разряда, поскольку такую величину будет иметь большинство данных, обрабатываемых процессором. Если вход или выход блока будет иметь ширину, отличающуюся от 32 разрядов, об этом будет сообщено дополнительно. Шины, имеющие сигнальные линии ширины одного разряда, будут показаны на рисунках толстыми линиями. Иногда нам потребуется объединить несколько шин для формирования более широкой шины, например может потребоваться получить 32-разрядную шину путем объединения двух 16-разрядных шин. В таких случаях надписи на шинах будут пояснять, что мы объединили шины для формирования более широкой шины. Будут также добавлены стрелки, чтобы помочь разобраться с направлением потока данных между элементами. И наконец, управляющий сигнал, в отличие от сигнала, переносящего данные, будет выделен другим оттенком, это различие станет понятнее по мере изучения данной главы.

Самопроверка

Определите правдивость или ложность следующего утверждения: поскольку файл регистров за один тактовый цикл подвергается чтению и записи, любые операционные блоки MIPS, использующие запись по фронту синхроимпульса, должны иметь более одной копии файла регистров.

Уточнение. Существует также 64-разрядная версия MIPS-архитектуры, и, что вполне естественно, большинство каналов в ее реализации будут иметь ширину 64 разряда. К тому же мы используем термины «выставленный» и «невыставленный», потому что бывает так, что единица представляет собой логически высокий уровень, а временами она может представлять и логически низкий уровень.

4.3. Создание операционного блока

Вполне разумно будет приступить к разработке операционного блока с изучения основных компонентов, необходимых для выполнения каждого класса MIPS-инструкций. Давайте начнем с рассмотрения вопроса, в каких **элементах операционного блока** нуждается каждая инструкция. При демонстрации элементов операционного блока будут показаны также и их управляющие сигналы.

На рис. 4.5, *а* показан первый необходимый нам элемент: блок памяти для хранения инструкций программы, предоставляющий инструкции, хранящиеся по заданному адресу. Также на рис. 4.5, *б* показан **счетчик команд** (**PC**), который, как мы видели в главе 2, является регистром, содержащим адрес текущей инструкции. И наконец, нам понадобится сумматор для увеличения значения счетчика команд с целью получения адреса следующей инструкции. Этот сумматор, являющийся комбинационным элементом, может быть создан на основе АЛУ просто путем такого монтажа управляющих линий, при котором всегда определяется только операция сложения. На рисунках такое АЛУ будет помечено названием «*Add*», как на рис. 4.5, чтобы показать, что оно постоянно служит в качестве сумматора и не может выполнять другие функции АЛУ.

Чтобы выполнить любую инструкцию, нужно сначала извлечь эту инструкцию из памяти. Для подготовки к выполнению следующей инструкции нужно также увеличить значение счетчика команд, чтобы он указывал на эту инструкцию, расположенную четырьмя байтами дальше. На рис. 4.6 показано, как объединить три элемента с рис. 4.5 для формирования операционного блока, извлекающего инструкции и увеличивающего значение PC для получения адреса следующей по порядку инструкции.

Теперь давайте рассмотрим инструкции R-формата (см. табл. 2.7). Все они считывают содержимое двух регистров, осуществляют над содержимым регистров АЛУ-операцию и записывают результат в регистр. Мы называем такие инструкции либо *инструкциями R-типа*, либо *арифметико-логическими инструкциями*

Элемент операционного блока
Блок, используемый для проведения операции над данными или для их сохранения внутри процессора. В реализации MIPS элементы операционного блока включают в себя запоминающие устройства инструкций и данных, файл регистров, АЛУ и сумматоры.

Счетчик команд (PC)
Регистр, содержащий адрес выполняемой инструкции программы.

(поскольку они выполняют арифметические или логические операции). К этому классу относятся инструкции add, sub, AND, OR и slt, представленные в главе 2. Вспомним, что типичным примером такой инструкции может послужить инструкция add $t1, $t2, $t3, которая читает содержимое регистров $t2 и $t3 и ведет запись в регистр $t1.

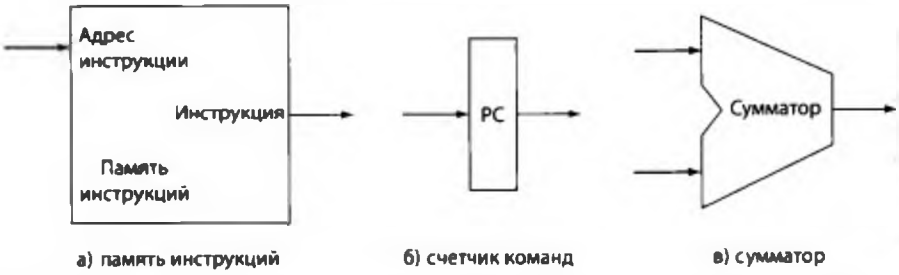


Рис. 4.5. Два запоминающих элемента, необходимых для хранения инструкций и доступа к ним, и сумматор, необходимый для вычисления адреса следующей инструкции. Этими запоминающими элементами являются память инструкций и счетчик команд. Память инструкций должна обеспечивать только лишь доступ по чтению, потому что операционный блок не записывает инструкции. Поскольку память инструкций подвергается только чтению, мы рассматриваем ее как комбинационную логику: выход в любое время отражает содержимое места, указанного на адресном входе, и не требует сигнала управления, инициирующего чтение. (Запись в память инструкций понадобится при загрузке программы, ее нетрудно добавить, но, чтобы не усложнять схему, она будет проигнорирована.) Счетчик команд является 32-разрядным регистром, в который ведется запись в конце каждого тактового цикла, и поэтому он не нуждается в сигнале управления, инициирующем запись. Сумматор представляет собой АЛУ, которое скоммутировано на постоянное сложение двух 32-разрядных входных данных и помещение на выходе их суммы

Принадлежащие процессору 32 регистра общего назначения хранятся в структуре под названием **файл регистров**. Этот файл представляет собой набор регистров, в котором каждый регистр может быть прочитан или записан путем указания его номера в файле. Файл регистров содержит регистровое состояние компьютера. Вдобавок ко всему прочему нам понадобится АЛУ для оперирования значениями, прочитанными из регистров.

Инструкции R-формата имеют три регистровых операнда, поэтому для каждой инструкции потребуется чтение двух слов данных из файла регистров и запись одного слова данных в этот файл. Для того чтобы любые данные были прочитаны из регистров, нужен вход в файл регистров, указывающий номер регистра, из которого нужно считать данные, и выход из файла регистров, который бы переносил значение, считываемое с регистра. Чтобы записать слово данных, нужны два входа: один, указывающий *номер регистра*, в который будет вестись запись, и еще один, предоставляющий записываемые в регистр *данные*. Файл регистров всегда выводит содержимое любых регистров, номера которых выставлены на входах «Считываемый регистр». А вот операции записи управляются специальным сигналом записи, который должен быть выставлен, чтобы запись состоялась при прохождении фронта синхроимпульса. Результат

Файл регистров
Запоминающий элемент, состоящий из набора регистров, значения которых могут быть прочитаны и записаны путем предоставления для доступа к данным номера регистра.

выполнения всех этих требований показан на рис. 4.7, *а*: нам нужны всего четыре входа (три для номеров регистров и один для данных) и два выхода (оба для данных). Входы для номеров регистров имеют ширину 5 разрядов для указания одного из 32 регистров, а вход и два выхода для данных имеют ширину 32 разряда.

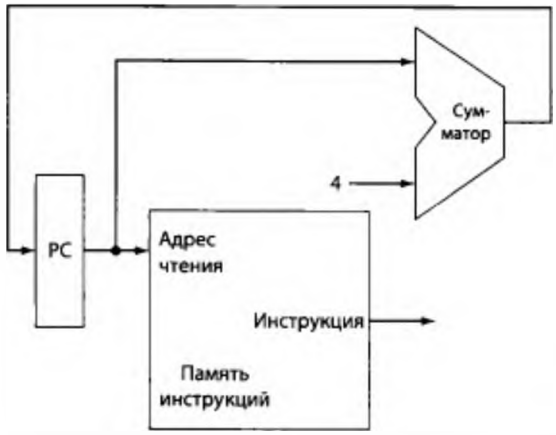


Рис. 4.6. Часть операционного блока, используемая для извлечения инструкций и увеличения значения счетчика команд. Извлекаемая инструкция используется другими частями операционного блока

На рис. 4.7, *б* показано АЛУ, имеющее два 32-разрядных входа и выдающее 32-разрядный результат, а также одноразрядный сигнал, если результат равен нулю. Также на АЛУ подается 4-разрядный управляющий сигнал; управление АЛУ будет коротко рассмотрено, когда возникнет необходимость разобраться в его установке.

Теперь рассмотрим MIPS-инструкции загрузки слова (load word) и сохранения слова (store word), имеющие общий вид `lw $t1,значение_смещения($t2)` или `sw $t1,значение_смещения($t2)`. Эти инструкции вычисляют адрес в памяти, прибавляя значение базового регистра, хранящееся в регистре `$t2`, к 16-разрядному полю смещения со знаком, содержащемуся в инструкции. Если это инструкция сохранения, то сохраняемое значение должно также быть считано из файла регистров, и находиться в регистре `$t1`. Если это инструкция загрузки, то значение, считываемое из памяти, должно быть записано в файл регистров в указанный регистр, в качестве которого используется регистр `$t1`. Таким образом, нам понадобятся файл регистров и АЛУ, изображенные на рис. 4.7.

Кроме этого, нам понадобится блок для **расширения знака**, позволяющий преобразовать 16-разрядное поле смещения в 32-разрядное значение со знаком, и блок памяти данных для чтения и записи данных. Память данных должна принимать запись при выполнении инструкций сохранения; следовательно, эта память должна иметь управляющие сигналы чтения и записи, вход для адреса и вход для записываемых в память данных. Эти два элемента показаны на рис. 4.8.

Расширение знака
Используется для увеличения размера элемента данных путем повторения самого старшего знакового разряда исходного элемента в самых старших знаковых разрядах более крупного элемента — приемника данных.

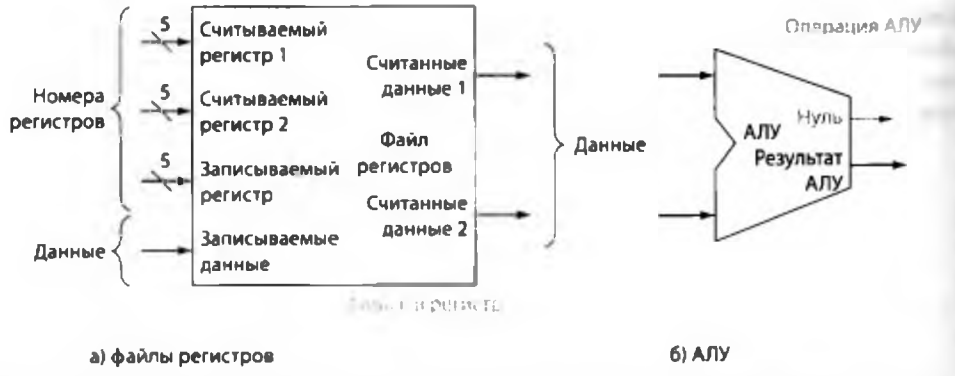


Рис. 4.7. Для реализации АЛУ-операций R-формата понадобятся два элемента: файл регистров и АЛУ. Файл регистров содержит все регистры и имеет два порта для чтения и один для записи. Файл регистров всегда выставляет на выходах содержимое регистров, соответствующее входам «Считываемый регистр»; выставление других управляющих входных сигналов не требуется. В отличие от этого запись регистра должна быть конкретно обозначена выставлением управляющего сигнала записи. Следует помнить, что запись запускается при прохождении фронта синхроимпульса, поэтому к моменту его прохождения должны быть выставлены все входные данные для записи (то есть записываемое значение, номер регистра и управляющий сигнал записи). Поскольку запись в файл регистров ведется по фронту синхроимпульса, наша конструкция может вполне легально считывать данные и записывать их в один и тот же регистр за один тактовый цикл: при чтении будет извлекаться значение, записанное при прежнем тактовом цикле, а записываемое значение станет доступным для чтения при последующем тактовом цикле. Все входы, переносящие номер регистра в файл регистров, имеют ширину 5 разрядов, а шины, переносящие данные, имеют ширину 32 разряда. Операция, выполняемая АЛУ, управляется сигналом «Операция АЛУ», который имеет ширину 4 разряда. Совсем скоро выход Zero (обнаружение нулевого значения) АЛУ будет использоваться нами для реализации условных переходов. Выход переполнения не понадобится вплоть до раздела 4.9, в котором будут рассматриваться исключения, поэтому пока мы его опускаем

У инструкции `beq` три операнда: два регистра, проверяемых на равенство, и 16-разрядное смещение, используемое для вычисления **целевого адреса условного перехода** относительно адреса инструкции условного перехода. Эта инструкция имеет формат `beq $t1,$t2,offset`. Для реализации данной инструкции нужно вычислить целевой адрес условного перехода — сложить поле смещения инструкции, прошедшее процедуру расширения, со значением счетчика команд (PC). В определении инструкций условного перехода есть две особенности (см. главу 2), на которые следует обратить внимание:

- Архитектура набора инструкций определяет, что базой для вычисления адреса условного перехода является адрес инструкции, следующей за инструкцией условного перехода. Поскольку в операционном блоке при извлечении инструкции вычисляется значение PC + 4 (адрес следующей инструкции), это значение можно использовать в качестве базы для вычисления целевого адреса условного перехода.

Целевой адрес условного перехода
Адрес, указанный в условном переходе, который при осуществлении перехода становится новым значением счетчика команд (PC). В архитектуре MIPS целевой адрес условного перехода получается из суммы значения поля смещения (offset) инструкции и адреса инструкции, следующей за инструкцией условного перехода.

- В архитектуре также существует положение, согласно которому поле смещения, `offset`,

сдвигается влево на 2 разряда, чтобы из него получилось смещение, кратное словам; этот сдвиг увеличивает эффективный диапазон поля смещения в 4 раза.

а) блок памяти данных б) блок расширения знака

Рис. 4.8. Два блока, необходимые для реализации загрузок и сохранений в дополнение к файлу регистров и АЛУ с рис. 4.7, — память данных и блок расширения знака. Блок памяти является запоминающим устройством с входами для адреса и для записываемых данных и одним выходом для результата чтения данных. У этого блока имеются отдельные выводы управления, хотя в конкретном тактовом цикле подтверждение может быть выставлено только на одном из них. Блок памяти нуждается в сигнале чтения, поскольку, в отличие от файла регистров, чтение значения с использованием неправильного адреса, как будет показано в главе 5, может вызвать проблемы. Блок расширения знака имеет 16-разрядный вход, данные с которого превращаются при расширении знака в 32-разрядный результат, выставляемый на выходе (см. главу 2). Мы предполагаем, что запись данных в память осуществляется по фронту синхроимпульса. На самом деле у стандартных микросхем памяти есть сигнал разрешения записи, используемый при этой операции. Хотя разрешение записи не управляется по фронту синхроимпульса, наша конструкция, управляемая по этому фронту, может быть легко приспособлена для работы с настоящими микросхемами памяти

Чтобы справиться с последней трудностью, нам нужно будет сдвинуть поле смещения на 2 разряда. Кроме вычисления целевого адреса условного перехода, мы также должны определить, какой будет следующая инструкция: той, которая стоит сразу же за текущей, или той, которая находится по целевому адресу условного перехода. При соблюдении условия (то есть когда операнды равны друг другу) целевой адрес условного перехода становится новым значением счетчика команд (PC), и тогда говорится, что **условный переход состоялся**. Если операнды не равны друг другу, текущее значение PC будет заменено его увеличенным значением (чтобы указать на следующую обычную инструкцию); в данном случае говорится, что **условный переход не состоялся**.

Состоявшийся условный переход
Условный переход с выполненным условием и значением счетчика команд (PC), получившим значение целевого адреса условного перехода. Все безусловные переходы считаются состоявшимися.

Несостоявшийся условный переход
Условный переход с невыполненным условием и значением счетчика команд (PC), получившим значение адреса инструкции, непосредственно следующей за инструкцией условного перехода.

Таким образом, операционный блок, обеспечивающий условный переход, должен выполнять две операции: вычислять целевой адрес условного перехода и сравнивать содержимое регистров. (Совсем скоро мы столкнемся с тем, что условные переходы также влияют и на ту часть операционного блока, которая занимается

извлечением инструкций.) На рис. 4.9 показана структура той части операционного блока, которая занимается обеспечением условных переходов. Для вычисления целевого адреса условного перехода операционный блок, обеспечивающий этот переход, включает в себя блок расширения знака с рис. 4.8 и сумматор. Для выполнения сравнения нужно использовать файл регистров с рис. 4.7, *а*, который предоставит два регистровых операнда (хотя запись в файл регистров нам в данном случае не понадобится). Кроме этого, сравнение должно быть выполнено с использованием АЛУ. Поскольку АЛУ предоставляет выходной сигнал, свидетельствующий о нулевом результате, мы можем отправить два регистровых операнда в АЛУ, настроив его сигналом управления на вычитание. Если на выходе блока АЛУ будет выставлен сигнал Zero (Нуль), мы будем знать, что два значения равны друг другу. Хотя нуль на выходе всегда сигнализирует, что результат равен нулю, мы будем использовать его только для реализации тестирования на равенство при выполнении условных переходов. Позже будет подробно показано, как подвести сигналы управления к АЛУ, использующемуся в операционном блоке.

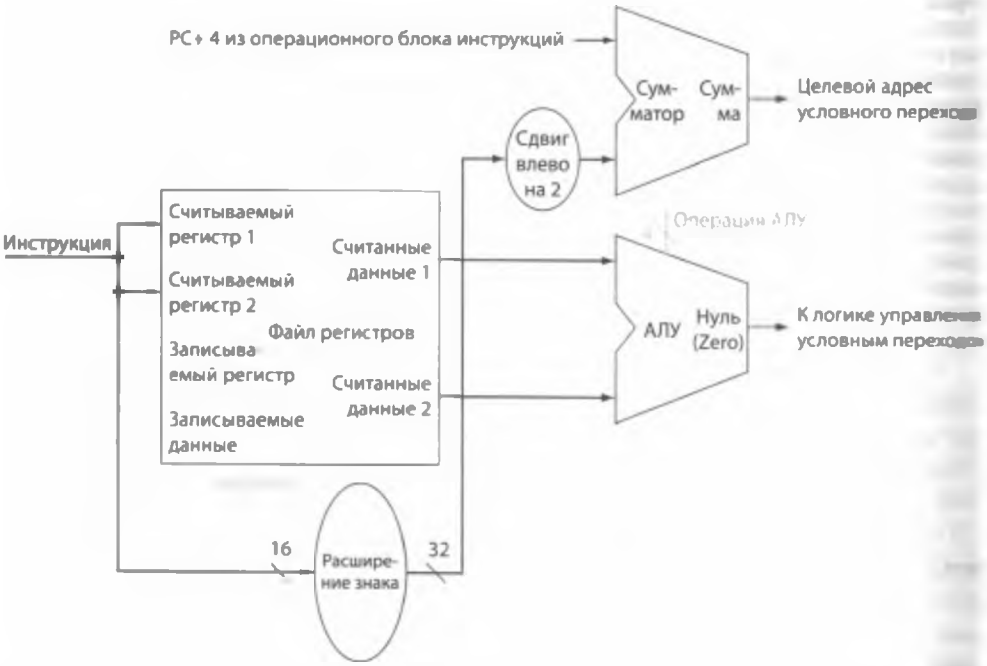


Рис. 4.9. Операционный блок, обеспечивающий условный переход, использует АЛУ для вычисления условия перехода и отдельный сумматор для вычисления целевого адреса перехода в виде суммы приращенного показателя PC и прошедших через расширение знака младших 16 разрядов инструкции (смещения условного перехода), сдвинутых на два разряда. Блок, обозначенный как «Сдвиг влево на 2», — это простой переходник сигналов, получающий их на входе и выдающий их на выходе со значением 00_2, добавленным к младшим разрядам поля смещения, прошедшего расширение знака; здесь не нужен настоящий сдвиг, поскольку количество разрядов «сдвига» является числом постоянным. Так как нам известно, что поле смещения прошло расширение знака, будучи ранее 16-разрядным, то при таком сдвиге будут отброшены только «знаковые разряды». Для решения о том, нужно ли просто приращать значение PC или заменить его значение целевым адресом условного перехода, используется управляющая логика на основе имеющегося в АЛУ выхода Нуль

Инструкция безусловного перехода выполняется путем замены младших 28 разрядов счетчика команд младшими 26 разрядами инструкции, сдвинутыми влево на два разряда. Этот сдвиг осуществляется простым присоединением 00 к смещению перехода, что уже было рассмотрено в главе 2.

Уточнение. В наборе инструкций MIPS условные переходы являются **отложенными**, а это значит, что инструкция, которая следует сразу же после инструкции условного перехода, выполняется всегда, *независимо* от соблюдения или несоблюдения условия перехода. Когда условие не соблюдается, выполнение выглядит вполне естественным продолжением программы. Когда условие соблюдается, отложенный условный переход до реального перехода на целевой адрес сначала выполняет ту инструкцию, которая следует непосредственно за его инструкцией. Причина таких отложенных переходов кроется во влиянии конвейеризации на условные переходы (см. раздел 4.8). Чтобы не усложнять ситуацию, в данной главе отложенные условные переходы чаще всего будут игнорироваться, и мы будем реализовывать инструкцию beq, выполняемую без каких-либо задержек.

Создание единого операционного блока

После изучения компонентов операционного блока, необходимых для отдельных классов инструкций, их можно объединить в единый операционный блок и для завершения его реализации добавить управляющую структуру. Простейший операционный блок будет стараться выполнить любую инструкцию за один тактовый цикл. Это означает, что ни один из ресурсов операционного блока не может быть использован при выполнении отдельной инструкции более одного раза, поэтому любой его элемент, необходимость в котором возникает более одного раза, должен быть продублирован. Поэтому нам понадобится память для инструкций отдельно от памяти для данных. Несмотря на необходимость дублирования некоторых функциональных блоков, многие элементы могут совместно использоваться разными потоками инструкций.

Для совместного использования элементов операционного блока двумя различными классами инструкций может потребоваться наличие нескольких подключений к входу того или иного элемента с использованием мультиплексора и управляющего сигнала для выбора из нескольких входов.

Упражнение

Создание операционного блока

Операционные блоки для выполнения арифметико-логических инструкций (или инструкций R-типа) и инструкций для работы с памятью очень похожи друг на друга. Можно отметить следующие ключевые различия:

- Арифметико-логические инструкции используют АЛУ, на вход которого поступают данные из двух регистров. Инструкции для работы с памятью также могут использовать АЛУ для вычисления адреса, хотя при этом второй вход получает 16-разрядное значение поля смещения инструкции, прошедшее расширение знака.
- Значение, сохраняемое в регистре-получателе, поступает из АЛУ (для инструкции R-типа) или из памяти (для загрузки).

Отложенный условный переход
Разновидность условного перехода, предполагающая неизменное выполнение той инструкции, которая следует непосредственно за инструкцией условного перехода, независимо от выполнения или невыполнения условия перехода.

Покажите, как можно создать исполнительную часть операционного блока для выполнения инструкции обращения к памяти и арифметико-логических инструкций, использующих единый файл регистров и единое АЛУ для работы с обоими типами инструкций, добавив любое количество необходимых для этого мультиплексоров.

Ответ

Для создания операционного блока с единым файлом регистров и единым АЛУ нужно обеспечить поддержку двух разных источников для второго входа АЛУ, а также двух разных источников для данных, хранящихся в файле регистров. Поэтому один мультиплексор помещается на входе АЛУ, а второй — на входе данных файла регистров. Исполнительная часть объединенного операционного блока показана на рис. 4.10.

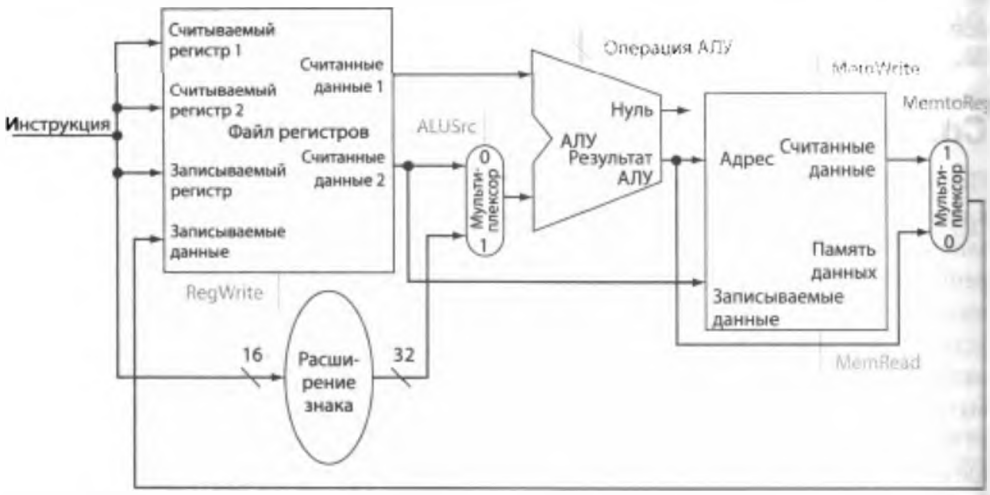


Рис. 4.10. Операционный блок для выполнения инструкций, предназначенных для работы с памятью, и инструкций R-типа. В этом примере показано, как единый операционный блок может быть собран из частей, показанных на рис. 4.7 и 4.8 путем, добавления мультиплексоров. Исходя из описания упражнения, здесь нужны два мультиплексора

Теперь можно объединить все составные части для создания единого операционного блока для MIPS-архитектуры, сложив вместе операционный блок для извлечения инструкций (рис. 4.6), операционный блок для выполнения инструкций обращения к памяти и инструкций R-типа (рис. 4.10) и операционный блок для выполнения условных переходов (рис. 4.9). На рис. 4.11 показан операционный блок, полученный за счет объединения отдельных составных частей. Инструкция условного перехода использует основное АЛУ для сравнения регистровых операндов, поэтому сумматор с рис. 4.9 можно сохранить для вычисления целевого адреса условного перехода. Здесь потребуется еще один мультиплексор для выбора между адресом инструкции, непосредственно следующей за текущей инструкцией (PC + 4), и целевым адресом условного перехода, который записывается в PC.

После того как комплектование этого единого операционного блока завершено, к нему можно добавить блок управления. Этот блок должен воспринимать входные

сигналы и генерировать сигнал записи для каждого запоминающего устройства, сигнал выбора для каждого мультиплексора и сигнал управления для АЛУ. Этот последний сигнал отличается сразу по нескольким признакам, и лучше будет сначала приступить к разработке системы его генерации, а уж потом заняться конструированием всего остального блока управления.

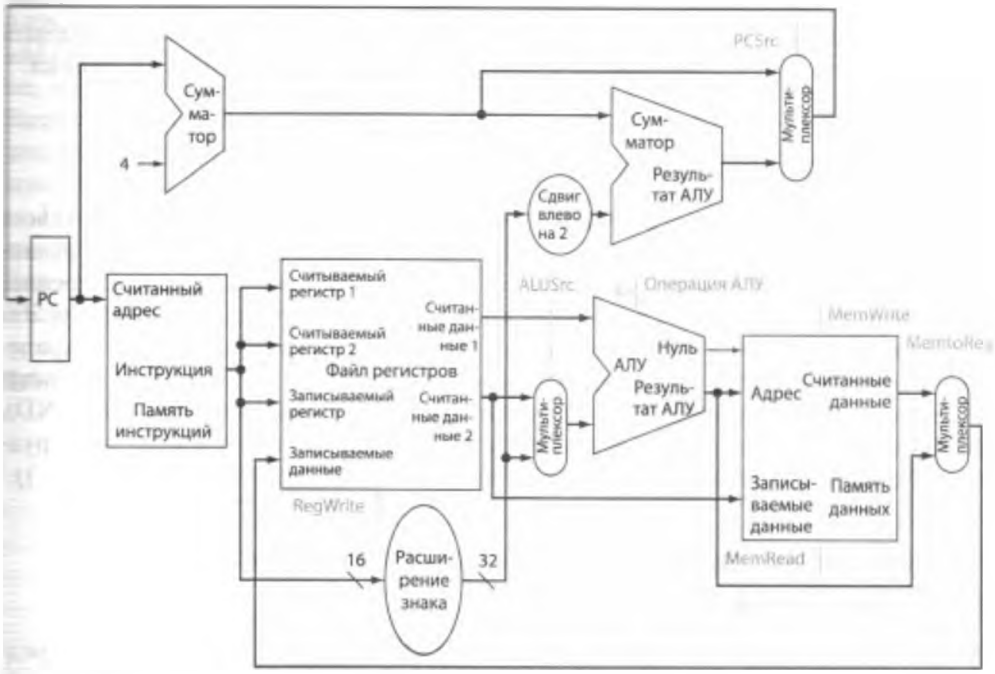


Рис. 4.11. Простой операционный блок для MIPS-архитектуры, объединяющий элементы, необходимые для разных классов инструкций. Он составлен из компонентов, показанных на рис. 4.6, 4.9 и 4.10. Этот операционный блок способен выполнять основные инструкции (загрузки-сохранения слова, АЛУ-операции и условные переходы) за один тактовый цикл. Для включения условных переходов требуется еще один мультиплексор. Поддержка безусловных переходов будет добавлена позже

Самопроверка

1. Что из нижеперечисленного справедливо для инструкции загрузки? Ответьте применительно к операционному блоку, изображенному на рис. 4.10.

   а) MemtoReg должен быть выставлен для извлечения данных из памяти и их последующей отправки в файл регистров.

   б) MemtoReg должен быть выставлен для правильного выбора регистра назначения и последующей отправки его значения в файл регистров.

   в) Для загрузки выставлять MemtoReg не нужно.

2. Операционный блок, выполняющий любую инструкцию за один цикл, концепция которого описана в данном разделе, должен иметь раздельную память для инструкций и данных, потому что:

   а) в MIPS форматы данных и инструкций отличаются друг от друга, и поэтому для них нужны разные устройства памяти;

   б) отдельные устройства памяти обходятся дешевле;

   в) процессор выполняет инструкцию за один цикл, и не может использовать память, имеющую один порт для двух разных обращений к ней за этот цикл.

4.4. Простая схема реализации

В этом разделе будет рассмотрено то, что можно считать наиболее простой из всех возможных реализаций нашего поднабора инструкций MIPS. Эта простая реализация будет построена путем использования операционного блока, рассмотренного в последнем разделе, и добавления к нему простой функции управления. Эта реализация включает в себя инструкции загрузки слова (load word (`lw`)), сохранения слова (store word (`sw`)), условного перехода по равенству (branch equal (`beq`)) и арифметико-логические инструкции сложения (add), вычитания (sub), И (AND), ИЛИ (OR) и установки, если меньше чем (set on less than). Чуть позже конструкция будет дополнена включением инструкции безусловного перехода (jump (`j`)).

Управление арифметико-логическим устройством

В АЛУ для MIPS-архитектуры определены шесть следующих комбинаций, предназначенных для четырех управляющих входов:

Линии управления АЛУ	Функция
0000	AND (И)
0001	OR (ИЛИ)
0010	add (сложение)
0110	subtract (вычитание)
0111	set on less than (установка, если меньше чем)
1100	NOR (НЕ-ИЛИ)

В зависимости от класса инструкции АЛУ придется выполнять одну из этих первых пяти функций. (NOR нужна для другой части набора инструкций MIPS, отсутствующей в реализуемом нами поднаборе.) При выполнении инструкций загрузки слова (load word) и сохранения слова (store word) АЛУ используется для вычисления адреса памяти путем сложения чисел. Для инструкций R-типа АЛУ должно выполнить одно из пяти действий (И — AND, ИЛИ — OR, вычитание — subtract, сложение — add или установка, если меньше чем — set on less than), в за-

висимости от значения 6-разрядного поля funct (или функции) в младших разрядах инструкции (см. главу 2). Для инструкции условного перехода по равенству АЛУ должно выполнить вычитание.

Четырехразрядный управляющий входной сигнал АЛУ можно сгенерировать с помощью небольшого блока управления, получающего на входе принадлежащее инструкции поле функции и двухразрядное поле управления, которое называется ALUOp. Это поле показывает, должна ли выполняемая операция быть сложением — add (00) для загрузки и сохранения, вычитанием — subtract (01) для инструкции beq, или определяться операцией, закодированной в поле функции, — funct (10). Выход блока управления АЛУ представляет собой 4-разрядный сигнал, который непосредственно управляет АЛУ, генерируя одну из ранее показанных 4-разрядных комбинаций.

В табл. 4.1 показано, как установить входные сигналы управления АЛУ на основе двухразрядного управляющего поля ALUOp и шестиразрядного кода функции. Далее в этой главе будет показано, как из основного блока управления генерируются разряды ALUOp.

Таблица 4.1. Порядок установки разрядов управления АЛУ в зависимости от содержимого управляющих разрядов ALUOp и различных кодов функции для инструкции R-типа. Коды операций — opcode, перечисленные в первом столбце, определяют установку разрядов ALUOp. Все кодирование показано в двоичной форме. Следует учесть, что при значении кода ALUOp, равном 00 или 01, заданное действие АЛУ не зависит от поля кода функции; в таком случае говорится, что нам «безразлично», какое значение у поля кода функции, и содержимое поля funct показано как XXXXXX. Если код ALUOp имеет значение 10, для установки входа управления АЛУ используется код функции

Opcode инструкции	ALUOp	Операция инструкции	Поле funct	Задаваемое действие АЛУ	Вход управления АЛУ
LW	00	load word (загрузка слова)	XXXXXX	сложение	0010
SW	00	store word (сохранение слова)	XXXXXX	сложение	0010
Branch equal	01	branch equal (условный переход по равенству)	XXXXXX	вычитание	0110
R-тип	10	add (сложение)	100000	сложение	0010
R-тип	10	subtract (вычитание)	100010	вычитание	0110
R-тип	10	AND (И)	100100	И	0000
R-тип	10	OR (ИЛИ)	100101	ИЛИ	0001
R-тип	10	set on less than (установка, если меньше чем)	101010	установка, если меньше чем	0111

Подобный стиль использования нескольких уровней декодирования, при котором основной блок управления генерирует биты ALUOp, использующие затем в качестве входных сигналов управления блока АЛУ, является весьма распространенной технологией реализации. Использование нескольких уровней управления может сократить размер основного блока управления. Использование нескольких более мелких блоков управления может также потенциально увеличить скорость работы блока управления. Подобные способы оптимизации играют важную роль, поскольку скорость работы блока управления является критическим фактором, определяющим продолжительность тактового цикла.

Существует несколько различных путей реализации отображения 2-разрядного поля ALUOp и 6-разрядного поля funct на четыре разряда управления операцией АЛУ. Поскольку интерес представляет только небольшое количество из 64 возможных значений поля функции и это поле используется, только когда значение разрядов ALUOp равно 10, мы можем использовать небольшой фрагмент логической схемы, распознающей поднабор возможных значений и обеспечивающей правильную установку разрядов управления АЛУ.

В качестве промежуточного шага по конструированию подобной логики полезно создать таблицу истинности для интересующих нас комбинаций поля кода функции и разрядов ALUOp, как это сделано в табл. 4.2; эта **таблица истинности** показывает, как в зависимости от значений этих двух входных полей устанавливается 4-разрядный сигнал управления АЛУ. Поскольку вся таблица истинности слишком велика (2^8 = 256 записей) и нас не интересуют значения управляющих сигналов АЛУ для многих из этих входных комбинаций, здесь показаны только те записи таблицы истинности, для которых управляющие сигналы АЛУ должны иметь вполне определенное значение. На протяжении всей этой главы мы будем демонстрировать только те записи таблицы истинности, которые формируют выходные сигналы, предназначенные для выставления, и игнорировать записи, которые не могут быть выставлены или не представляют для нас никакого интереса.

Поскольку во многих случаях нас не интересуют значения некоторых входных сигналов и поскольку нам желательно сохранить таблицы компактными, мы также включаем **безразличные элементы.** Эти элементы в данной таблице истинности (представленные символами X в столбце входного сигнала) показывают, что выходной сигнал не зависит от значения входного сигнала, соответствующего этому столбцу. Например, когда разряды ALUOp имеют значение 00, как в первой строке табл 4.2, мы всегда устанавливаем значение управляющего сигнала АЛУ равным 0010, независимо от кода функции. В таком случае код функции в данной строке таблицы истинности нам безразличен. Чуть позже будут показаны примеры других типов безразличных элементов.

Как только будет построена таблица истинности, она может быть оптимизирована, а затем

Таблица истинности
Предназначена для логических схем и является представлением логической операции путем перечисления всех значений на входах и демонстрации для каждого отдельного случая того, что должно получаться на выходах.

Безразличный элемент
Элемент логической функции, в котором выход не зависит от значений на всех входах. Безразличные элементы могут быть обозначены разными способами.

превращена в выходные сигналы. Этот процесс носит чисто механический характер, поэтому заключительные шаги здесь не показаны.

Таблица 4.2. Таблица истинности для четырех разрядов управления АЛУ (называемых Операцией). В качестве входных сигналов используются ALUOp и поле кода функции. Показаны только те записи, для которых выставляется сигнал управления АЛУ. К ним добавлены некоторые записи безразличных состояний. Например, в ALUOp не используется код 11, поэтому таблица истинности вместо 10 и 01 может содержать записи 1X и X1. Следует заметить, что при использовании в инструкциях поля функции первые два разряда (F5 и F4) этих инструкций всегда имеют значение 10, поэтому они являются безразличными элементами, и заменяются в таблице истинности символами XX

ALUOp		Поле Funct						Операция
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	
0	0	X	X	X	X	X	X	0010
0	1	X	X	X	X	X	X	0110
1	0	X	X	0	0	0	0	0010
1	X	X	X	0	0	1	0	0110
1	0	X	X	0	1	0	0	0000
1	0	X	X	0	1	0	1	0001
1	X	X	X	1	0	1	0	0111

Конструирование основного блока управления

После представления описания конструкции АЛУ, которая использует на входах управления код функции и двухразрядный сигнал, мы можем вернуться к рассмотрению всей остальной части блока управления. Для начала давайте обозначим поля инструкции и линии управления, необходимые для операционного блока, показанного на рис. 4.11. Чтобы понять, как связать поля инструкции с операционным блоком, полезно будет рассмотреть форматы трех классов инструкций: R-типа, условного перехода и загрузки-сохранения. Эти форматы показаны на рис. 4.12.

В отношении этого формата инструкции есть ряд важных наблюдений, на которые мы будем полагаться:

- Поле op, которое также называется **opcode**, всегда содержится в разрядах 31:26. Мы будем ссылаться на это поле как на `Op[5:0]`.
- Два регистра, из которых будет проводиться чтение, всегда указываются в полях `rs` и `rt`, в позициях 25:21 и 20:16. Это обстоятельство справедливо для инструкций R-типа, инструкции условного перехода по равенству и инструкции сохранения.
- Базовый регистр для инструкций загрузки и сохранения всегда указывается в позициях разрядов 25:21 (`rs`).

Opcode
Поле, которое указывает на операцию и на формат инструкции.

- 16-разрядное смещение для условного перехода по равенству, загрузки и сохранения всегда находится в позициях разрядов 15:0.
- Регистр-получатель может указываться в одном из двух мест. Для загрузки он указывается в позициях разрядов 20:16 (rt), а для инструкций R-типа — в позициях разрядов 15:11 (rd). Поэтому для выбора поля инструкции, используемого для указания номера записываемого регистра, нужно добавить мультиплексор.

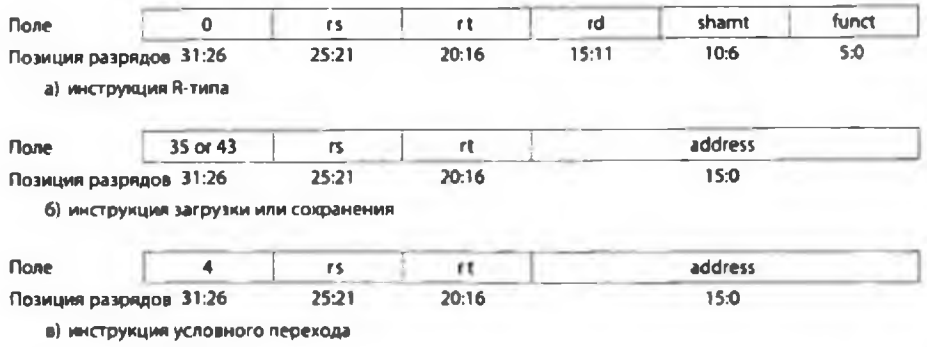


Рис. 4.12. Три класса инструкций (R-типа, загрузки и сохранения, условного перехода), использующие разные форматы инструкции. Инструкции безусловного перехода используют еще один формат, который вскоре будет рассмотрен. *а*) Формат инструкций R-типа, у которых значение opcode всегда равно 0. У этих инструкций имеется три регистровых операнда: rs, rt и rd. Поля rs и rt являются полями источника, а rd является полем назначения. Функция АЛУ представлена полем funct и декодируется с помощью блока управления АЛУ, конструкция которого была рассмотрена в предыдущем разделе. К реализуемым нами инструкциям R-типа относятся сложение — add, вычитание — sub, И — AND, ИЛИ — OR и установка, если меньше чем — slt. Поле shamt используется только для сдвигов; в данной главе оно будет проигнорировано. *б*) Формат инструкций загрузки (opcode = 35_{10}) и сохранения (opcode = 43_{10}). Регистр, указанный в поле rs, является базовым, и его значение складывается с 16-разрядным полем для формирования адреса памяти. Для инструкций загрузок регистр, указанный в поле rt, является регистром-получателем загружаемого значения. Для инструкций сохранений регистр, указанный в поле rt, является регистром-источником, чье значение должно быть сохранено в памяти. Формат инструкции для условного перехода по равенству (opcode = 4). Регистры, указанные в полях rs и rt, являются регистрами-источниками, проходящими тест на равенство. 16-разрядное поле адреса подвергается расширению знака, сдвигу и добавляется к значению счетчика команд (PC) для вычисления целевого адреса условного перехода

Здесь в определении средств управления приносит свои плоды первый принцип конструирования, рассмотренный в главе 2, — *простота и постоянство*.

Используя эту информацию, мы можем добавить к простому операционному блоку подписи для инструкций и еще один мультиплексор (для ввода номера записываемого регистра файла регистров). Эти дополнения плюс блок управления, сигналы записи для запоминающих элементов, сигнал чтения для памяти данных и сигналы управления для мультиплексоров показаны на рис. 4.13. Поскольку все мультиплексоры имеют по два входа, каждый из них нуждается в отдельной линии управления.

На рис. 4.13 показаны семь одноразрядных линий управления плюс двухразрядный сигнал управления ALUOp. Порядок работы сигнала управления ALUOp уже был определен, и теперь целесообразно будет неформально определить, что делают остальные семь сигналов управления еще до того, как мы определим, как установить эти сигналы управления в процессе выполнения инструкции. Предназначение этих семи линий управления показано в табл. 4.3.

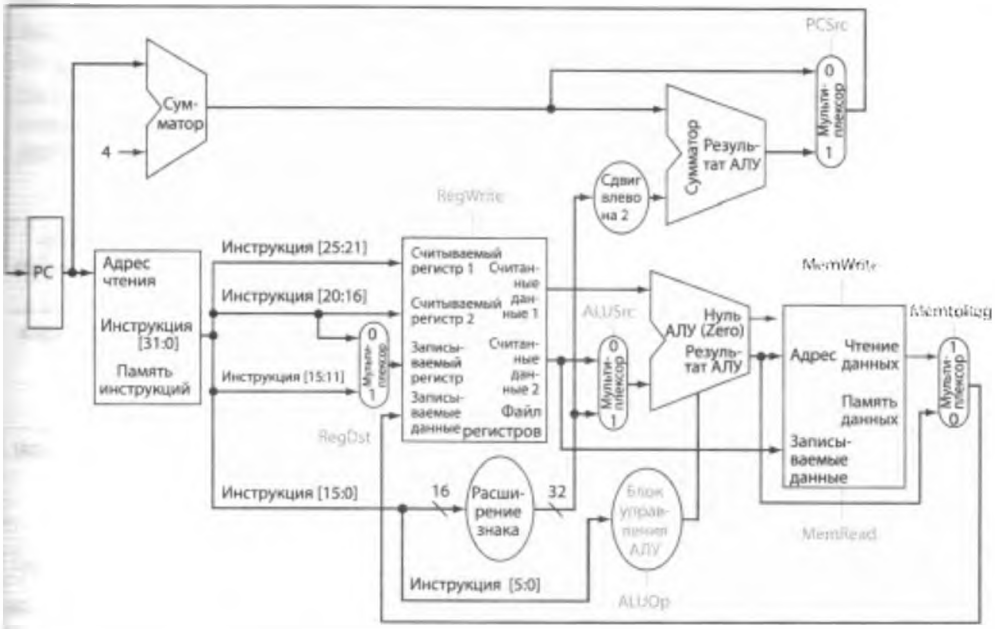


Рис. 4.13. Операционный блок с рис. 4.11 со всеми необходимыми мультиплексорами и всеми идентификаторами линий управления. Линии управления выделены серым цветом. Сюда также был добавлен блок управления АЛУ. Счетчику команд (PC) не нужен сигнал управления записью, поскольку значение в него записывается один раз в конце каждого тактового цикла; логика управления условными переходами определяет, когда в него записывается его же текущее значение с приращением или целевой адрес условного перехода

После того как мы посмотрели на функцию каждого из сигналов управления, можно посмотреть на то, как их нужно выставить. Блок управления может выставить все, кроме одного из сигналов управления, основываясь только лишь на содержимом поля opcode инструкции. Исключением является линия PCSrc. Сигнал на этой линии управления должен выставляться в том случае, если выполняется инструкция условного перехода по равенству — branch on equal (это решение может быть принято блоком управления) и на АЛУ выставлен сигнал Нуль (Zero), используемый для теста на равенство. Для генерации сигнала PCSrc нужно провести операцию И (AND) над сигналом с блока управления, который называется *Branch* (условный переход), и сигналом Zero на выходе АЛУ.

Таблица 4.3. Действие каждого из семи сигналов управления. Когда на мультиплексоре с двумя входами выставляется одноразрядный сигнал управления, мультиплексор выбирает вход номер 1. В противном случае, если управляющий сигнал не выставлен, мультиплексор выбирает вход номер 0. Следует помнить, что у всех запоминающих элементов имеется подразумеваемый вход для синхроимпульсов, используемый для управления записью. Внешний подвод синхроимпульсов к запоминающим элементам может вызвать проблемы синхронизации.

Название сигнала	Действие, когда сигнал не выставлен	Действие, когда сигнал выставлен
RegDst	Номер регистра-получателя для записываемого регистра, поступает из поля rt (разряды 20:16)	Номер регистра-получателя для записываемого регистра поступает из поля rd (разряды 15:11)
RegWrite	Нет	В регистр, указанный на входе «Записываемый регистр», записывается значение с входа «Записываемые данные»
ALUSrc	Второй операнд АЛУ поступает со второго выхода файла регистров («Считанные данные 2»)	Второй операнд АЛУ — младшие 16 разрядов инструкции со знаком
PCSrc	Значение PC заменяется значением с выхода сумматора, вычисляющего значение PC + 4	Значение PC заменяется значением с выхода сумматора, вычисляющего целевой адрес условного перехода
MemRead	Нет	На выход «Считанные данные» поступает содержимое блока «Память данных», соответствующее значению на входе «Адрес»
MemWrite	Нет	Содержимое блока «Память данных», соответствующее значению на входе «Адрес», заменяется значением, указанным на входе «Записываемые данные»
MemtoReg	Значение, поступающее в регистр на вход «Записываемые данные», берется из АЛУ	Значение, поступающее в регистр на вход «Записываемые данные», берется из памяти данных

Эти девять сигналов управления (семь из табл. 4.3 и два для ALUOp) теперь могут быть выставлены на основании шести входных сигналов в блоке управления, являющихся разрядами поля opcode с 31 по 26. На рис. 4.14 показан операционный блок с блоком управления и управляющими сигналами.

Перед тем как составить для блока управления систему уравнений или таблицу истинности, нужно попытаться неформально определить функцию управления. Поскольку выставление сигналов на линии управления зависит только от содержимого поля opcode, мы определяем для каждого из значений opcode, должен ли сигнал управления иметь значение 0, 1 или безразличное значение (X). На рис. 4.15 определен порядок сигналов управления для каждого значения поля opcode; эта информация непосредственно следует из данных, отображенных в табл. 4.1, 4.3 и на рис. 4.14.

Работа операционного блока

Исходя из информации, предоставленной в табл. 4.3 и на рис. 4.15, мы можем разработать логику блока управления, но перед этим давайте посмотрим на то, как инструкция использует операционный блок. На нескольких следующих изображениях будет показан поток данных, проходящих через операционный блок и связанных с выполнением трех разных классов инструкций. На каждом из них будут выделены выставленные сигналы управления и активные элементы операционного блока. Обратите внимание на то, что мультиплексор, на входе которого выставлен сигнал 0, работает предопределенным образом, даже если его линия управления не выделена. Многоразрядные сигналы управления выделены в том случае, если выставлен любой из входящих в состав сигналов.

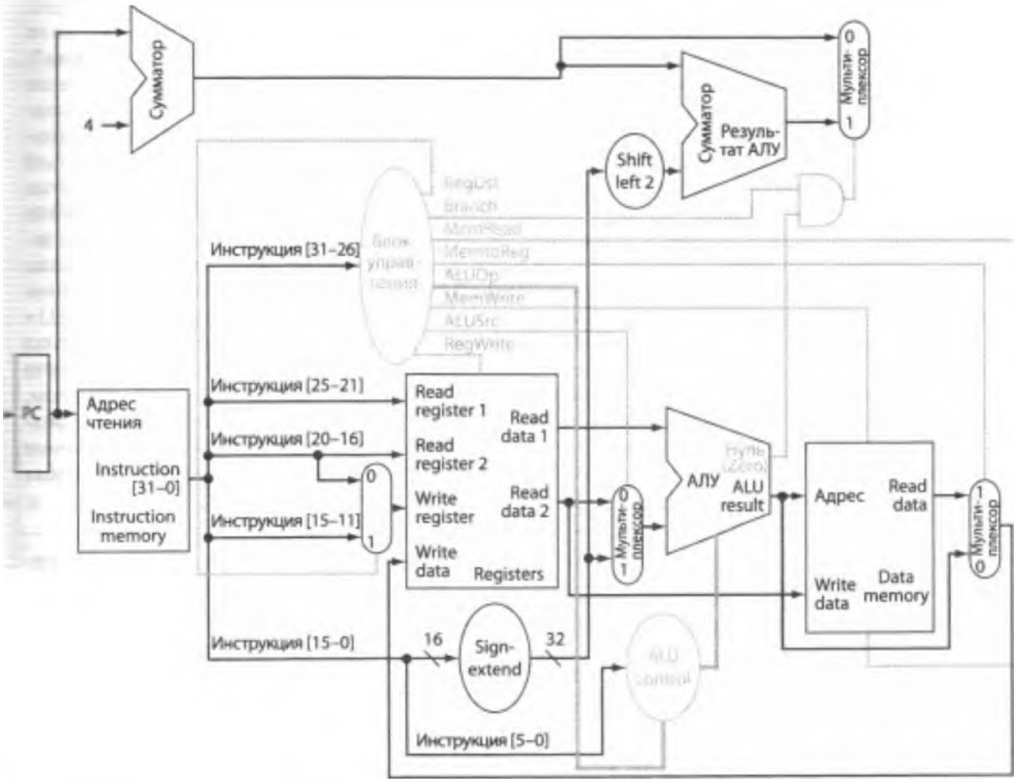


Рис. 4.14. Простейший операционный блок с блоком управления. На вход блока управления подается значение 6-разрядного поля opcode, взятое из инструкции. Выходы блока управления состоят из трех одноразрядных сигналов, используемых для управления мультиплексорами (RegDst, ALUSrc и MemtoReg), трех сигналов для управления процессами чтения и записи файла регистров и памяти данных (RegWrite, MemRead и MemWrite) и одноразрядного сигнала, используемого для определения возможного условного перехода (Branch), и двухразрядного сигнала для АЛУ (ALUOp). Для объединения сигнала управления условным переходом и АЛУ-выхода Нуль (Zero) используется элемент И (AND); выход этого элемента управляет выбором следующего значения PC. Обратите внимание на то, что теперь PCSrc является получаемым сигналом, а не одним из тех сигналов, который приходит непосредственно из блока управления. Поэтому на следующих изображениях название сигнала мы пропускаем

Инструкция	RegDst	ALUSrc	MemtoReg	RegWrite	MemRead	MemWrite	Branch	ALUOp1	ALUOp0
R-формат	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

Рис. 4.15. Выставление сигналов на управляющих линиях полностью определяется полями opcode инструкций. Первая строка таблицы относится к инструкциям R-формата (add, sub, AND, OR и slt). Для всех этих инструкций полями регистров-источников являются rs и rt, а полем регистра-получателя является rd; это определяет сигналы ALUSrc и RegDst. Кроме этого, инструкция R-типа ведет запись в регистр (RegWrite = 1), но не обращается к памяти данных ни по чтению, ни по записи. Когда управляющий сигнал условного перехода Branch имеет значение 0, значение PC безо всяких условий заменяется значением PC + 4; в противном случае значение PC заменяется целевым адресом условного перехода, если сигнал Нуль (Zero) АЛУ также имеет высокий уровень. Значение поля ALUOp для инструкций R-типа устанавливается в 10, чтобы показать, что сигнал управления АЛУ должен быть сгенерирован из поля funct. Во второй и третьей строках этой таблицы показаны сигналы управления для инструкций lw и sw. Значения сигналов ALUSrc и ALUOp обеспечивают вычисление адреса. Значения сигналов MemRead и MemWrite обеспечивают обращение к памяти. И наконец, сигналы RegDst и RegWrite выставляются на загрузку, заставляя результат сохраняться в регистре rt. Инструкция условного перехода похожа на операцию R-формата, поскольку она отправляет в АЛУ значение регистров, указанных в полях rs и rt. Сигнал ALUOp обеспечивает условный переход на вычитание (управляющий сигнал ALU = 01), которое используется для тестирования на равенство. Обратите внимание на то, что сигнал MemtoReg не учитывается, когда значение сигнала RegWrite равно 0: поскольку запись в регистр не ведется, значение данных в порту записи данных в регистр не используется. Таким образом, запись для MemtoReg в последних двух строках таблицы содержит знак X, что говорит о безразличности значения. Такой же знак может быть проставлен для сигнала RegDst при значении RegWrite, равном 0. Этот тип безразличности должен быть добавлен разработчиком, поскольку он зависит от знания тонкостей работы операционного блока

На рис. 4.16 показана работа операционного блока при выполнении такой инструкции R-типа, как `add $t1, $t2, $t3`. Хотя все происходит за один тактовый цикл, можно говорить о четырех шагах выполнения инструкции; эти шаги выстроены за счет потока информации:

1. Извлечение инструкции и приращение значения счетчика команд (PC).
2. Считывание из регистра файлов значений `$t2` и `$t3`; во время выполнения данного шага кроме этого происходит вычисление основным блоком управления значений, выставляемых на линиях управления.
3. Работа АЛУ с данными, считанными из файла регистров, с использованием кода функции (разряды 5:0, поле `funct` инструкции) для генерации функции АЛУ.
4. Запись результата, полученного из АЛУ, в файл регистров с использованием разрядов 15:11 инструкции для выбора регистра-получателя (`$t1`).

Аналогично этому можно проиллюстрировать выполнение инструкции загрузки слова (load word):

```
lw $t1, offset($t2)
```

На рис. 4.17 показаны задействованные функциональные блоки и линии управления с выставленными сигналами для загрузки. Инструкцию загрузки можно рассматривать как выполняемую за пять шагов (подобно тому как инструкция R-типа рассматривается как выполняемая за четыре шага):

1. Извлечение инструкции и приращение значения счетчика команд (PC).
2. Считывание из регистра файлов значения регистра ($t2).
3. Вычисление АЛУ суммы значения, считанного из файла регистров и младших 16 разрядов инструкции (смещения), прошедших расширение знака.
4. Использование суммы, полученной из АЛУ в качестве адреса для данных памяти.
5. Запись в файл регистров данных из блока памяти; регистр-получатель ($t1) задается разрядами 20:16 инструкции.

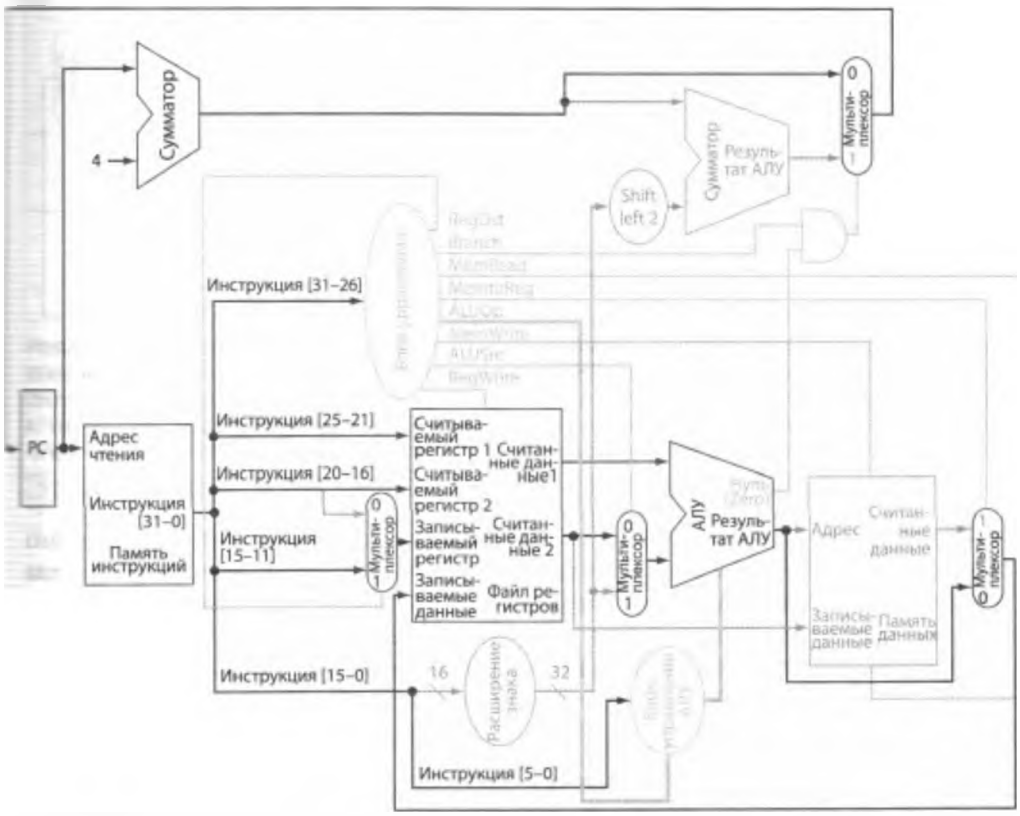


Рис. 4.16. Операционный блок в работе при выполнении инструкций R-типа, таких как add $t1,$t2,$t3. На рисунке выделены активные линии управления, части операционного блока и соединения

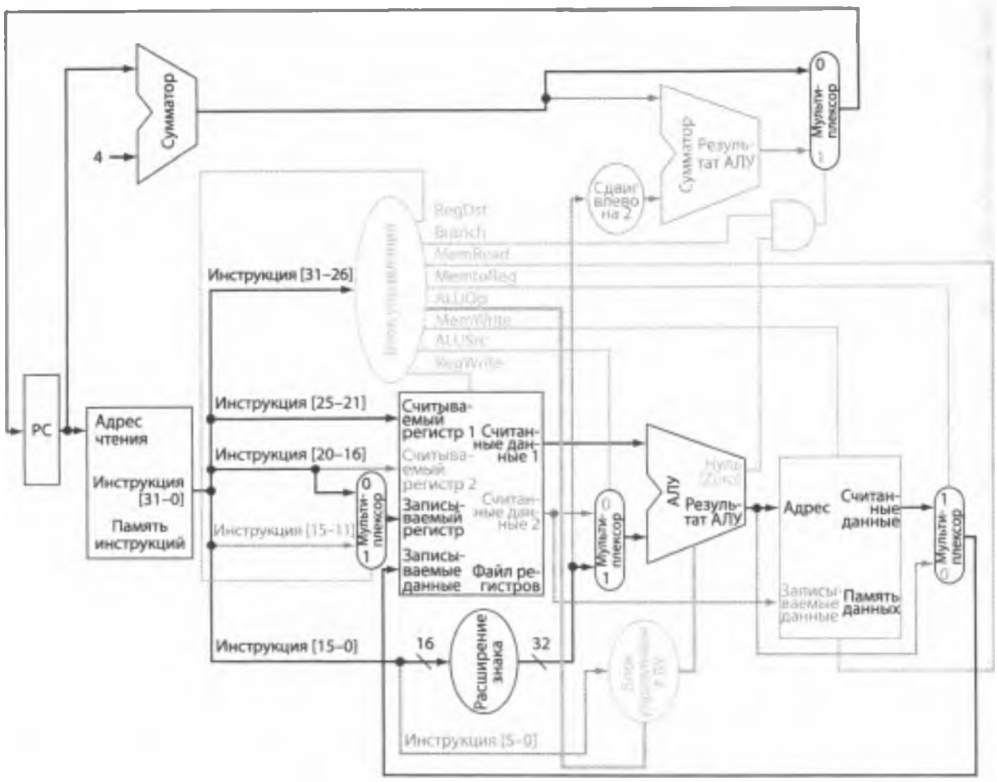


Рис. 4.17. Операционный блок, выполняющий инструкцию загрузки. Выделены задействованные линии управления, блоки и соединения. Инструкция сохранения выполняется весьма похожим образом. Основное различие в том, что показываются линии управления памятью, рассчитанные на запись, а не на чтение, считанное второе значение регистра используется для сохраняемых данных и не будет выполняться операция записи данных памяти в файл регистров

И наконец, в том же ключе мы можем показать работу инструкции условного перехода по равенству — branch-on-equal, вида `beq $t1,$t2,offset`. Она во многом похожа на выполнение инструкции R-формата, но выход АЛУ используется для определения, нужно ли записывать в счетчике команд (PC) значение PC + 4 или в него нужно записывать целевой адрес условного перехода. На рис. 4.18 показаны четыре шага выполнения:

1. Извлечение инструкции и приращение значения счетчика команд (PC).
2. Чтение из файла регистров значений двух регистров, `$t1` и `$t2`.
3. Выполнение АЛУ вычитания значений данных, считанных из файла регистров. Прибавление значения PC + 4 к прошедшим расширение знака младшим 16 разрядам инструкции (смещению), сдвинутым влево на два разряда; получение в качестве результата целевого адреса условного перехода.
4. Использование результата на выходе Нуль (`Zero`) АЛУ для принятия решения о том, результат с какого сумматора нужно сохранить в PC.

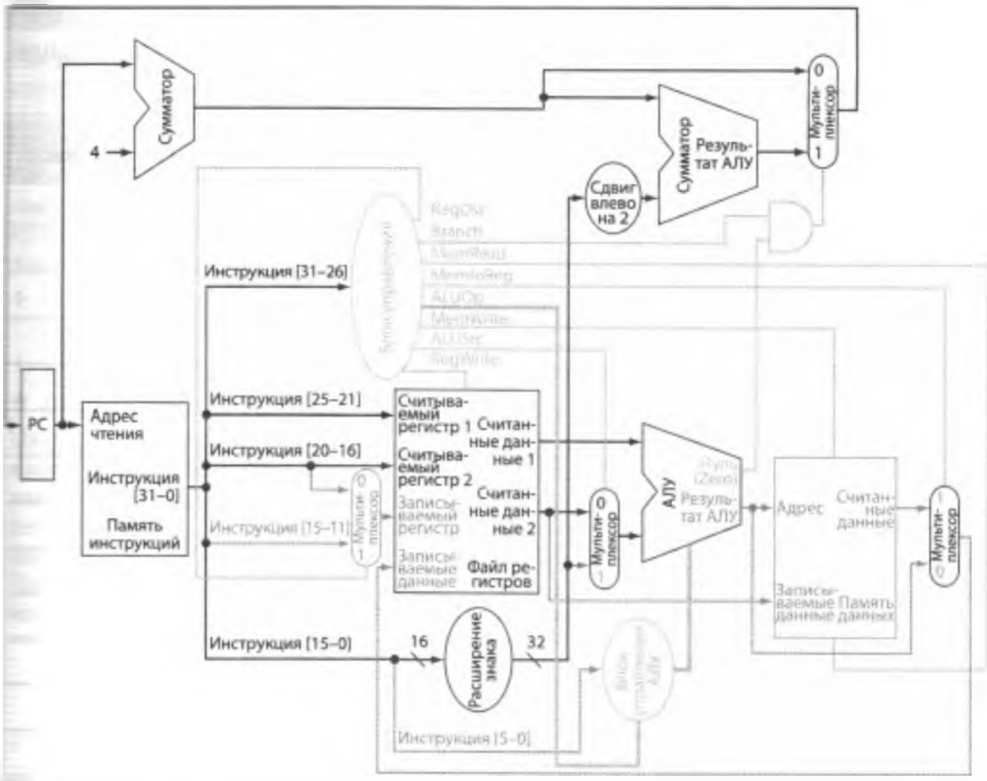


Рис. 4.18. Операционный блок, выполняющий инструкцию условного перехода по равенству. Выделены задействованные линии управления, блоки и соединения. После использования файла регистров и АЛУ для выполнения сравнения выход Нуль используется для выбора следующего значения счетчика команд из двух возможных вариантов

Завершение разработки системы управления

После пошагового рассмотрения работы инструкций продолжим реализацию системы управления. Функция управления может быть в точности определена с использованием содержимого таблицы, показанной на рис. 4.15. Выходные сигналы выставляются на линии управления, а на входе используется 6-разрядное поле opcode, `Op[5:0]`. Таким образом мы можем создать таблицу истинности для каждого выходного сигнала на основе двоичного кодирования полей opcode.

На рис. 4.19 показана логика, применяемая в блоке управления, в виде одной большой таблицы истинности, объединяющей все выходы и использующей в качестве входов разряды opcode. Она полностью определяет функцию управления, и мы можем реализовать ее непосредственно на логических элементах, работающих в автоматическом режиме.

Теперь, имея **реализацию, использующую один цикл** для набора из большинства основных инструкций MIPS, давайте добавим инструк-

Реализация, использующая один цикл
Также называется реализацией, использующей один тактовый цикл. Реализация, при которой инструкция выполняется за один тактовый цикл.

цию безусловного перехода, чтобы показать, как основной операционный блок и блок управления могут быть расширены для обработки других инструкций, имеющихся в наборе.

Вход или выход	Название сигнала	R-формат	lw	sw	beq
Входы	Op5	0	1	1	0
	Op4	0	0	0	0
	Op3	0	0	1	0
	Op2	0	0	0	1
	Op1	0	1	1	0
	Op0	0	1	1	0
Выходы	RegDst	1	0	X	X
	ALUSrc		1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	Mem Write	0		1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

Рис. 4.19. В таблице истинности полностью определена функция управления для простой реализации, использующей один тактовый цикл. В верхней половине таблицы даны комбинации входных сигналов, соответствующие четырем значениям поля opcode, по одному значению в каждом столбце, которые определяют выходные сигналы управления. (Не забудьте, что Op[5:0] соответствует разрядам 31:26 инструкции, составляющим поле op.) В нижней части таблицы даны выходные сигналы для каждого из четырех полей opcode. Таким образом, сигнал на выходе RegWrite выставляется для двух разных комбинаций входных сигналов. Если мы будем рассматривать только четыре значения поля opcode, показанные в данной таблице, то можно будет упростить таблицу истинности, используя в ее входной части безразличные элементы. Например, инструкцию R-формата можно определить с помощью выражения (НЕ Op5) И (НЕ Op2), поскольку этого достаточно для того, чтобы отличить инструкции R-формата от инструкций lw, sw и beq. Но мы не воспользуемся этим упрощением, поскольку в полной реализации используются все остальные значения opcode MIPS

Упражнение

Реализация безусловных переходов

На рис. 4.14 показана реализация многих инструкций, рассмотренных в главе 2. Но пропущен один класс инструкций — инструкция безусловного перехода. Расширьте операционный блок и блок управления, которые показаны на рис. 4.14, чтобы включить инструкцию безусловного перехода. Опишите, как выставить сигнал на любой новой линии управления.

Ответ

Инструкция безусловного перехода (jump), показанная на рис. 4.20, чем-то похожа на инструкцию условного перехода, но с другим вычислением целевого значения PC и без условия. Как и при условном переходе, младшие два разряда адреса перехода всегда равны 00_2. Следующие младшие 26 разрядов этого 32-разрядного адреса берутся из 26-разрядного поля непосредственного значения инструкции. Старшие 4 разряда адреса, который должен заменить значение PC, получаются добавлением 4 к прежнему значению PC, указывающему за инструкцию безусловного перехода. Таким образом, безусловный переход можно реализовать, сохраняя в PC цепочку из

- старших 4 разрядов текущего значения PC + 4 (это разряды 31:28 адреса непосредственно следующей инструкции);
- 26-разрядного поля непосредственного значения инструкции безусловного перехода;
- разрядов 00_2.

На рис. 4.21 показано добавление управляющей структуры для безусловного перехода к конструкции, показанной на рис. 4.14. Дополнительный мультиплексор выбирает источник для нового значения PC, который будет либо приращенным значением PC (PC + 4), либо целевым адресом условного перехода, либо целевым адресом безусловного перехода. Для дополнительного мультиплексора требуется еще один дополнительный сигнал управления. Этот сигнал управления (Jump) выставляется только при выполнении инструкции безусловного перехода (jump), то есть когда значение поля opcode равно 2.

Поле	000010	address
Позиции разрядов	31:26	25:0

Рис. 4.20. Формат инструкции для безусловного перехода (opcode = 2). Адрес назначения для инструкции безусловного перехода формируется путем объединения старших 4 разрядов текущего значения PC + 4 с 26-разрядным полем адреса (address) инструкции безусловного перехода и добавления 00 в качестве двух самых младших разрядов

Почему реализация, использующая один тактовый цикл, в настоящее время не используется

При всей работоспособности реализации, использующей один тактовый цикл, из-за своей неэффективности в современных конструкциях она использоваться не будет. Чтобы узнать, почему так сложилось, следует заметить, что тактовый цикл должен иметь одну и ту же продолжительность для каждой инструкции в этой завязанной на один цикл конструкции. Разумеется, тактовый цикл определяется наибольшим возможным путем данных в процессоре. Этот путь почти наверняка относится к инструкции загрузки, использующей пять функциональных блоков в следующей последовательности: память инструкций, файл регистров, АЛУ, память данных и файл регистров. Хотя показатель CPI при этом равен 1 (см. главу 1), общая производительность реализации, использующей один цикл, скорее всего, будет невысокой, поскольку тактовый цикл будет слишком продолжительным.

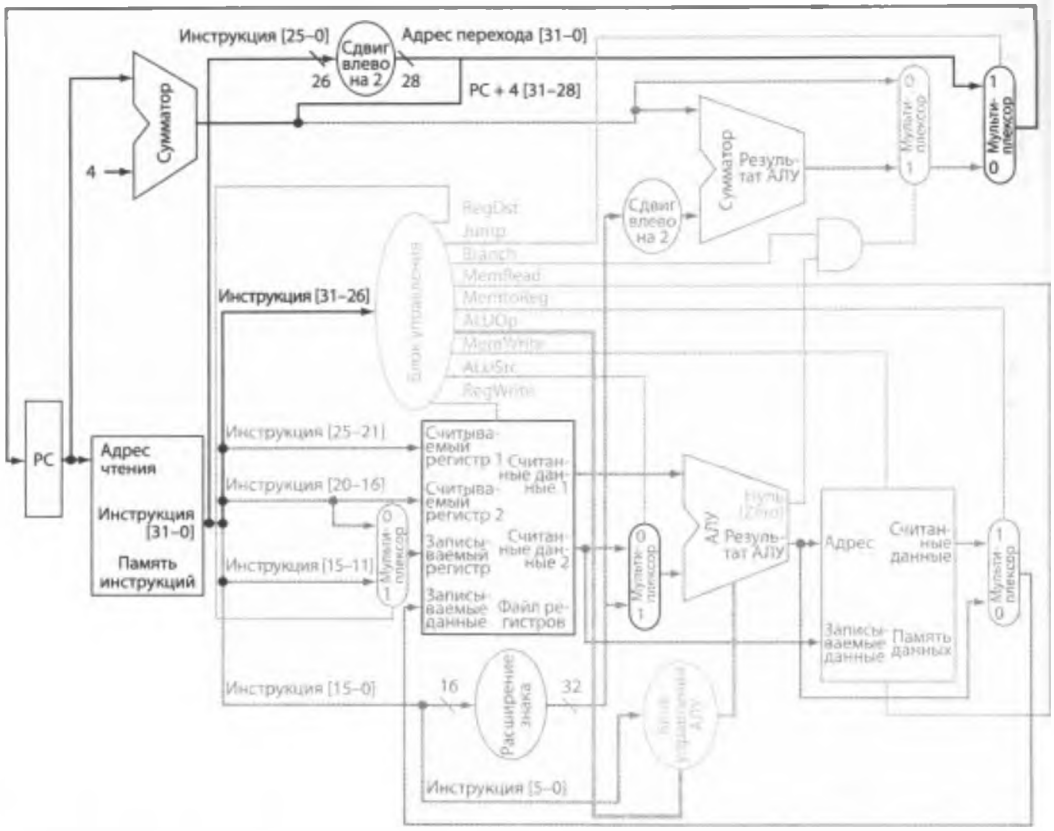


Рис. 4.21. Простые блок управления и операционный блок, расширенные для выполнения инструкции безусловного перехода. Дополнительный мультиплексор (в верхней части справа) используется для выбора между целевым адресом безусловного перехода и либо целевым адресом условного перехода, либо адресом инструкции, непосредственно следующей за текущей инструкцией. Этот мультиплексор управляется сигналом jump. Целевой адрес безусловного перехода получается за счет сдвига младших 26 разрядов инструкции безусловного перехода на 2 разряда влево, что на самом деле приводит к добавлению в самые младшие разряды значения 00 и последующего объединения старших 4 разрядов PC + 4 в качестве самых старших разрядов, из чего складывается 32-разрядный адрес

Расплата за использование конструкции одного цикла с фиксированной продолжительностью является весьма существенной, но может считаться приемлемой для небольшого набора инструкций. Исторически сложилось так, что эта технология реализации использовалась в самых первых компьютерах с очень простым набором инструкций. Но если попытаться реализовать блок арифметики с плавающей точкой или набор, состоящий из более сложных инструкций, то эффективной работы от конструкции одного цикла нам не добиться.

Поскольку нужно принять условие, что тактовый цикл равен продолжительности самой большой задержки при выполнении всех инструкций, бесполезно пытаться использовать технологии реализации, сокращающие задержку в общих случаях, но не улучшающую наихудшую продолжительность тактового цикла.

Таким образом, реализация, использующая один тактовый цикл, нарушает ключевой принцип конструирования из главы 2, который гласит, что часто встречающиеся задачи должны выполняться быстрее.

В следующем разделе будет рассмотрена еще одна технология реализации, называемая конвейеризацией, которая использует операционный блок, во многом похожий на операционный блок одного цикла, но гораздо более эффективный за счет высокой пропускной способности. Конвейеризация повышает эффективность за счет одновременного выполнения нескольких инструкций.

Самопроверка

Посмотрите на сигналы управления, показанные на рис. 4.19. Можно ли объединить некоторые из них? Могут ли некоторые выходы управляющих сигналов быть заменены обратным значением других сигналов? (Подсказка: обратите внимание на безразличные состояния.) Если да, то можно ли использовать один сигнал вместо другого без добавления инвертора?

4.5. Обзор конвейеризации

Никогда не тратьте время понапрасну.
Американская поговорка

Конвейеризация — это техника реализации, в которой выполнение нескольких инструкций осуществляется одновременно. В наши дни конвейеризация приобрела практически повсеместный характер.

Этот раздел во многом полагается на аналогию, позволяющую дать общее представление о понятиях и вопросах конвейеризации. Если вас интересует только общее представление, то можете обратить внимание лишь на этот раздел и пропустить разделы 4.10 и 4.11, представляющие усовершенствованную технологию конвейеризации, используемую в таких последних процессорах, как AMD Opteron X4 (Barcelona) или Intel Core. Если вы интересуетесь внутренним устройством компьютера, использующего конвейер, то это раздел будет неплохим введением для разделов 4.6–4.9.

Тот, у кого была когда-нибудь большая стирка, невольно использовал конвейеризацию. Без нее подход к стирке был бы следующим:

1. Загрузить стиральную машину одной порцией белья.
2. Когда стиральная машина завершит стирку, поместить мокрое белье в сушилку.
3. Когда сушилка завершит работу, положить высушенное белье на стол и сложить его.
4. Когда белье будет сложено, попросить кого-нибудь из совместно проживающих его унести.

Когда белье будет унесено, повторить все это с новой партией белья.

Конвейеризация
Технология реализации, в которой выполнение сразу нескольких инструкций происходит одновременно во многом напоминая сборочную линию.

Подход, использующий *конвейеризацию*, как показано на рис. 4.22, занимает намного меньше времени. Как только стиральная машина завершит стирку первой партии белья и оно будет помещено в сушилку, вы загрузите стиральную машину второй партией белья. Когда первая партия будет высушена, вы выложите ее на стол, чтобы приступить к складыванию, переместите партию мокрого белья в сушилку и загрузите в стиральную машину новой партией белья. Затем вы заставите кого-нибудь из совместно проживающих унести первую партию белья и приступите к складыванию второй партии, в сушилку будет загружена третья партия, а в стиральную машину — четвертая. К этому моменту все шаги — называемые в конвейеризации стадиями — будут выполняться параллельно. Поскольку для каждой стадии имеются отдельные ресурсы, задачи можно пустить по конвейеру.

Парадокс конвейеризации состоит в том, что время от помещения одного носка в стиральную машину и до его сушки, складывания и переноса в другое место при конвейеризации не сокращается; причина, по которой конвейеризация экономит время для множества загрузок состоит в том, что все работает в параллельном режиме, поэтому стирка множества партий завершается за час. Конвейеризация улучшает пропускную способность прачечной. Следовательно, конвейеризация не сократит время завершения одной загрузки прачечной, но при наличии множества загрузок прачечной повышение пропускной способности сокращает общее время работы.

Если все стадии занимают примерно одинаковое время и есть достаточно большой объем работы, то ускорение, полученное благодаря конвейеризации, эквивалентно количеству стадий в конвейере, в данном случае это число равно четырем: стирка, сушка, складывание и перенос к месту хранения. Благодаря этому конвейеризированная прачечная потенциально работает в четыре раза быстрее неконвейеризированной: 20 загрузок займут в пять раз больше времени, чем одна, в то время как 20 загрузок в последовательной прачечной займут в 20 раз больше времени, чем одна. На рис. 4.22 время сокращено всего в 2,3 раза, поскольку показано только четыре загрузки. Обратите внимание, что в начале и в конце рабочего времени конвейеризированной версии, показанной на рис. 4.22, конвейер не имеет полной загрузки; этот запуск и сворачивание работы оказывают влияние на производительность, если количество задач невелико по сравнению с количеством стадий в конвейере. Если количество загрузок намного больше четырех, тогда стадии будут заполнены большую часть времени и пропускная способность возрастет в 4 раза.

Сходные принципы применимы к процессорам, осуществляющим конвейерное выполнение инструкций. В классическом варианте выполнение MIPS-инструкций занимает пять шагов:

1. Извлечение инструкции из памяти.
2. Чтение регистров в процессе декодирования инструкции. Постоянный формат MIPS-инструкций позволяет вести одновременное чтение и декодирование.
3. Выполнение операции или вычисление адреса.
4. Доступ к операнду в памяти данных.
5. Запись результата в регистр.

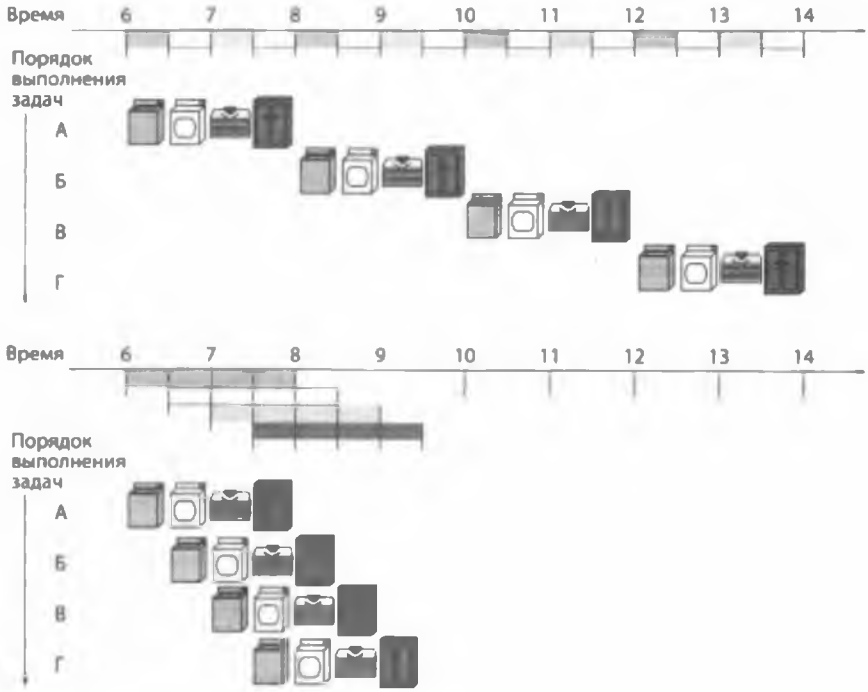


Рис. 4.22. Аналогия конвейеризации на примере прачечной. Энн, Брайан, Кэти и Дон накопили белье для стирки, сушки, складывания и переноса в шкаф. Для этой задачи каждая стадия: стирка, сушка, складывание и перенос на место хранения — занимает 30 минут. Последовательная работа прачечной займет 8 часов для 4 загрузок, а конвейеризированная — 3,5 часа. Мы показали конвейерную стадию для разных загрузок путем демонстрации копий четырех ресурсов на двумерной шкале времени, но на самом деле мы располагаем только одним экземпляром каждого ресурса

Следовательно, исследуемый в данной главе MIPS-конвейер имеет пять стадий. В следующем упражнении показано, что конвейеризация ускоряет выполнение инструкции, подобно тому как она ускоряла работу прачечной.

Упражнение

Сравнение производительности однотактной и конвейерной реализации

Чтобы конкретизировать дискуссию, давайте создадим конвейер. В данном упражнении и во всей остальной части главы мы ограничим наше внимание восемью инструкциями: загрузки слова (load word (`lw`)), сохранения слова (store word (`sw`)), сложения (add (`add`)), вычитания (subtract (`sub`)), операции И (AND (`and`)), операции ИЛИ (OR (`or`)), операции установки, если меньше чем (set less than (`slt`)) и операции условного перехода по равенству (branch on equal (`beq`)).

Сравните среднее время выполнения инструкций при однотактной реализации, в которой выполнение каждой инструкции занимает один тактовый цикл, и при конвейерной реализации. Рабочее время для основных функциональных блоков в этом упражнении составляет 200 пикосекунд (пс) для обращения к памяти, 200 пс для операции АЛУ и 100 пс для чтения или записи файла регистров. В однотактной модели выполнение каждой инструкции

занимает ровно один тактовый цикл, поэтому тактовый цикл должен быть растянут, чтобы приспособиться под самую медленную инструкцию.

Ответ

В табл. 4.4 показано время, необходимое для выполнения каждой из восьми инструкций. Однотактная конструкция должна допускать выполнение самой медленной инструкции — в табл. 4.4 это инструкция `lw`, — поэтому время, требуемое для выполнения каждой инструкции, составляет 800 пс. По аналогии с картиной, показанной на рис. 4.22, на рис. 4.23 приводится сравнение неконвейеризированного и конвейеризированного выполнения трех инструкций загрузки слова — load word. В соответствии с этим сравнением время между первой и четвертой инструкцией в конструкции, не использующей конвейер, составляет 3 × 800 пс, или 2400 пс. Все стадии конвейера занимают один тактовый цикл, поэтому тактовый цикл должен быть достаточно продолжительным, чтобы вместить в себя самую медленную операцию. Точно так же как однотактная конструкция должна быть равна продолжительности тактового цикла, равной времени выполнения самой медленной операции (800 пс), хотя некоторые инструкции могут выполняться быстрее, за 500 пс, при конвейеризированном выполнении тактовый цикл должен иметь время выполнения самой медленной операции, которое составляет 200 пс, хотя некоторые стадии выполняются и за 100 пс. И все же конвейеризация предлагает четырехкратный рост производительности: время между первой и четвертой инструкцией составляет 3 × 200 пс, или 600 пс.

Таблица 4.4. Общее время, затрачиваемое на выполнение каждой инструкции, вычисляется из времени, затрачиваемого на каждую ее составляющую. Это вычисление предполагает, что мультиплексоры, блоки управления обращения к PC и блоки расширения знаков не создают никаких задержек

Класс инструкции	Извлечение инструкции	Чтение регистра	Операция АЛУ	Обращение к данным	Запись регистра	Общее время
Загрузка слова — Load word (`lw`)	200 пс	100 пс	200 пс	200 пс	100 пс	800 пс
Сохранение слова — Store word (`sw`)	200 пс	100 пс	200 пс	200 пс		700 пс
R-формат (сложение — `add`, вычитание — `sub`, операция И — `AND`, операция ИЛИ — `OR`, операция установки, если меньше чем — `slt`)	200 пс	100 пс	200 пс		100 пс	600 пс
Условный переход — Branch (`beq`)	220 пс	100 пс	200 пс			500 пс

Представленную выше дискуссию о приращении скорости работы за счет использования конвейера можно превратить в формулу. Если стадии идеально сбалансированы, то время между инструкциями в процессоре с конвейером — при создании идеальных условий — равно:

$$\text{Время между инструкциями}_{\text{с конвейером}} = \frac{\text{Время между инструкциями}_{\text{без конвейера}}}{\text{Количество стадий конвейера}}.$$

При идеальных условиях и большом количестве инструкций ускорение за счет конвейеризации примерно равно количеству стадий конвейера; конвейерная линия, имеющая пять стадий, работает примерно в пять раз быстрее.

Формула предполагает, что конвейер, имеющий пять стадий, должен предложить почти пятикратное улучшение по отношению к 800 пс, или задать продолжительность тактового цикла 160 пс. Но упражнение показывает, что стадии могут быть сбалансированы совсем не идеально. Кроме того, конвейеризация не обходится без некоторых издержек, источник которых вскоре будет рассмотрен. Таким образом, время, затрачиваемое в среднем на инструкцию, превысит возможный минимум, и ускорение будет меньшим, чем количество стадий конвейера.

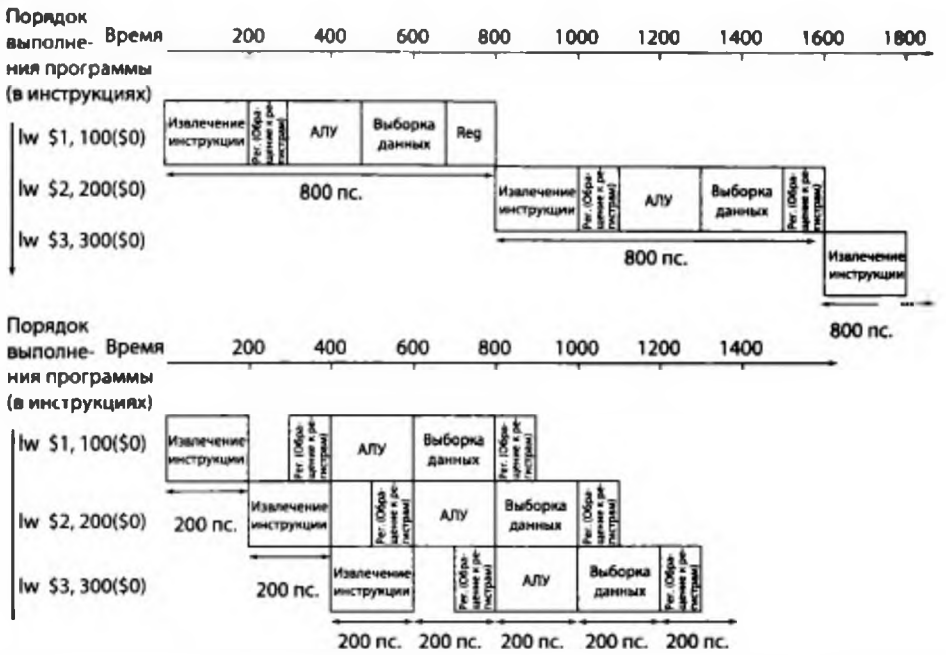


Рис. 4.23. Однотактное неконвейеризированное выполнение, показанное в верхней части, сравнивается с конвейеризированным выполнением, показанным в нижней части. Используются одинаковые компоненты оборудования, чье время работы показано в табл. 4.4. В данном случае наблюдается четырехкратное ускорение среднего времени между инструкциями с 800 пс до 200 пс. Сравните этот рисунок с рис. 4.22. В случае с прачечной мы предположили, что все стадии равны по продолжительности. Если бы сушка работала медленнее всего остального, то она задавала бы продолжительность стадии. Продолжительность стадии компьютерного конвейера также ограничена временем работы самого медленного ресурса — либо операции АЛУ, либо обращения к памяти. Мы предполагаем, что запись в файл регистров происходит в первой половине тактового цикла, а чтение из файла регистров происходит во второй его половине. Это предположение будет использоваться на протяжении всей этой главы

Кроме того, даже утверждение о четырехкратном улучшении производительности для нашего упражнения не отражено в общем времени выполнения для трех инструкций: оно составляет 1400 пс по сравнению с 2400 пс. Разумеется, причина заключается в небольшом количестве инструкций. А что получится, если увеличить количество инструкций? Мы можем расширить предыдущие цифры до 1 000 003 инструкций. В упражнение с использованием конвейера мы добавили бы 1 000 000 инструкций; каждая инструкция добавит к общему времени выполнения 200 пс. Общее время выполнения будет равно 1 000 000 × 200 пс + 1400 пс, или 200 001 400 пс. Без использования конвейеризации нужно сложить 1 000 000 инструкций, выполнение каждой из которых занимает 800 пс, поэтому общее время выполнения будет равно 1 000 000 × 800 пс + 2400 пс, или 800 002 400 пс. При таких условиях соотношение показателей общего времени выполнения для реальных программ на процессорах без конвейера и с конвейером приближается к соотношению времени между инструкциями:

$$\frac{800\,002\,400 \text{ пс}}{200\,001\,400 \text{ пс}} \approx \frac{800 \text{ пс}}{200 \text{ пс}} = 4{,}00.$$

Конвейеризация повышает производительность путем *увеличения пропускной способности по выполнению инструкций в отличие от сокращения времени выполнения отдельной инструкции*, но пропускная способность по выполнению инструкций является важным показателем, поскольку в реальной программе выполняются миллионы инструкций.

Разработка набора инструкций для конвейерного выполнения

Даже при таком простом объяснении конвейеризации мы можем проникнуть в конструирование набора инструкций MIPS, который был разработан для выполнения с использованием конвейера.

Во-первых, все инструкции MIPS имеют одну и ту же длину. Это ограничение существенно упрощает извлечение инструкций на первой стадии работы конвейера и их декодирование на второй стадии. В наборах инструкций наподобие x86, где длина инструкции варьируется от 1 до 17 байт, конвейеризация осуществляется значительно труднее. Последняя реализация архитектуры x86 на самом деле транслирует инструкции x86 в простые операции, похожие на MIPS-инструкции, а затем проводит конвейерное выполнение простых операций, а не настоящих инструкций x86 (см. раздел 4.10.)

Во-вторых, в MIPS используется не большое количество форматов, где поля регистров-источников в каждой инструкции находятся в одном и том же месте. Такая соразмерность означает, что вторая стадия может начинаться с чтения файла регистров одновременно с определением с помощью оборудования типа извлеченной инструкции. Если бы форматы MIPS-инструкций не обладали соразмерностью, стадию 2 пришлось бы разбивать, получая в результате шесть стадий работы конвейера.

В-третьих, операнды, находящиеся в памяти, появляются в MIPS только при загрузках или сохранениях. Это ограничение означает, что мы можем использовать исполнительную стадию для вычисления адреса памяти с обращением к памяти в следующей стадии. Если бы мы могли работать с операндами в памяти, как в x86, стадии 3 и 4 требовали бы расширения на стадию адреса, стадию памяти и исполнительную стадию.

В-четвертых, как говорилось в главе 2, операнды должны быть выровнены в памяти. Следовательно, нам нужно позаботиться об отдельной инструкции переноса данных, требующей двух обращений к памяти; запрошенные данные могут переноситься между процессором и памятью за одну стадию работы конвейера.

Конфликты, связанные с работой конвейера

При конвейеризации создаются ситуации, когда следующая инструкция не может выполняться в следующем тактовом цикле. Такие ситуации называются *конфликтами* и делятся на три разных типа.

Структурные конфликты

Первый из этих типов называется **структурным конфликтом**. Он означает, что оборудование не может поддержать комбинацию инструкций, которую мы хотим выполнить за один и тот же тактовый цикл. Структурный конфликт в прачечной возникнет, если будет использоваться комбинация стирки-сушки вместо отдельных стирки и сушки или если совместно с нами проживающие будут заняты чем-нибудь другим и не перенесут белье к месту его хранения. Тогда наши тщательно регламентированные планы работы конвейера будут нарушены.

Как уже говорилось, набор инструкций MIPS был разработан с учетом возможности конвейеризации, значительно упрощая разработчикам обход структурных конфликтов при конструировании конвейера. И тем не менее предположим, что у нас вместо двух блоков памяти используется единый блок. Если конвейер, показанный на рис. 4.23, получил четыре инструкции, мы увидим, что за один и тот же тактовый цикл первая инструкция обращается к данным в памяти, в то время как четвертая инструкция извлекает инструкцию из той же самой памяти. Без двух блоков памяти наш конвейер может столкнуться со структурным конфликтом.

Структурный конфликт
Ситуация, когда запланированная инструкция не может быть выполнена в подходящем тактовом цикле, потому что оборудование не поддерживает комбинацию инструкций, предназначенную для выполнения.

Конфликты данных

Конфликты данных возникают в том случае, когда конвейер может быть остановлен из-за того, что один шаг должен будет ожидать завершения другого. Предположим, что вы обнаружили на столе для складывания белья носок без соответствующей пары. Одним из путей решения проблемы может стать возвращение в свою комнату и поиск в комоде соответству-

Конфликт данных
Также называется конфликтом данных конвейера. Ситуация, когда запланированная инструкция не может быть выполнена в нужный тактовый цикл, поскольку данные, в которых она нуждается, еще недоступны.

ющей пары. Вполне очевидно, что во время этого поиска партия белья будет высушена и готова к складыванию, а та порция, которая будет выстирана и готова к сушке, вынуждена будет ожидать своей очереди.

В компьютерном конвейере конфликт данных возникает из-за зависимости от какой-нибудь инструкции из предыдущих, которая все еще находится в конвейере (в процессе стирки такие взаимоотношения никогда не возникают). Например, представим, что у нас есть инструкция сложения — add, за которой сразу же следует инструкция вычитания — sub, которая использует сумму ($s0):

```
add    $s0, $t0, $t1
sub    $t2, $s0, $t3
```

Без вмешательства конфликт данных мог бы просто остановить конвейер. Инструкция add не записывает свой результат, пока не настанет пятая стадия, а это означает, что мы впустую потеряем в конвейере три тактовых цикла.

Хотя в ликвидации всех подобных конфликтов мы можем положиться на компиляторы, результаты могут быть неудовлетворительными. Эти зависимости случаются довольно часто, и задержка является слишком продолжительной, чтобы ожидать, что компилятор избавит нас от этой дилеммы.

Основное решение основано на том наблюдении, что нам не нужно ждать, пока завершится выполнение инструкции, перед тем как попытаться решить проблему конфликта данных. Для показанной выше программной последовательности как только АЛУ выдаст сумму для инструкции сложения, мы можем предоставить ее в качестве входа для инструкции вычитания. Добавление дополнительного оборудования для извлечения недостающего элемента на ранней стадии из внутренних ресурсов называется **препровождением данных**, или **обходом**.

Упражнение

Препровождение данных при выполнении двух инструкций

Покажите для двух предыдущих инструкций, какие стадии конвейера будут соединены за счет препровождения данных. Используйте изображение на рис. 4.24 для представления пути данных во время выполнения пяти стадий работы конвейера. Выстройте копию пути данных для каждой инструкции по образцу конвейера прачечной, показанного на рис. 4.22.

Ответ

На рис. 4.25 показана связь для передачи значения в $s0 после исполнительной стадии инструкции add в качестве входных данных в исполнительную стадию инструкции sub.

В этом графическом представлении событий путь препровождения данных имеет смысл только в том случае, если стадия-получатель наступает позже стадии-источника. Например, не может быть приемлемого пути препровождения данных с выхода стадии обращения к памяти выполнения первой инструкции на вход исполнительной стадии следующей инструкции, поскольку это будет означать возвращение по времени назад.

Препровождение данных
Также называется обходом. Метод разрешения конфликта данных путем извлечения элемента недостающих данных из внутренних буферов вместо ожидания их поступления из программно-видимых регистров или памяти.

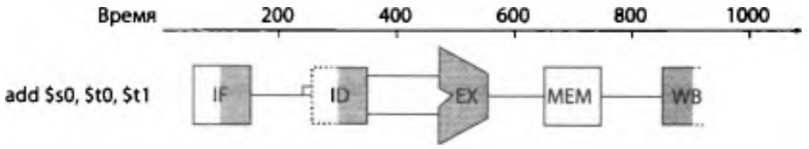


Рис. 4.24. Графическое представление конвейера инструкций, который можно вообразить похожим на конвейер прачечной, показанный на рис. 4.22. Здесь использованы обозначения, представляющие физические ресурсы, с аббревиатурами для стадий работы конвейера, используемыми во всем тексте данной главы. Для пяти стадий используются следующие обозначения: IF для стадии извлечения инструкции (прямоугольник, представляет память инструкций); ID для стадии декодирования инструкции — чтения файла регистров (показывает файл регистров в процессе чтения); EX для исполнительной стадии (показывает АЛУ); MEM для стадии обращения к памяти (прямоугольник показывает память данных); и WB для стадии обратной записи, показывает файл регистров в процессе записи). Серая заливка показывает элемент, используемый инструкцией. Блок MEM не закрашен, поскольку сложение не использует обращение к памяти данных. Заливка правой половины файла регистров или блока памяти означает, что на этой стадии происходит чтение элемента, а заливка левой половины означает, что на этой стадии происходит запись в элемент. Следовательно, правая половина ID на второй стадии закрашена, потому что происходит чтение файла регистров, а на пятой стадии закрашена левая половина WB, потому что файл регистров подвергается записи

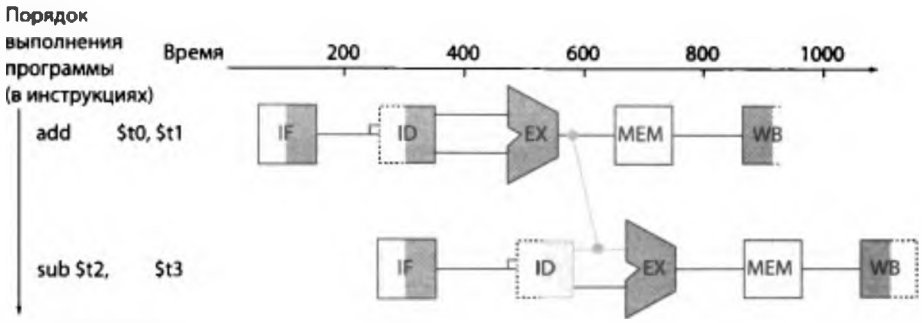


Рис. 4.25. Графическое представление препровождения данных. Связь показывает путь препровождения данных от выхода стадии EX при выполнении инструкции add на вход стадии EX для инструкции sub, изменяя значение регистра $s0, считанное на второй стадии выполнения инструкции sub

Препровождение данных работает очень хорошо и подробно рассматривается в разделе 4.7. Но оно не может предотвратить все замедления работы конвейера. Предположим, к примеру, что первой инструкцией было не сложение, а загрузка регистра $s0. Как можно себе представить, глядя на рис. 4.25, нужные данные будут доступны только после четвертой стадии выполнения первой инструкции, создавая зависимость, в силу которой вход в третью стадию выполнения инструкции sub запоздает. Следовательно, даже при препровождении данных одна из стадий замедлится в силу **конфликта загрузки-использования данных**, как показано на рис. 4.26. На этом рисунке изображено важное понятие, касающееся конвейера, которое официально называется **замедлением конвейера** (иногда его называют **пузырем**). Мы увидим замедления и в других местах конвейера. В разделе 4.7 показано, как можно справиться с подобными трудными случаями, используя либо

аппаратное обнаружение и замедления, либо программные средства, изменяющие порядок выполнения кода, чтобы избежать замедлений конвейера, связанных с загрузкой-использованием, как показывает данный пример.

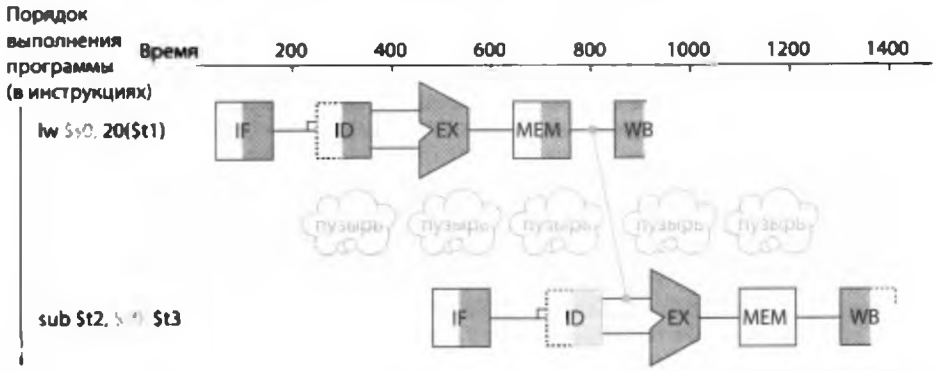


Рис. 4.26. Нам нужна задержка даже при препровождении данных, когда инструкция R-формата, следующая за загрузкой, пытается воспользоваться этими данными. Без задержки путь с выхода стадии обращения к памяти к входу исполнительной стадии должен идти назад по времени, что невозможно осуществить. Фактически этот рисунок является упрощением, поскольку мы не можем знать, пока инструкция вычитания не будет извлечена и декодирована, понадобится нам это замедление или нет. В разделе 4.7 рассмотрены подробности того, что на самом деле происходит в случае возникновения конфликта

Упражнение

Перестановка кода в целях задержек конвейера

Рассмотрим следующий фрагмент кода на языке C:

```
a = b + e;
c = b + f;
```

Так выглядит сгенерированный MIPS-код для этого фрагмента, при условии, что все переменные находятся в памяти и к ним можно обратиться с помощью смещения от значения регистра $t0:

```
lw $t1, 0($t0)
lw $t2, 4($t0)
add $t3, $t1,$t2
sw $t3, 12($t0)
lw $t4, 8($t0)
add $t5, $t1,$t4
sw $t5, 16($t0)
```

Найдите конфликты в этом фрагменте кода и переставьте инструкции так, чтобы избежать любых задержек конвейера.

Ответ

Обе инструкции сложения подвержены конфликтам в силу своей соответственной зависимости от непосредственно стоящих перед ними инструкций lw. Следует заметить, что развязка этой ситуации исключает ряд других потенциальных конфликтов, включая зависимость первой инструкции add от первой инструкции lw и любые конфликты для инструкций сохранения. Перемещение третьей инструкции lw на третью позицию в кодовом фрагменте исключает оба конфликта:

```
lw $t1, 0($t0)
lw $t2, 4($t0)
lw $t4, 8($t0)
add $t3, $t1,$t2
sw $t3, 12($t0)
add $t5, $t1,$t4
sw $t5, 16($t0)
```

На процессоре с конвейером и препровождением данных переставленная последовательность к тому же уменьшит количество циклов на два по сравнению с первоначальной версией.

Препровождение данных приводит к еще одному проникновению в MIPS-архитектуру в дополнение к тем четырем, которые рассматривались в разделе «Разработка набора инструкций для конвейерного выполнения». Каждая инструкция MIPS записывает не более одного результата и делает это на последней стадии конвейера. Препровождение данных затрудняется, если для него имеется сразу несколько результатов для одной инструкции или если требуется записать результат в начале выполнения инструкции.

Уточнение. Название «препровождение данных» появилось благодаря идее заблаговременной передачи результата от предыдущей инструкции последующей инструкции. Название «обход» появилось из-за передачи результата непосредственно в нужный блок, минуя файл регистров.

Конфликты управления

Третий тип конфликтов называется **конфликтом управления** и возникает из потребности принимать решение на основе результата выполнения одной инструкции во время выполнения другой инструкции.

Предположим, что наш персонал прачечной получил удачный заказ на стирку формы футбольной команды. Зная о несовершенстве нашей прачечной, нужно определить, достаточно ли моющих средств и соответствует ли температура воды, чтобы отстирать форму, но при этом не привести к ее ускоренному износу. В нашем прачечном конвейере нужно ждать завершения второй стадии, чтобы осмотреть высушенную форму и определить, нужно менять настройку стиральной машины или нет. Как следует поступить?

Вот как выглядят первые два решения конфликта управления в прачечной и их компьютерные эквиваленты.

Задержка. Нужно все делать последовательно, пока не будет высушена первая партия, а затем повторить процесс, пока не будет найдена правильная формула.

Конфликт загрузки-использования данных
Специфическая форма конфликта данных, при которой данные, загруженные инструкцией load, еще не стали доступны к тому моменту, когда они понадобились другой инструкции.

Замедление конвейера
Также называется пузырем. Задержка, инициированная в целях разрешения конфликта.

Конфликт управления
Также называется конфликтом условного перехода. Возникает, когда нужная инструкция не может быть выполнена в нужном тактовом цикле конвейера по причине того, что извлеченная инструкция не является той инструкцией, которая нужна, то есть поток адресов инструкций не соответствует ожиданиям конвейера.

Это консервативный вариант, надежно работающий, но медленный.

Эквивалентная задача по принятию решения в компьютере — это инструкция условного перехода. Учтите, что мы должны приступить к извлечению инструкции, следующей за инструкцией условного перехода, сразу же во время следующего тактового цикла. Тем не менее конвейер не располагает возможностью узнать, какой должна быть следующая инструкция, поскольку он только что получил из памяти инструкцию условного перехода! Как и в прачечной, одним из возможных решений станет немедленная задержка после извлечения инструкции условного перехода, в ожидании пока конвейер не определит исход условного перехода и не узнает, по какому адресу нужно будет извлечь инструкцию.

Предположим, что мы дополнили состав оборудования, получив возможность тестировать содержимое регистров, вычислять адрес условного перехода и обновлять значение PC во время второй стадии конвейера (подробности см. в разделе 4.8). Даже с этим дополнительным оборудованием конвейер, включающий условный переход, будет работать, как показано на рис. 4.27. Запуск инструкции lw, выполняемой, если переход не состоится, задерживается на дополнительный тактовый цикл в 200 пс.

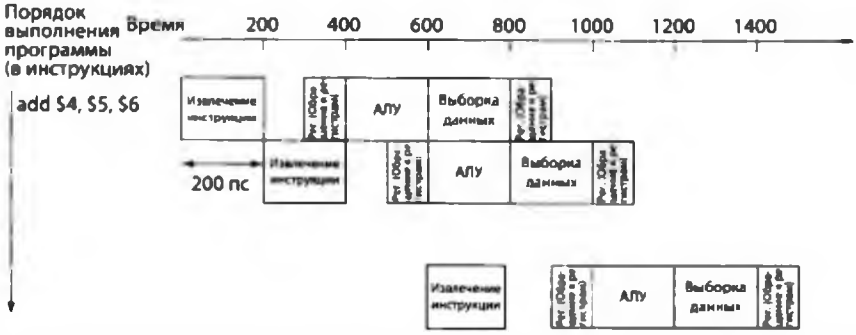


Рис. 4.27. Конвейер, демонстрирующий задержку при каждом условном переходе в качестве разрешения конфликта управления. В этом примере предполагается, что условный переход состоялся и инструкция, которая находится по целевому адресу условного перехода, является инструкцией OR. Это задержка конвейера на одну стадию, или пузырь, после условного перехода. В реальной ситуации процесс создания задержки, как мы увидим в разделе 4.8, выглядит немного сложнее. Но влияние на производительность будет таким же, как и при вставке пузыря

Упражнение

Выполнение «задержки при условном переходе»

Оцените влияние на показатель количества тактовых циклов на инструкцию (CPI) от задержки при условном переходе, при условии, что у всех остальных инструкций CPI равен 1.

Ответ

В главе 3 в табл. 3.11 показано, что условные переходы составляют 17% от общего количества инструкций, выполняемых в контрольных задачах SPECint2006. Поскольку все остальные запущенные инструкции имеют CPI, равный 1, и условные переходы затрачивают на задерж-

дополнительный тактовый цикл, мы определим CPI равным 1,17, а производительность по сравнению с идеальными условиями будет снижена в 1,17 раза.

Если условный переход во второй стадии разрешить не удается, что нередко бывает в длинных конвейерах, то, при наличии задержек на условный переход, мы столкнемся с более существенным снижением производительности. Цена подобного выбора для большинства компьютеров слишком высока, чтобы им воспользоваться, что служит мотивацией для второго варианта разрешения конфликта управления:

Прогнозирование. Если есть уверенность в правильности формулы для стирки формы, то нужно просто спрогнозировать, что она сработает, и приступить к стирке второй партии, ожидая просушки первой партии.

Если ваш прогноз верен, то этот выбор не приведет к замедлению работы конвейера. А если он неверен, вам нужно будет извлечь ту партию, которая стиралась, пока вы строили прогнозы.

Компьютер действительно использует *прогноз* для обработки условных переходов. Один из простых подходов заключается в постоянном прогнозировании того, что переходы не состоятся. Когда прогноз сбывается, конвейер работает на полной скорости. Его задержку вызывают только состоявшиеся переходы. Пример подобной работы показан на рис. 4.32.

Более сложная версия **прогнозирования условного перехода** предусматривает прогнозирование выполнения одних переходов и невыполнения других. В нашей аналогии темная, или домашняя, форма должна стираться по одной формуле, а светлая, или гостевая, форма — по другой. В программировании в нижней части цикла находятся условные переходы на вершину цикла. Поскольку они, скорее всего, состоятся и произойдет переход назад, для переходов на более ранние адреса мы можем всегда прогнозировать выполнение.

Такие жесткие подходы к прогнозированию условного перехода основываются на стереотипе поведения и не рассчитаны на индивидуальный характер отдельной инструкции условного перехода. Абсолютной противоположностью являются *динамические* аппаратные средства прогнозирования, которые строят свой прогноз в зависимости от поведения каждого условного перехода и способны изменять прогноз на переход за время жизни программы. Следуя нашей аналогии, при динамическом прогнозировании кто-нибудь будет следить за степенью загрязнения формы и прогнозировать особенности формулы, выстраивая следующий прогноз в зависимости от степени успеха предыдущего.

Одним из популярных подходов к динамическому прогнозированию условных переходов является ведение истории для каждого состоявшегося и несостоявшегося условного перехода, а затем использования недавнего поведения из прошлого для предсказания будущего. Как будет показано далее, рост количества и типа хронологических данных отразится на результате

Прогнозирование условного перехода Метод разрешения конфликта условного перехода, при котором для этого перехода прогнозируется заданный исход и соответствующее ему развитие событий вместо ожидания подтверждения фактического исхода.

динамических прогнозов условного перехода, позволяя достичь точности прогноза более 90% (см. раздел 4.8). Когда прогноз не подтвердится, система управление конвейером должна сделать так, чтобы инструкция, следующая непосредственно за неверно спрогнозированной инструкцией условного перехода, не повлияла на дальнейший ход событий, и обеспечить перезапуск конвейера с подходящего адреса условного перехода. В нашей аналогии с прачечной нужно остановить загрузку новых партий, чтобы можно было перезапустить обработку той партии, для которой была неверно спрогнозирована формула.

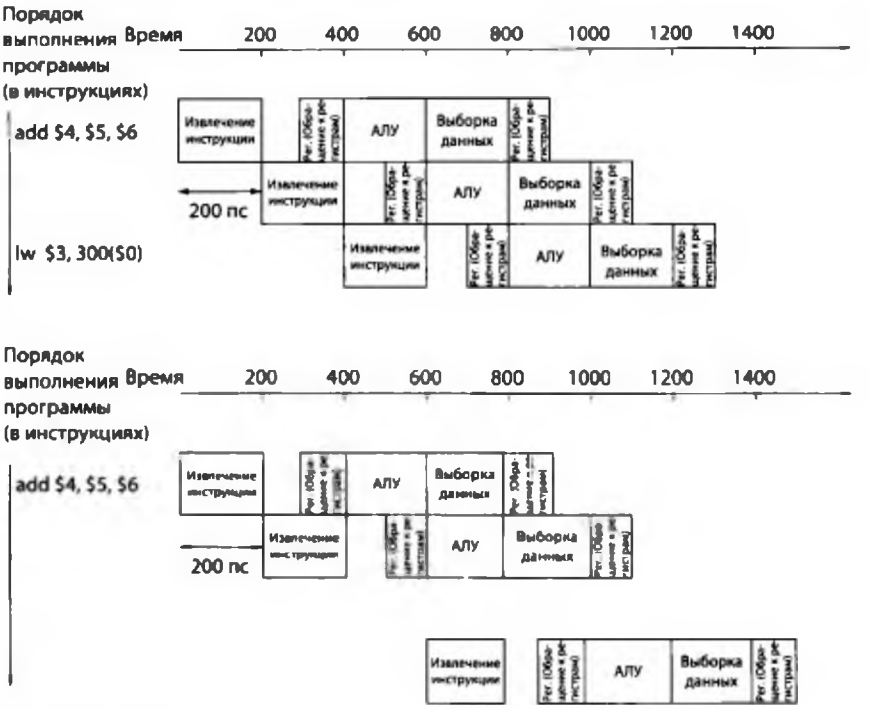


Рис. 4.28. Прогноз того, что условные переходы не будут выполнены, как средство разрешения конфликта управления. В верхней части показан конвейер, когда условный переход не состоялся. В нижней части показан конвейер, когда условный переход состоялся. Как было показано на рис. 4.27, вставка пузыря является упрощенной картиной того, что происходит на самом деле, по крайней мере, в период первого тактового цикла, который следует сразу же за условным переходом. Подробности будут рассмотрены в разделе 4.8

Как и в случае всех других способов разрешения конфликтов управления, более длинные конвейеры усложняют проблему, повышая, таким образом, ущерб от неверного прогнозирования. Разрешение конфликтов управления более подробно рассмотрено в разделе 4.8.

Уточнение. Существует еще и упомянутый выше третий подход к разрешению конфликта управления, который называется *отложенным решением*. В нашей аналогии при каждом таком решении в отношении прачечной нужно просто загрузить в стираль-

ную машину одежду, не являющуюся футбольной формой, ожидая, пока футбольная форма высохнет. Пока у вас будет достаточно нестираной одежды, которой не нужно стирание, это решение будет работать хорошо.

В компьютере это решение называется *отложенным условным переходом* и реально используется архитектурой MIPS. Отложенный условный переход всегда выполняет следующую сразу за ним инструкцию, а условный переход осуществляется *после* задержки на эту одну инструкцию. Это все скрыто от программиста на языке ассемблера MIPS, потому что ассемблер может автоматически расположить инструкции в определенном порядке, чтобы получить от условного перехода то поведение, которое нужно программисту. Программное обеспечение MIPS поставит сразу же после инструкции отложенного условного перехода инструкцию, не влияющую на переход, и состоявшийся условный переход изменит адрес той инструкции, которая *следует за* этой безопасной инструкций. В нашем примере, показанном на рис. 4.27, инструкция сложения перед инструкцией условного перехода не влияет на условный переход и может быть переставлена на место после перехода, чтобы полностью скрыть задержку перехода. Поскольку отложенные условные переходы полезны при коротких переходах, ни один из процессоров не использует отложенный условный переход, требующий более одного цикла. Для более длительных задержек, связанных с условными переходами, обычно используется прогнозирование условного перехода, основанное на работе аппаратуры.

Краткие выводы по общему обзору конвейера

Конвейеризация представляет собой технологию, использующую параллельное выполнение инструкций в их последовательном потоке. Она имеет существенное преимущество, поскольку, в отличие от программирования многопроцессорной системы, абсолютно невидима программисту.

В следующих разделах данной главы концепция конвейеризации будет рассмотрена с использованием поднабора MIPS-инструкций из однотактной реализации, рассмотренной в разделе 4.4, и показана упрощенная версия конвейера, предназначенного для их выполнения. Затем будут рассмотрены проблемы, связанные с конвейеризацией, и производительность, достижимая при типовых ситуациях.

Если вы захотите больше сконцентрироваться на программном обеспечении и влиянии конвейеризации на производительность, то теперь у вас вполне достаточно знаний, чтобы сразу перейти к разделу 4.10, представляющему такие передовые концепции конвейеризации, как суперскалярность и динамическая диспетчеризация, а в разделе 4.11 рассматриваются конвейеры недавно выпущенных микропроцессоров.

Ну а если вы хотите разобраться с тем, как реализована конвейеризация, и с трудностями разрешения конфликтов, вы можете перейти к изучению конструкции конвейеризированного операционного блока и основной системы управления, рассматриваемых в разделе 4.6. Вы сможете воспользоваться этими знаниями для изучения реализации препровождения данных и задержек в разделе 4.7. Затем

вы сможете перейти к чтению раздела 4.8, чтобы больше узнать о разрешении конфликтов условных переходов, и, наконец в разделе 4.9 посмотреть, как обрабатываются исключения.

Самопроверка

Для каждой из показанных ниже кодовых последовательностей определите, могут ли они вызвать задержку, можно ли ее избежать, используя только лишь препровождение данных, или выполнение может обойтись без задержки или препровождения данных.

Последовательность 1	Последовательность 2	Последовательность 3

lw $t0,0($t0)
add $t1,$t0,$t0
	add $t1,$t0,$t0
	addi $t2,$t0,#5
	addi $t4,$t1,#5
		addi $t1,$t0,#1
		addi $t2,$t0,#2
		addi $t3,$t0,#2
		addi $t3,$t0,#4
		addi $t5,$t0,#5

Представление о производительности программ

Не считая системы памяти, эффективная работа конвейера обычно является наиболее важным фактором в определении CPI процессора, а следовательно, его производительности. Как будет показано в разделе 4.10, разобраться в производительности современного конвейеризированного процессора с запуском сразу нескольких инструкций нелегко, и для этого нужно разбираться не только в проблемах, возникающих в простом конвейеризированном процессоре. Несмотря на это, структурные конфликты, а также конфликты данных и управления сохраняют важность как в простых, так и в более сложных конвейерах.

Для современных конвейеров структурные конфликты обычно связаны с блоком арифметики с плавающей точкой, который может быть конвейеризирован не полностью, а конфликты управления обычно составляют большинство проблем в целочисленных программах, которые склонны к частым условным переходам, а также к менее предсказуемым переходам. Конфликты данных могут стать узким местом производительности как в целочисленных программах, так и в программах с плавающей точкой. Зачастую проще разрешить конфликты данных в программах с плавающей точкой из меньшего количества условных переходов и более постоянных схем обращения к памяти, позволяющих компилятору заниматься планированием выполнения инструкций для того, чтобы избежать конфликтов. Подобная оптимизация в целочисленных программах дается сложнее, потому что у них менее постоянные схемы обращения к памяти и более частое использование указателей, работы компиляторов и оборудования.

Общее представление

Конвейеризация увеличивает количество одновременно выполняемых инструкций и скорость, с которой инструкции запускаются и завершаются. Конвейеризация не сокращает время, требуемое для завершения отдельной инструкции, которое также называется **латентностью**. Например, конвейер с пятью стадиями по-прежнему затрачивает 5 тактовых циклов для завершения инструкции. В понятиях, использованных в главе 1, конвейеризация повышает *пропускную способность* инструкций, а не *время выполнения*, или *латентность*, отдельно взятой инструкции.

Набор инструкций может либо упростить, либо усложнить жизнь разработчикам конвейера, которые должны справляться со структурными конфликтами, а также с конфликтами управления и данных. Прогнозирование условного перехода и препровождение данных помогают ускорить работу компьютера, сохраняя получение правильных ответов.

4.6. Операционный блок и система управления, оснащенные конвейером

Это меньшее, что бросается в глаза.
Таллула Бэнкхед
(*замечание, сделанное Александру Вуллкотту*)

На рис. 4.29 показан однотактный операционный блок из раздела 4.4 с обозначенными стадиями конвейера. Деление инструкций на пять стадий означает наличие конвейера с таким же количеством стадий, что, в свою очередь, означает, что на один тактовый цикл приходится до пяти инструкций. Таким образом, нам нужно разделить операционный блок на пять частей, где каждая из них названа в соответствии со стадией выполнения инструкции:

1. IF: Извлечение инструкции (Instruction fetch).
2. ID: Декодирование инструкции и чтение файла регистров (Instruction decode and register file read).
3. EX: Исполнение или вычисление адреса (Execution or address calculation).
4. MEM: Обращение к памяти данных (Data memory access).
5. WB: Обратная запись (Write back).

На рис. 4.29 эти пять компонентов примерно соответствуют нарисованному маршруту данных; инструкции и данные по мере завершения выполнения движутся в основном слева направо через пять стадий.

Вернемся к нашей прачечной: белье проходит стирку, сушку и дальнейшие операции по мере продвижения по линии, и оно никогда не возвращается назад.

Латентность (конвейерная)
Количество стадий в конвейере или количество стадий между двумя инструкциями в процессе выполнения.

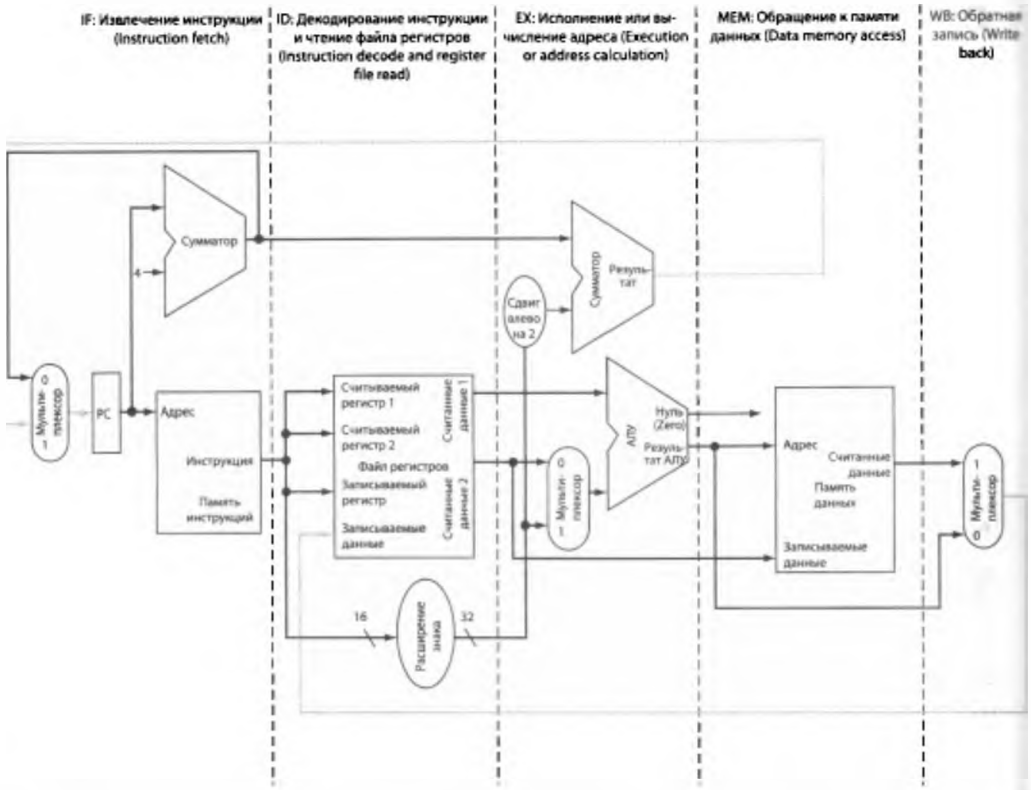


Рис. 4.29. Однотактный операционный блок из раздела 4.4 (похожий на тот, что изображен на рис. 4.14). Каждый шаг инструкции может быть отображен на операционный блок слева направо. Единственное исключение касается обновления счетчика команд (PC) и шага обратной записи, показанных серым цветом, которые оправляют либо результат АЛУ, либо данные из памяти влево, чтобы они были записаны в файл регистров. (Обычно выделение серым цветом используется для линий управления, но здесь оно использовано для линий данных.)

Но здесь есть два исключения из этого течения инструкций, идущего слева направо:

- стадия обратной записи, в которой результат помещается обратно в файл регистров, находящийся в середине операционного блока;
- выбор следующего значения PC между приращенным значением PC и адресом перехода из MEM-стадии.

Продвижение данных справа налево не влияет на текущую инструкцию; это реверсивное перемещение данных влияет в конвейере только на более поздние инструкции. Следует заметить, что первое продвижение данных справа налево может привести к конфликтам данных, а второе — к конфликтам управления.

Один из способов показать, что происходит при конвейерном выполнении, заключается в представлении, что у каждой инструкции есть свой собственный операционный блок, и в последующем размещении этих операционных блоков

на шкале времени для демонстрации их взаимоотношений. На рис. 4.30 показано выполнение инструкций с рис. 4.23 путем изображения их собственных операционных блоков на общей шкале времени. Для показа взаимоотношений на рис. 4.30 мы воспользовались стилизованной версией операционного блока с рис. 4.29.

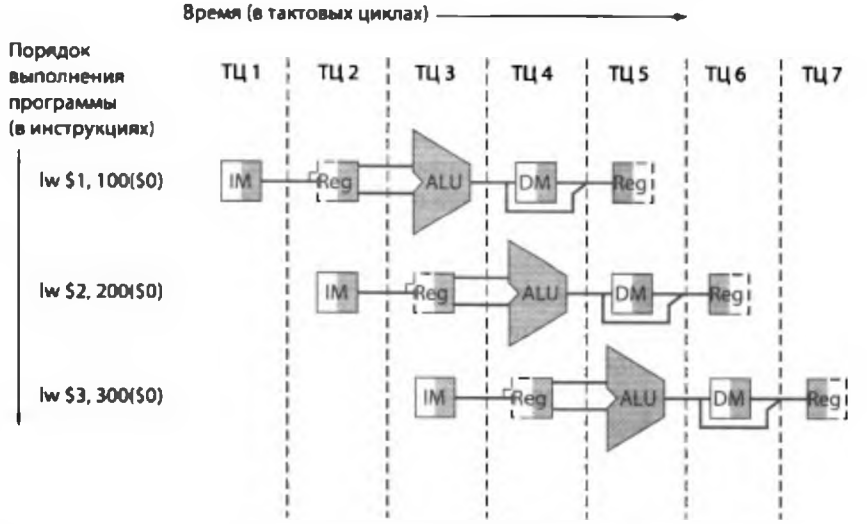


Рис. 4.30. Инструкции, выполняемые с использованием однотактного операционного блока с рис. 4.29, при условии конвейерного выполнения. Как и на рис. 4.24–4.26, у каждой инструкции есть свой собственный операционный блок, и каждая его часть закрашена в соответствии с применением. В отличие от этих рисунков каждой стадии соответствует физический ресурс (рис. 4.29). Аббревиатура IM представляет память инструкций (instruction memory) и счетчик команд (PC) на стадии извлечения инструкции. Reg означает файл регистров и расширитель знака на стадии декодирования инструкции или чтения файла регистров (ID) и т. д. Чтобы выдержать верный порядок по времени, этот стилизованный операционный блок разбивает файл регистров на две логические части: чтение из регистров в процессе извлечения регистра (ID) и запись в регистры в процессе обратной записи (WB). Двойное использование видно по незакрашенной левой половине файла регистров, имеющей прерывистую границу на стадии ID, когда в него не ведется запись, и незакрашенной правой половине, имеющей прерывистую границу на стадии WB, когда из него не ведется чтение. Как и раньше, мы предполагаем, что запись в файл регистров ведется в первой половине тактового цикла, а чтение из него — во второй половине

Глядя на рис. 4.30, можно подумать, что для трех инструкций нужны три операционных блока. Но на самом деле мы добавили регистры для хранения данных, поэтому части единого операционного блока могут совместно использоваться в процессе выполнения инструкции.

Например, как показано на рис. 4.30, память инструкций задействована только во время одной из пяти стадий выполнения инструкции и может совместно использоваться следующими инструкциями во время других четырех стадий. Чтобы сохранить в памяти значение отдельной инструкции для ее остальных четырех стадий, значение, считанное из памяти инструкций, должно быть сохранено в регистре. Подобные аргументы относятся к каждой стадии конвейера, поэтому регистры должны быть помещены везде, где есть разделительные линии. Возвращаясь

к нашей аналогии с прачечной, у нас должна быть корзина между каждой парой стадий, в которой белье хранилось бы для следующего шага.

На рис. 4.31 показан конвейеризированный операционный блок с выделенными регистрами конвейера. Все инструкции продвигаются во время каждого тактового цикла от одного конвейерного регистра к другому. Имена регистрам даны по названиям двух стадий, разделенных данным регистром. Например, регистр конвейера, расположенный между стадиями IF и ID, называется IF/ID.

Разумеется, каждая инструкция обновляет значение счетчика команд (PC), либо приращивая его значение, либо присваивая ему целевой адрес условного перехода. Поэтому PC может рассматриваться как регистр конвейера: один из тех, который способствует работе конвейера на стадии IF. Но в отличие от закрашенных регистров конвейера, показанных на рис. 4.31, PC является частью видимой архитектурной единицы; его содержимое должно быть сохранено при возникновении исключения, в то время как содержимое регистров конвейера должно быть сброшено. По аналогии с прачечной, счетчик команд должен соответствовать той корзине, в которой содержится партия нестираного белья перед шагом стирки.

Чтобы изложить принцип работы конвейера, во всем тексте данной главы мы будем показывать последовательность рисунков, демонстрирующую его работу. Может показаться, что дополнительные страницы потребуют от вас намного больше времени, чтобы в них разобраться. Но эти опасения совершенно напрасны; вопреки возможным ожиданиям, изучение последовательных изображений займет намного меньше времени, поскольку у вас появится возможность их сравнивать, чтобы увидеть изменения, происходящие с каждым тактовым циклом. В разделе 4.7 дается описание всего происходящего при возникновении конфликта данных между инструкциями конвейера; поэтому на данный момент этот вопрос мы пропустим.

На рис. 4.32–4.34 изображена первая последовательность, на которой выделена активная часть операционного блока в момент, когда инструкция загрузки проходит пять стадий конвейерного выполнения. Инструкция загрузки показана первой, потому что при ее выполнении активны все пять стадий. Как и на рис. 4.24–4.26, мы выделили *правую половину* регистров (или памяти), когда они находятся в *процессе чтения*, и выделили их *левую половину*, когда они находятся в *процессе записи*.

Обратите внимание на то, что в конце стадии обратной записи регистра конвейера нет. Все инструкции должны обновить какое-нибудь состояние в процессоре — файл регистров, память или счетчик команд (PC), — поэтому отдельный регистр конвейера для обновляемого состояния будет не нужен. Например, инструкция загрузки поместит свой результат в один из 32 регистров, а одна из следующих инструкций, которой понадобятся данные, будет просто читать соответствующий регистр.

На каждом рисунке мы показали аббревиатуру инструкции `lw` с названием той стадии конвейера, которая находится в активном состоянии. Рассматриваются следующие пять стадий:

1. *Извлечение инструкции*. В верхней части рис. 4.32 показана инструкция в процессе ее чтения из памяти с использованием адреса в счетчике команд (PC),

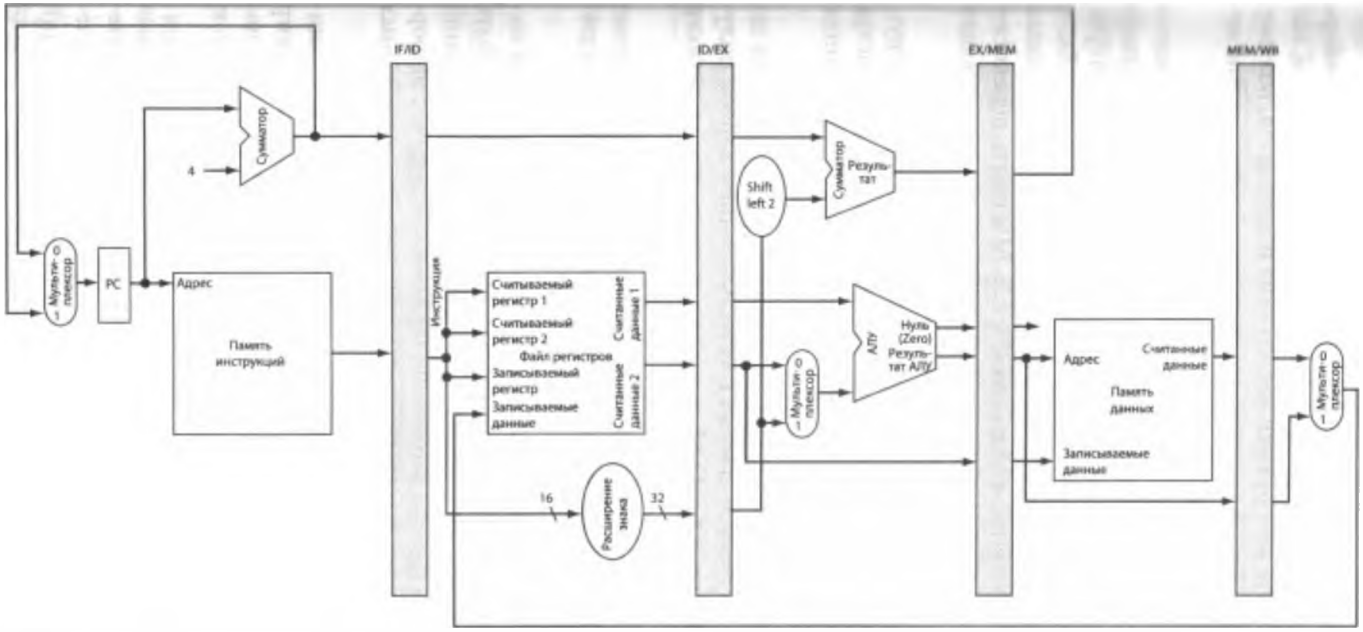


Рис. 4.31. Конвейеризированная версия операционного блока с рис. 4.29. Выделенные регистры конвейера разделяют все стадии конвейера. Они обозначены по именам тех стадий, которые ими разделены; например, первый из них имеет обозначение IF/ID, потому что он разделяет стадии извлечения инструкции (instruction fetch) и декодирования инструкции (instruction decode). Регистры должны иметь достаточную ширину, чтобы хранить все данные, соответствующие проходящим через них линиям. Например, регистр IF/ID должен иметь ширину 64 разряда, потому что он должен содержать и 32-разрядную инструкцию, извлеченную из памяти, и приращенное значение 32-разрядного адреса счетчика команд (PC). Далее регистры будут расширяться, но на данный момент остальные три регистра конвейера содержат 128, 97 и 64 разряда соответственно

с последующим помещением ее в регистр конвейера IF/ID. Адрес в PC увеличивается на 4, а затем записывается назад в PC в готовности к следующему тактовому циклу. Этот увеличенный адрес также сохраняется в регистре конвейера IF/ID на случай, если он понадобится в дальнейшем для такой инструкции, как beq. Компьютер не может знать, инструкция какого типа извлекается, поэтому он должен подготовиться для любой инструкции, передавая потенциально необходимую информацию по конвейеру.

2. *Декодирование инструкции и чтение из файла регистров.* В нижней части рис. 4.32 показана та часть регистра конвейера IF/ID, содержащая инструкцию и предоставляющая 16-разрядное поле непосредственного значения, которое проходит расширение до 32 разрядов, и номера регистров, чтобы можно было прочитать содержимое двух регистров. Все три значения, наряду с увеличенным адресом PC, сохраняются в регистре конвейера ID/EX. Мы опять переносим все, что может понадобиться любой инструкции в течение одного из следующих тактовых циклов.
3. *Исполнение или вычисление адреса.* На рис. 4.33 показано, что инструкция загрузки считывает содержимое регистра 1 и непосредственное значение из регистра конвейера ID/EX и складывает их, используя АЛУ. Эта сумма помещается в регистр конвейера EX/MEM.
4. *Обращение к памяти.* В верхней части рис. 4.34 показано, что инструкция загрузки производит чтение из памяти данных с использованием адреса, полученного из регистра конвейера EX/MEM, и данные загружаются в регистр конвейера MEM/WB.
5. *Обратная запись.* В нижней части рис. 4.34 показан завершающий шаг: чтение данных из регистра конвейера MEM/WB и запись их в файл регистров, приведенный в центре рисунка.

Этот маршрут выполнения инструкции загрузки показывает, что любая информация, необходимая на одной из последующих стадий конвейера, должна быть передана стадии через регистр конвейера. Отслеживание выполнения инструкции сохранения показывает схожесть работы данной инструкции с передачей информации для последующих стадий конвейера. Вот как выглядят пять конвейерных стадий для инструкции сохранения:

1. *Извлечение инструкции.* Инструкция считывается из памяти с использованием адреса в счетчике команд (PC), с последующим помещением ее в регистр конвейера IF/ID. Эта стадия предшествует идентификации инструкции, поэтому в верхней части рис. 4.32 продемонстрировано сохранение, полностью идентичное загрузке.
2. *Декодирование инструкции и чтение из файла регистров.* Инструкция, которая находится в регистре конвейера IF/ID, предоставляет номера регистров для чтения двух регистров и расширяет знак 16-разрядного непосредственного значения. ID/EX. В нижней части рис. 4.32, отображающего выполнение инструкций загрузки, также показаны и операции второй стадии выполнения инструкций сохранения. Эти первые две стадии выполняются всеми инструкциями, поскольку на этом этапе тип инструкции неизвестен.

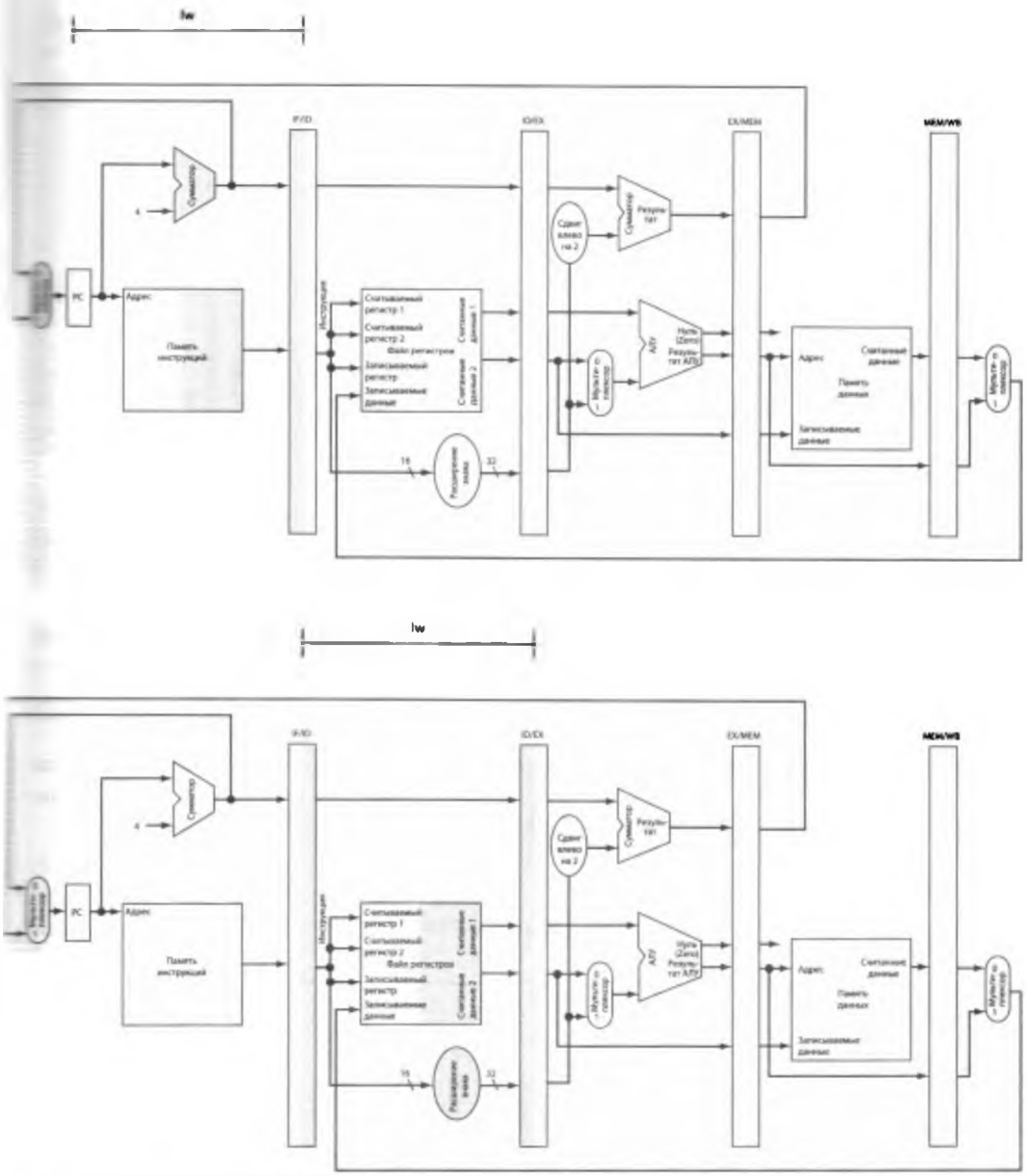


Рис. 4.32. IF и ID: первая и вторая стадии конвейера, используемые при выполнении инструкции с выделенными активными частями операционного блока, показанного на рис. 4.31. Здесь действует соглашение по выделению той или иной части, аналогичное тому, которое использовалось на рис. 4.24. Как описывалось в разделе 4.2, неразберихи при чтении и записи регистров здесь не возникает, потому что содержимое изменяется только при прохождении фронта синхроимпульса. Хотя на стадии 2 инструкция загрузки нуждается только в верхней части регистра, процессор не знает, какая именно инструкция декодируется, поэтому он осуществляет расширение знака 16-разрядной константы и считывает оба регистра в регистр конвейера ID/EX. Все три операнда нам не нужны, но их хранение упрощает управление

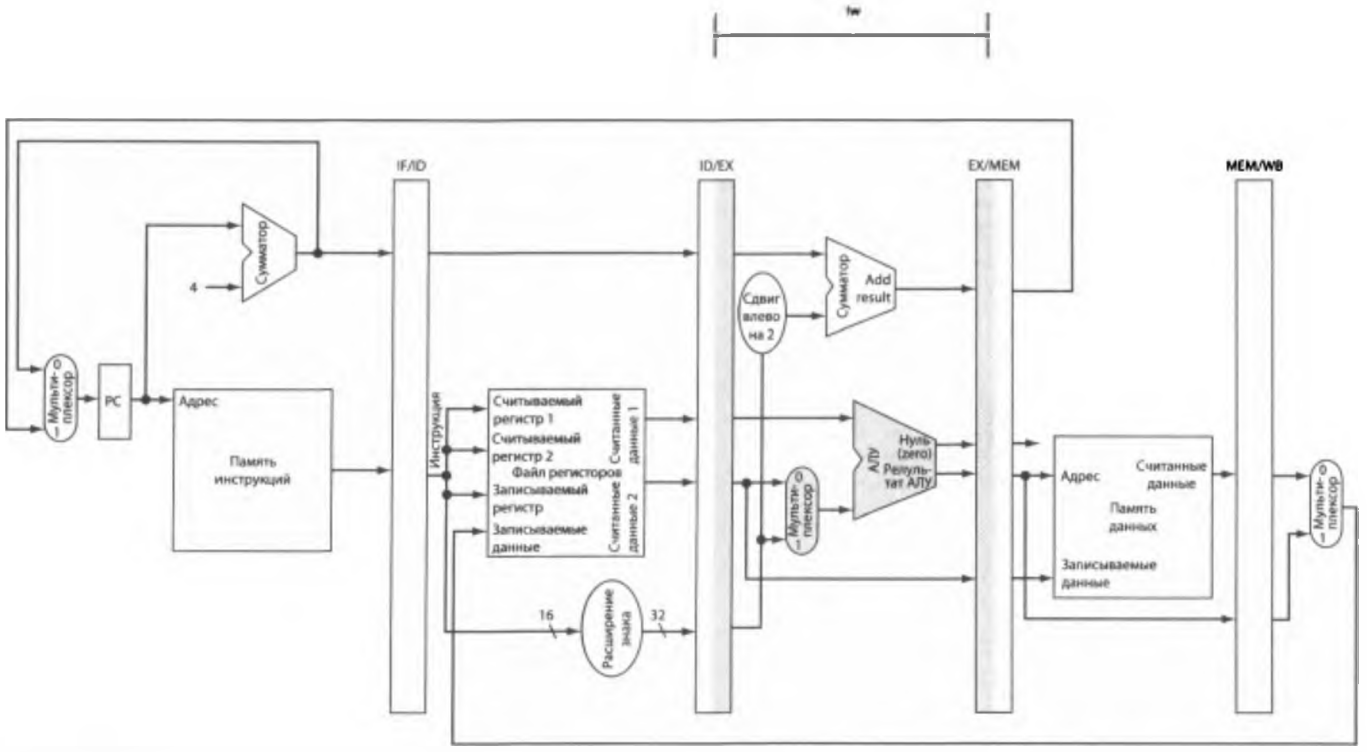


Рис. 4.33. EX: третья стадия конвейерного выполнения инструкции загрузки с выделенными частями операционного блока с рис. 4.31, используемыми на данной стадии. Значение регистра складывается с прошедшим расширение знака непосредственным значением, и сумма помещается в регистр конвейера EX/MEM

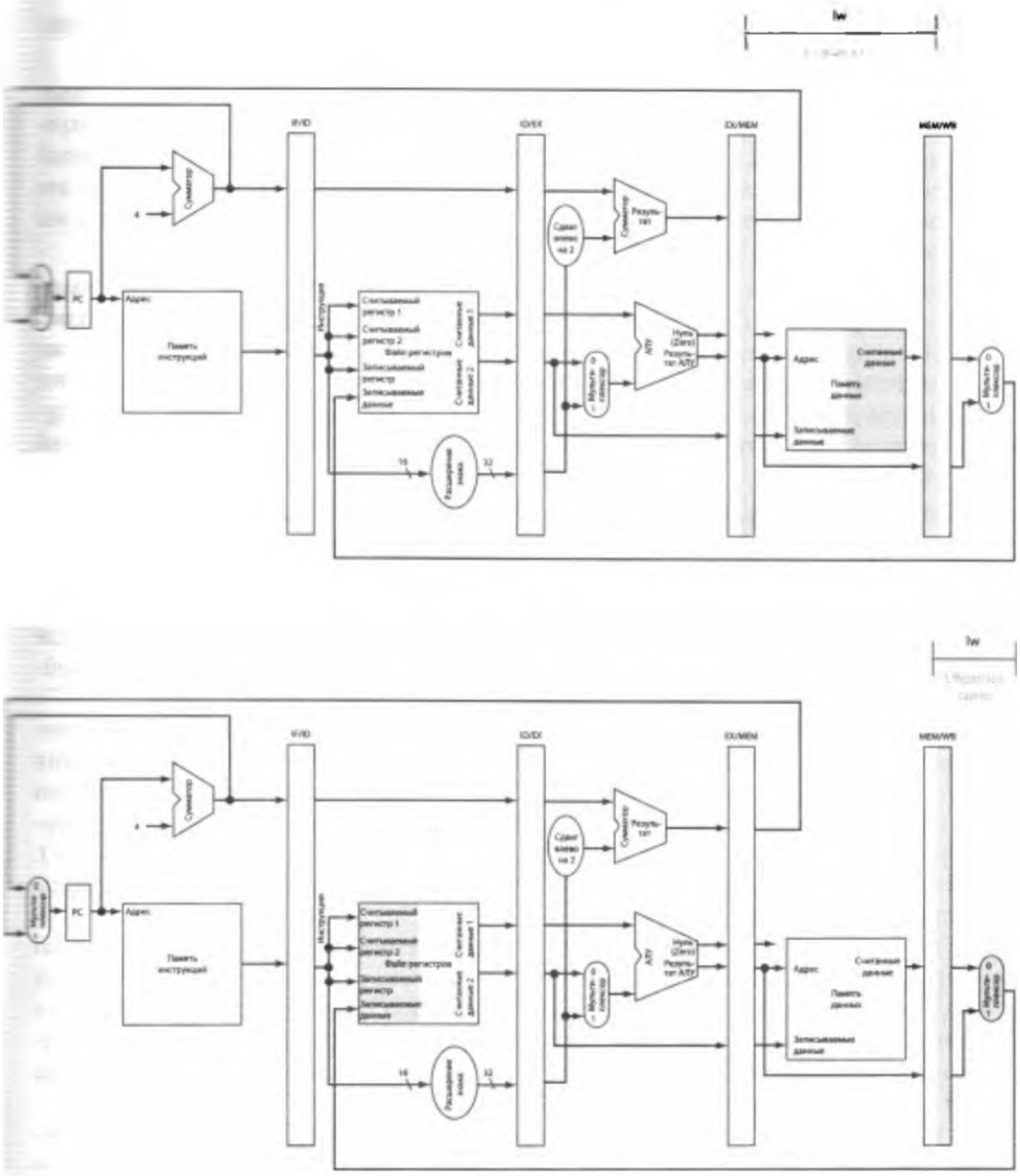


Рис. 4.34. MEM и WB: четвертая и пятая стадии конвейерного выполнения инструкции загрузки, с выделенными частями операционного блока с рис. 4.31, используемыми на данной стадии. Информация из памяти данных считывается с использованием адреса из регистра конвейера EX/MEM, и данные помещаются в регистр конвейера MEM/WB. Затем данные считываются из регистра конвейера MEM/WB и записываются в файл регистров, показанный в середине операционного блока. Примечание: этот недостаток в данной конструкции устраняется на рис. 4.37

3. *Исполнение или вычисление адреса*. На рис. 4.35 показан третий шаг; действующий адрес помещается в регистр конвейера EX/MEM.
4. *Обращение к памяти*. В верхней части рис. 4.36 показаны данные, записываемые в память. Обратите внимание на то, что регистр, содержащий данные для сохранения, был считан на более ранней стадии и сохранен в ID/EX. Единственный способ сделать данные доступными в процессе выполнения стадии MEM — это поместить их в регистр конвейера EX/MEM на стадии EX, точно так же, как мы сохраняли в EX/MEM действующий адрес.
5. *Обратная запись*. В нижней части рис. 4.36 показан завершающий шаг сохранения. Для этой инструкции на стадии обратной записи ничего не происходит. Поскольку любая инструкция, находящаяся за инструкцией сохранения, уже выполняется, у нас нет способов ускорения таких инструкций. Следовательно, инструкция проходит стадию, даже если ей нечего там делать, поскольку следующие за ней инструкции уже находятся в процессе выполнения с максимальной скоростью.

Инструкция сохранения показывает, что для прохождения чего-либо от ранней к более поздней стадии конвейера информация должна быть помещена в регистр конвейера, в противном случае информация будет потеряна при входе в стадию конвейера следующей инструкции. Для инструкции сохранения нам нужно передать данные, считанные из одного из регистров на стадии ID, в стадию MEM, где они будут сохранены в памяти. Данные сначала были помещены в регистр конвейера ID/EX, а затем переданы в регистр конвейера EX/MEM.

Загрузка и сохранение иллюстрируют второй ключевой момент: каждый логический компонент операционного блока, например память инструкций, порты чтения регистра, АЛУ, память данных и порт записи регистра, могут использоваться только во время *одной* стадии конвейера. В противном случае мы столкнемся со *структурным конфликтом* (см. раздел «Конфликты, связанные с работой конвейера»). Следовательно, эти компоненты и их системы управления должны быть связаны только с одной стадией конвейера.

Теперь мы должны выявить ошибку в конструкции, выполняющей инструкцию загрузки. А вы заметили эту ошибку? Какой регистр изменился на финальной стадии загрузки? Если говорить конкретнее, то какая инструкция предоставляет номер регистра, в который записываются данные? Инструкция в регистре конвейера IF/ID предоставляет номер регистра, в который будет вестись запись, но ведь эта инструкция появляется значительно *позже* инструкции загрузки!

Следовательно, нам нужно сохранить номер регистра-получателя в инструкции загрузки. Аналогично тому как инструкция сохранения передает *содержимое* регистра из регистра конвейера ID/EX в регистр конвейера EX/MEM с целью его использовании на стадии MEM, инструкция загрузки должна передать *номер* из регистра конвейера ID/EX через регистр конвейера EX/MEM в регистр конвейера MEM/WB с целью его использования на стадии WB. Еще один способ осмыслить передачу номера регистра заключается в том, что для общего использования конвейеризированного операционного блока нам нужно так сохранить инструкцию, прочитанную на стадии IF, чтобы каждый регистр конвейера содержал ту часть инструкции, которая необходима для данной и для последующих стадий.

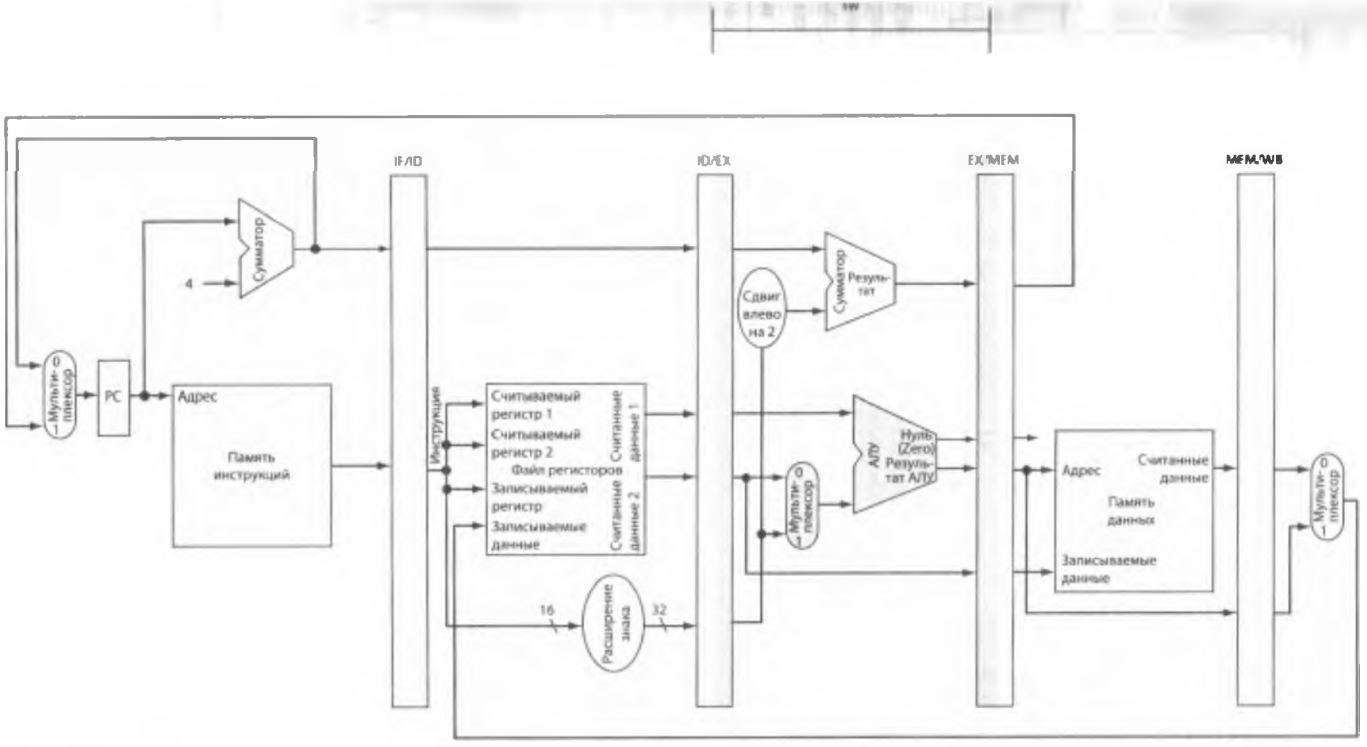


Рис. 4.35. EX: третья стадия конвейерного выполнения инструкции сохранения. В отличие от третьей стадии выполнения инструкции загрузки, показанной на рис. 4.33, в регистр конвейера EX/MEM загружается значение второго регистра, для того чтобы оно было использовано в следующей стадии. Хотя нам не помешало бы постоянно записывать значение этого второго регистра в регистр конвейера EX/MEM, мы записываем его только при выполнении инструкции сохранения, чтобы было проще разобраться с работой конвейера

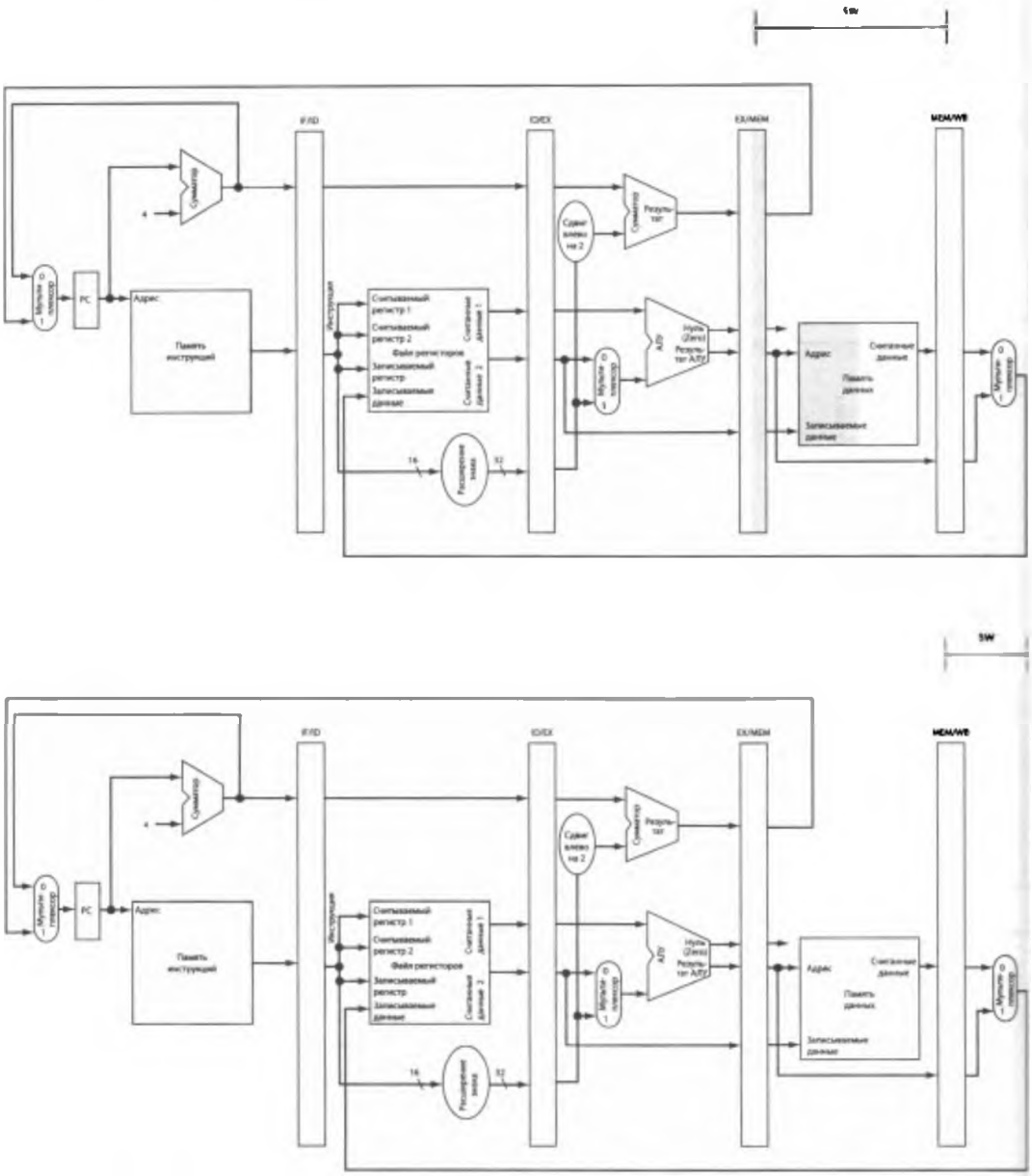


Рис. 4.36. MEM и WB: четвертая и пятая стадии конвейерного выполнения инструкции сохранения. На четвертой стадии данные записываются в память данных для сохранения. Учтите, что данные поступают из регистра конвейера EX/MEM, поэтому в содержимом регистра конвейера MEM/WB ничего не меняется. Как только данные будут записаны в память, инструкции сохранения уже нечего будет делать, поэтому на пятой стадии ничего происходить не будет

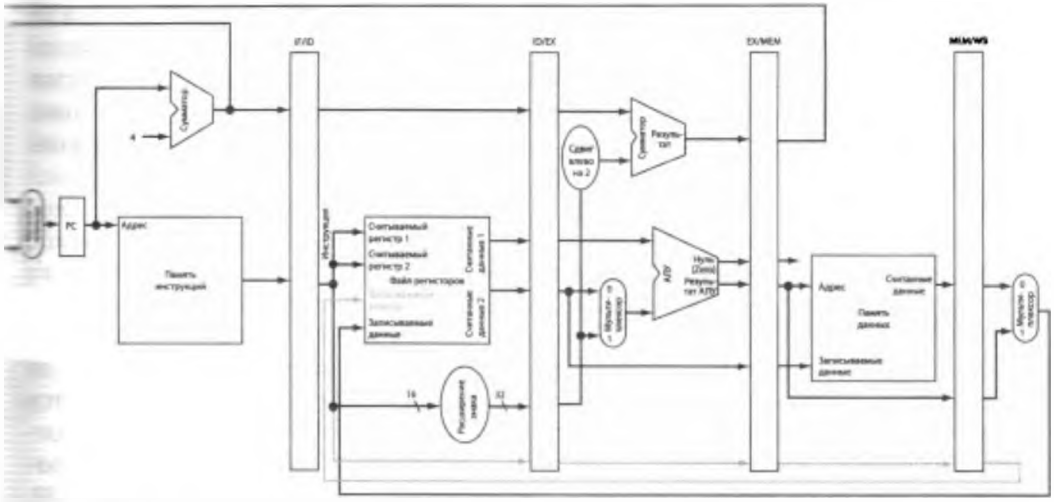


Рис. 4.37. Исправленный конвейеризированный операционный блок для правильной работы с инструкцией загрузки. Теперь номер регистра, в который будет вестись запись, поступает наряду с данными из регистра конвейера MEM/WB. Номер регистра передается из стадии конвейера ID до тех пор, пока он не достигнет регистра конвейера MEM/WB, для этого к последним трем регистрам конвейера добавляются пять дополнительных разрядов. Этот новый путь обозначен другим оттенком

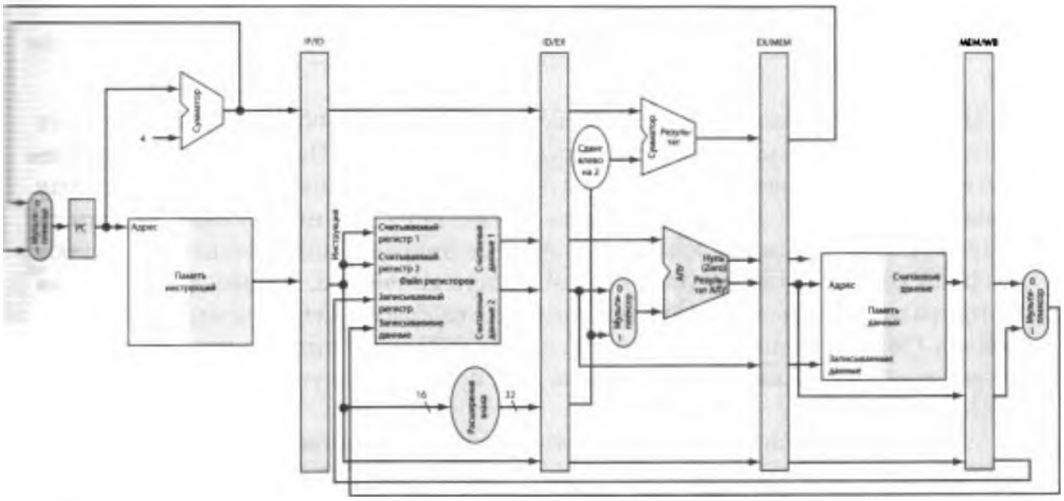


Рис. 4.38. Часть операционного блока, показанного на рис. 4.37, которая используется на всех пяти стадиях выполнения инструкции загрузки

На рис. 4.37 показана исправленная версия операционного блока, передающая номер регистра, в который ведется запись: сначала в регистр ID/EX, затем в регистр EX/MEM и, наконец, в регистр MEM/WB. Номер регистра используется

на стадии WB для указания регистра, в который ведется запись. На рис. 4.38 показано единое изображение исправленного операционного блока, в котором выделено оборудование, используемое на всех пяти стадиях инструкции загрузки слова (load word), показанных на рис. 4.32–4.34. Объяснения относительно того, как заставить работать инструкцию условного перехода ожидаемым образом, даны в разделе 4.8.

Конвейеры, представленные в графическом виде

Конвейеризация может быть довольно трудной для понимания, поскольку за каждый тактовый цикл одним операционным блоком одновременно выполняется множество инструкций. Чтобы во всем этом разобраться, применяются два основных способа демонстрации конвейера: *диаграмма работы конвейера с изображением нескольких тактовых циклов*, например схема, изображенная на рис. 4.30, и *диаграмма работы конвейера с изображением одного тактового цикла*, например схема, изображенная на рис. 4.32–4.36. Диаграммы работы конвейера с изображением нескольких тактовых циклов выглядят проще, но не содержат всех подробностей. Рассмотрим, к примеру, следующую последовательность из пяти инструкций:

```
lw    $10, 20($1)
sub   $11, $2, $3
add   $12, $3, $4
lw    $13, 24($1)
add   $14, $5, $6
```

На рис. 4.39 показана диаграмма работы конвейера с изображением нескольких тактовых циклов, использующаяся для этих инструкций. На этой схеме направление времени показано слева направо, а прогресс выполнения инструкций — сверху вниз. Изображение стадий конвейера привязано к оси инструкций и соответствующим тактовым циклам. Эти условные операционные блоки составляют графическое изображение пяти стадий нашего конвейера. На рис. 4.40 показана более традиционная схема работы конвейера с изображением нескольких тактовых циклов. Обратите внимание на то, что на рис. 4.39 показаны физические ресурсы, используемые на каждой стадии, а на рис. 4.40 используется название каждой стадии.

Схема работы конвейера с изображением одного тактового цикла показывает состояние всего операционного блока в течение одного тактового цикла, и все пять инструкций в конвейере различаются надписями над соответствующими стадиями конвейера. Этот тип изображения используется для того, чтобы показать подробности происходящего внутри конвейера за время каждого тактового цикла. Для этого обычно используются последовательные рисунки, демонстрирующие

работу конвейера за несколько тактовых циклов. Схема работы конвейера с изображением нескольких тактовых циклов используется для представления обзора ситуаций, возникающих на конвейере. Схема работы конвейера с изображением одного тактового цикла представляет вертикальный срез набора схем с изображением нескольких тактовых циклов, показывающий использование операционного блока каждой из инструкций, выполняемой в конвейере в заданном тактовом цикле. Например, на рис. 4.41 показана схема с изображением одного тактового цикла, соответствующая тактовому циклу 5 на рис. 4.39 и 4.40. Безусловно, схемы с изображением одного тактового цикла содержат больше подробностей и занимают намного больше места для показа такого же количества тактовых циклов. В упражнениях вам будет поставлена задача создать такие диаграммы для других кодовых последовательностей.

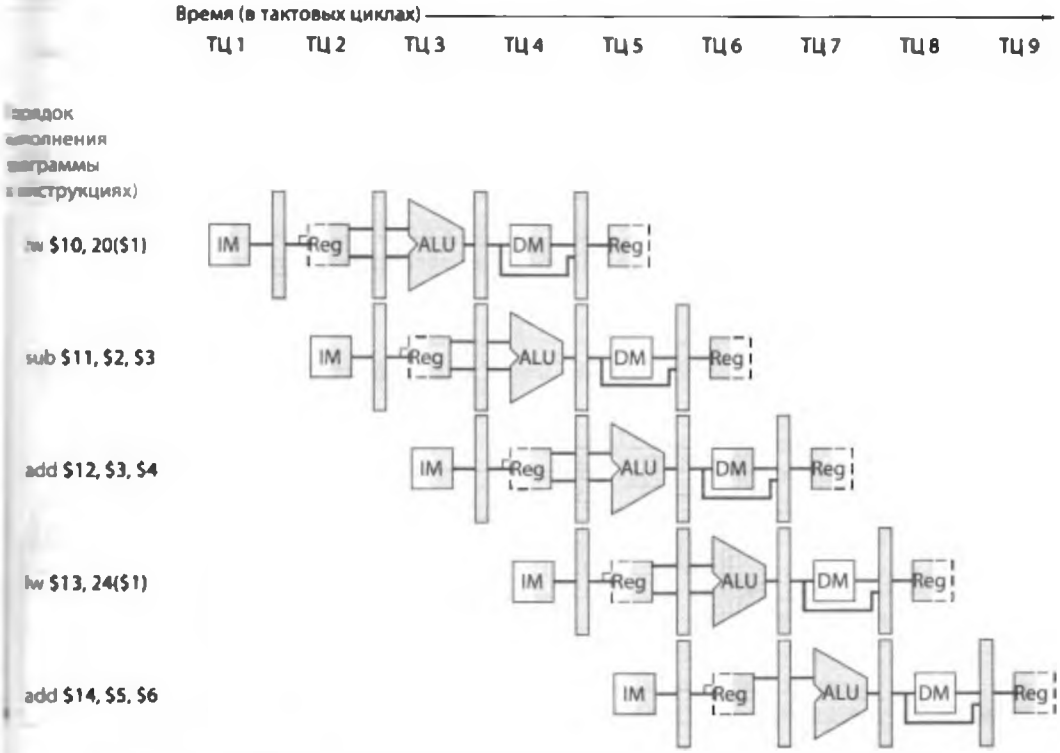


Рис. 4.39. Диаграмма работы конвейера с изображением нескольких тактовых циклов для пяти инструкций. Этот способ представления конвейера показывает весь процесс выполнения инструкций на одном рисунке. Инструкции перечисляются сверху вниз в порядке их выполнения, а тактовые циклы продвигаются слева направо. В отличие от диаграммы, показанной на рис. 4.24, здесь между каждыми стадиями изображены регистры конвейера. На рис. 4.40 показан традиционный способ изображения этой схемы

Рис. 4.40. Традиционная схема работы конвейера с изображением нескольких тактовых циклов для выполнения пяти инструкций, показанных на рис. 4.39

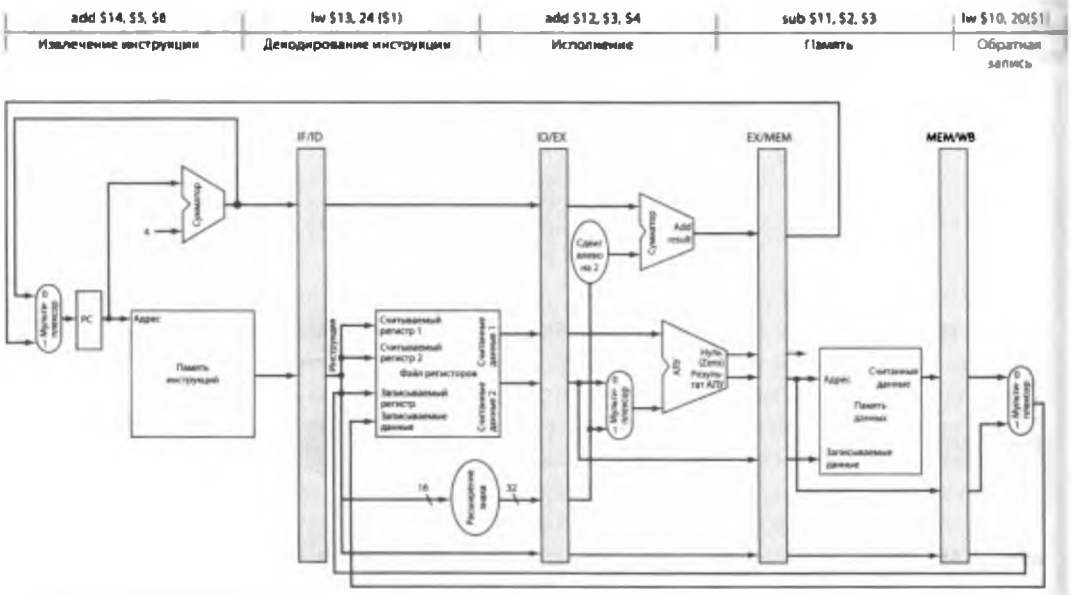


Рис. 4.41. Схема работы конвейера с изображением одного тактового цикла, соответствующая тактовому циклу 5 конвейера, изображенного на рис. 4.39 и 4.40. Как видите, рисунок с изображением одного тактового цикла является вертикальным срезом диаграммы с изображением нескольких тактовых циклов

Самопроверка

Группа студентов обсуждала эффективность работы конвейера из пяти стадий, когда один из студентов заметил, что не все инструкции активны на каждой ста-

ии конвейера. После того как было принято решение проигнорировать влияние конфликтов, студенты пришли к следующим пяти заключениям. Какие из этих заключений являются верными?

1. Если позволить инструкциям безусловных и условных переходов, а также инструкциям, применяющим АЛУ, использовать меньше пяти стадий, требуемых для выполнения инструкции загрузки, то это при любых обстоятельствах повысит производительность конвейера.
2. Попытка позволить некоторым инструкциям использовать меньшее количество стадий не поможет, поскольку пропускная способность определяется продолжительностью тактового цикла; количество стадий конвейера влияет на латентность, а не на пропускную способность.
3. Заставить инструкции, применяющие АЛУ, использовать меньшее количество стадий невозможно, поскольку им необходимо выполнять обратную запись результата, а вот инструкции условного и безусловного перехода могут использовать и меньшее количество стадий, поэтому есть возможности для усовершенствования их конвейерного выполнения.
4. Вместо того чтобы пытаться заставить инструкции сокращать количество стадий, нужно исследовать возможность удлинения конвейера, чтобы инструкции использовали большее количество, но более коротких стадий. Таким образом можно будет повысить производительность.

Конвейеризированная система управления

> В компьютере 6600, возможно, даже больше, чем в любом из предыдущих компьютеров, используется ни на что не похожая система управления.
>
> *Джеймс Торнтон*

Теперь, точно так же как мы в разделе 4.3 добавляли систему управления к однотактному операционному блоку, давайте добавим ее к конвейеризированному операционному блоку. Начнем с простой конструкции. В разделах 4.7–4.9 мы вскроем все конфликты конвейера, возникающие в реальном мире.

Для начала подпишем линии управления на уже имеющемся операционном блоке. Эти линии показаны на рис. 4.42. В данном случае, насколько это было возможно, использовались заимствования из системы управления простым операционным блоком, показанной на рис. 4.14. В частности, использовалась та же логика управления АЛУ, логика условных переходов, мультиплексор номера регистра-получателя и линии управления. Определения этим функциям были даны в табл. 4.1, 4.3 и на рис. 4.15. Ключевая информация, облегчающая отслеживание дальнейшего рассмотрения данного вопроса, повторно приведена в табл. 4.5, 4.6 и на рис. 4.43.

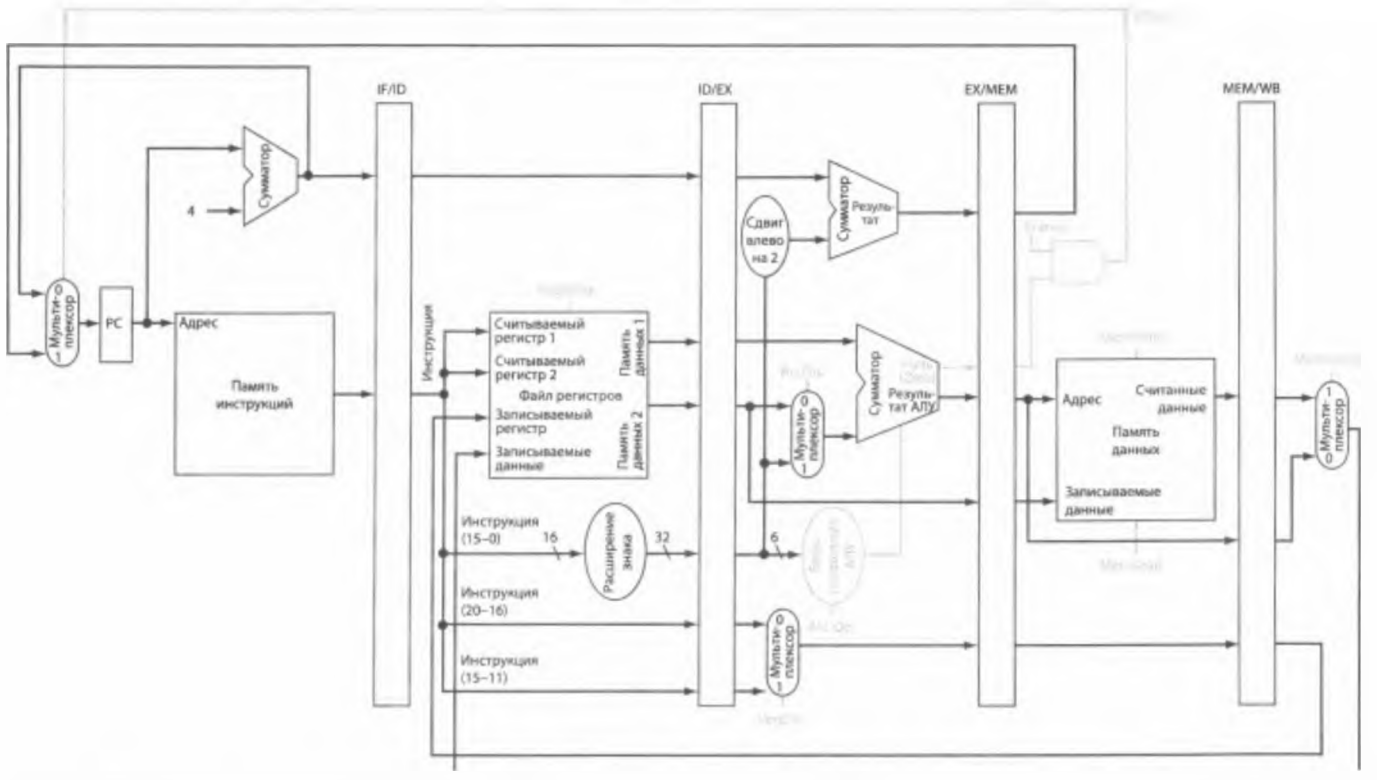


Рис. 4.42. Конвейеризированный операционный блок, изображенный на рис. 4.37, с обозначенными линиями управления. В этом операционном блоке для источника адреса счетчика команд (PC), номера регистра-получателя, АЛУ позаимствована логика управления, рассмотренная в разделе 4.4. Учтите, что теперь нам на стадии EX в качестве управляющего входа АЛУ понадобится 6-разрядное поле funct (код функции), следовательно, эти разряды также должны быть включены в регистр конвейера ID/EX. Следует напомнить, что эти же шесть разрядов являются в инструкциях также шестью самыми младшими разрядами поля непосредственного значения (immediate), поэтому регистр конвейера ID/EX может предоставить их из поля непосредственного значения, поскольку расширение знака не изменит значение этих разрядов

Таблица 4.5. Здесь показано, как устанавливаются разряды управления АЛУ в зависимости от разрядов управления ALUOp и разных кодов функции для инструкции R-типа

Opcode инструкции	ALUOp	Операция инструкции	Код функции	Задаваемое действие АЛУ	Вход управления АЛУ
LW	00	load word (загрузка слова)	XXXXXX	сложение	0010
SW	00	store word (сохранение слова)	XXXXXX	сложение	0010
Branch equal	01	branch equal (условный переход по равенству)	XXXXXX	вычитание	0110
R-тип	10	add (сложение)	100000	сложение	0010
R-тип	10	subtract (вычитание)	100010	вычитание	0110
R-тип	10	AND (И)	100100	И	0000
R-тип	10	OR (ИЛИ)	100101	ИЛИ	0001
R-тип	10	set on less than (установка, если меньше чем)	101010	установка, если меньше чем	0111

Как и для случая однотактной реализации, мы предполагаем, что запись в счетчик команд (PC) ведется при каждом тактовом цикле, поэтому отдельный сигнал записи для PC отсутствует. По той же причине отсутствуют и отдельные сигналы записи для регистров конвейера (IF/ID, ID/EX, EX/MEM, and MEM/WB), поскольку запись в эти регистры ведется также во время каждого тактового цикла.

Чтобы определить сигналы управления для конвейера, нам нужно лишь установить их значения на каждой стадии конвейера. Поскольку каждая линия управления связана с компонентом, активным только на одной стадии конвейера, линии управления можно разделить на пять групп, соответствующих стадиям конвейера.

1. *Извлечение инструкции*: сигналы управления на чтение памяти инструкций и на запись в PC всегда выставлены, поэтому данная стадия конвейера не нуждается ни в каком специфическом управлении.
2. *Декодирование инструкции и чтение из файла регистров*: как и на предыдущей стадии, в каждом тактовом цикле происходят одни и те же события, поэтому установка дополнительных линий управления не требуется.
3. *Исполнение или вычисление адреса*: должны быть выставлены сигналы RegDst, ALUOp и ALUSrc (см. табл. 4.5 и 4.6). Сигналы приводят к выбору регистра-приемника, операции АЛУ и направляемых на второй вход АЛУ данных: либо данных с выхода Считанные данные 2, либо непосредственных значений инструкции, прошедшей расширение знака.
4. *Обращение к памяти*: сигналами, выставленными на данной стадии на линиях управления, являются Branch, MemRead и MemWrite. Эти сигналы выставля-

ются соответственно инструкциями условного перехода по равенству (branch equal), загрузки (load) и сохранения (store). Следует напомнить, что сигнал PCSrc в табл. 4.6 приводит к выбору следующего по порядку адреса, кроме ситуации, когда выставлен сигнал Branch и результат, выданный АЛУ, был нулевым.

5. *Обратная запись*: Выставляется сигнал MemtoReg, позволяющий принять решение, посылать в файл регистров результат, выданный АЛУ, или значение из памяти, и RegWrite, разрешающий запись выбранного значения.

Таблица 4.6. Копия таблицы 4.3. Здесь определена функция каждого из семи сигналов управления. Линии управления ALU (ALUOp) определены во втором столбце таблицы 4.5. Когда выставлен одноразрядный сигнал управления мультиплексором с двумя входами, мультиплексор выбирает вход, соответствующий единице. В противном случае, если управляющий сигнал не выставлен, мультиплексор выбирает нулевой вход. Учтите, что PCSrc на рис. 4.42 управляется логическим элементом И (AND). Если выставлены оба сигнала Branch и ALU Zero (Нуль), сигнал PCSrc имеет значение 1; в противном случае он имеет значение 0. Система управления выставляет сигнал Branch только при выполнении инструкции beq; в противном случае сигнал PCSrc имеет значение 0

Название сигнала	Действие при невыставлении (0)	Действие при выставлении (1)
RegDst	Номер регистра-получателя для записываемого регистра, поступает из поля rt (разряды 20:16)	Номер регистра-получателя для записываемого регистра, поступает из поля rd (разряды 15:11)
RegWrite	Нет	В регистр, указанный на входе записываемого регистра, записывается значение со входа записываемых данных
ALUSrc	Второй операнд АЛУ поступает со второго выхода файла регистров	Второй операнд АЛУ — младшие 16 разрядные инструкции
PCSrc	Значение PC заменяется значением с выхода сумматора, вычисляющего значение PC + 4	Значение PC заменяется значением с выхода сумматора, вычисляющего целевой адрес условного перехода
MemRead	Нет	На выход считанных данных поступает содержимое блока памяти данных, указанное за счет выставления значения на входе Адрес
MemWrite	Нет	Содержимое блока памяти данных, указанное за счет выставления значения на входе Адрес, заменяется значением, указанным на входе записываемых данных
MemtoReg	Значение, поступающее в регистр на вход записываемых данных, берется из АЛУ	Значение, поступающее в регистр на вход записываемых данных, берется из памяти данных

	Исполнение или вычисление адреса				Обращение к памяти			Обратная связь	
Инструкция	RegDst	ALUOp1	ALUOp0	ALUSrc	Branch	Mem-Read	Mem-Write	Reg-Write	Memto-Reg
R-format	1	1	0	0	0	0	0	1	0
lw	0	0	0	1	0	1	0	1	1
sw	X	0	0	1	0	0	1	0	X
beq	X	0	1	0	1	0	0	0	X

Рис. 4.43. Значения сигналов на линиях управления, аналогичные тем, что были показаны на рис. 4.15, но объединенные в три группы, соответствующие последним трем стадиям конвейера

Поскольку конвейеризация операционного блока оставляет значение линий управления без изменений, мы можем использовать те же значения сигналов управления. На рис. 4.43 показаны те же значения, что и в разделе 4.4, но теперь девять линий управления сгруппированы по стадиям конвейера.

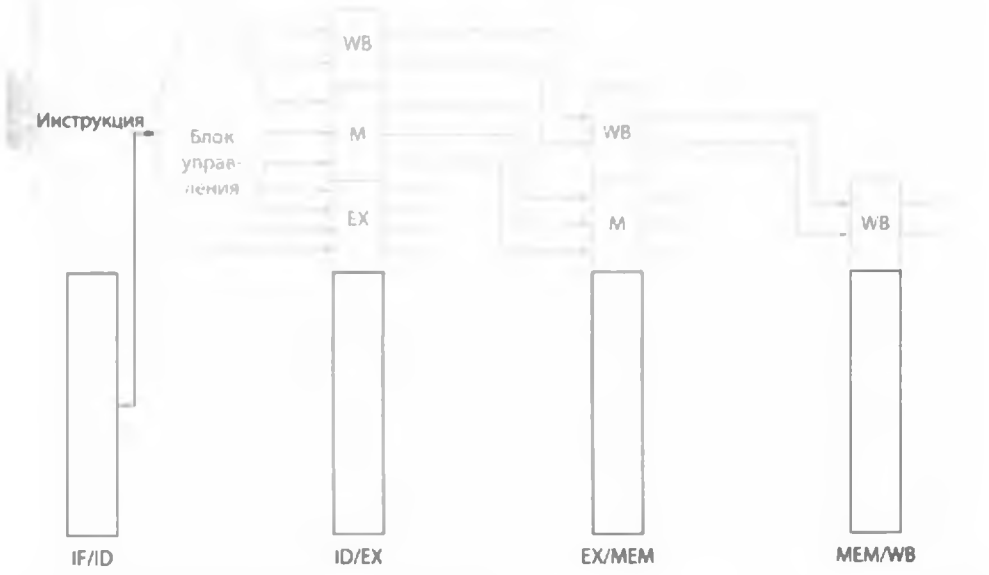


Рис. 4.44. Линии управления для последних трех стадий. Учтите, что четыре из девяти линий управления используются в стадии EX, а сигналы с остальных пяти линий управления передаются в регистр конвейера EX/MEM, расширенный для хранения сигналов линий управления; три линии управления используются на стадии MEM, а сигналы последних двух передаются MEM/WB для использования на стадии WB

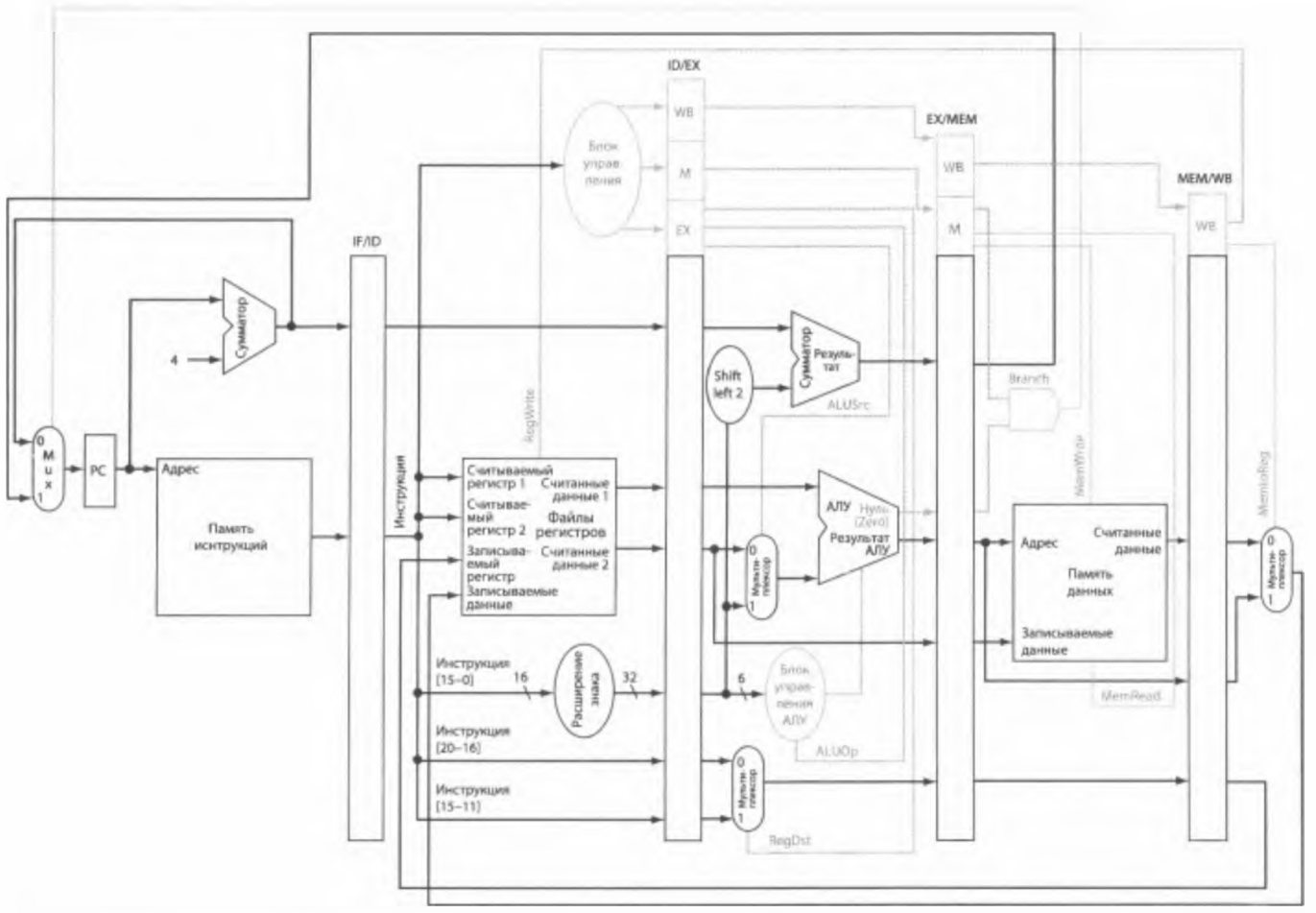


Рис. 4.45. Конвейеризированный операционный блок, показанный на рис. 4.42, с вводами сигналов управления, подключенными к управляющей части регистров конвейера. Значения сигналов управления для последних трех стадий создаются на стадии декодирования инструкции, а затем помещаются в регистр конвейера ID/EX. Используются сигналы линий управления для каждой стадии конвейера, а затем оставшиеся сигналы управления передаются следующей стадии конвейера

Реализация управления означает выставление сигналов на девяти линиях управления в соответствии с этими значениями в каждой стадии для каждой инструкции. Самый простой способ справиться с этой задачей заключается в расширении регистров конвейера, позволяющем включить в них информацию системы управления.

Поскольку линии управления берут свое начало на стадии EX, управляющая информация может быть создана в процессе декодирования инструкции. На рис. 4.44 показано, как эти сигналы управления затем используются на соответствующей стадии работы конвейера по мере продвижения инструкции вниз по конвейеру, наподобие того, как номер регистра-получателя для инструкции загрузки перемещается вниз по конвейеру на рис. 4.37. На рис. 4.45 показан весь операционный блок с расширенными регистрами конвейера и с линиями управления, подведенными к нужным стадиям.

4.7. Конфликты данных: сравнение препровождения данных и задержки

Это же объезд! Объезды нужно строить.
Дуглас Адамс

Примеры, приведенные в предыдущем разделе, продемонстрировали мощность конвейерного выполнения и порядок выполнения задачи компьютерным оборудованием. Теперь настало время снять розовые очки и посмотреть на то, что происходит с настоящими программами. Инструкции, показанные на рис. 4.29–4.31, не зависели друг от друга, ни одна из них не использовала результаты, вычисленные какой-либо из остальных. И все же в разделе 4.5 мы узнали, что конвейерному выполнению препятствуют конфликты данных.

Рассмотрим последовательность, содержащую множество зависимостей, выделенных жирным шрифтом:

```
sub   $2, $1,$3       # В регистр $2 ведется запись инструкцией sub
and   $12,$2,$5       # Первый операнд ($2) зависит от инструкции sub
or    $13,$6,$2       # Второй операнд ($2) зависит от инструкции sub
add   $14,$2,$2       # Первый ($2) и второй ($2) операнды зависят от sub
sw    $15,100($2)     # База ($2) зависит от инструкции sub
```

Все четыре последние инструкции зависят от результата в регистре $2 первой инструкции. Если регистр $2 имел до инструкции вычитания значение 10, а после выполнения этой инструкции значение −20, программист рассчитывает на то, что значение −20 будет использовано в следующей инструкции, ссылающейся на регистр $2.

А как эта последовательность будет выполняться в нашем конвейере? На рис. 4.46 показано выполнение этих инструкций в виде схемы работы конвейера с изображением нескольких тактовых циклов. Для демонстрации выполнения этой последовательности инструкций в нашем текущем конвейере в верхней части

рис. 4.46 показано значение регистра $2 с изменениями, происшедшими в середине пятого тактового цикла, когда инструкция sub записывает свой результат.

Последний потенциальный конфликт может быть разрешен за счет конструкции файла регистров оборудования. Что произойдет, когда регистр считывается и записывается в течение одного и того же тактового цикла? Мы предполагаем, что запись происходит в первой половине тактового цикла, а чтение — во второй его половине, поэтому считывается то, что было записано. В случае многих реализаций файла регистров в подобной ситуации конфликт данных не возникает.

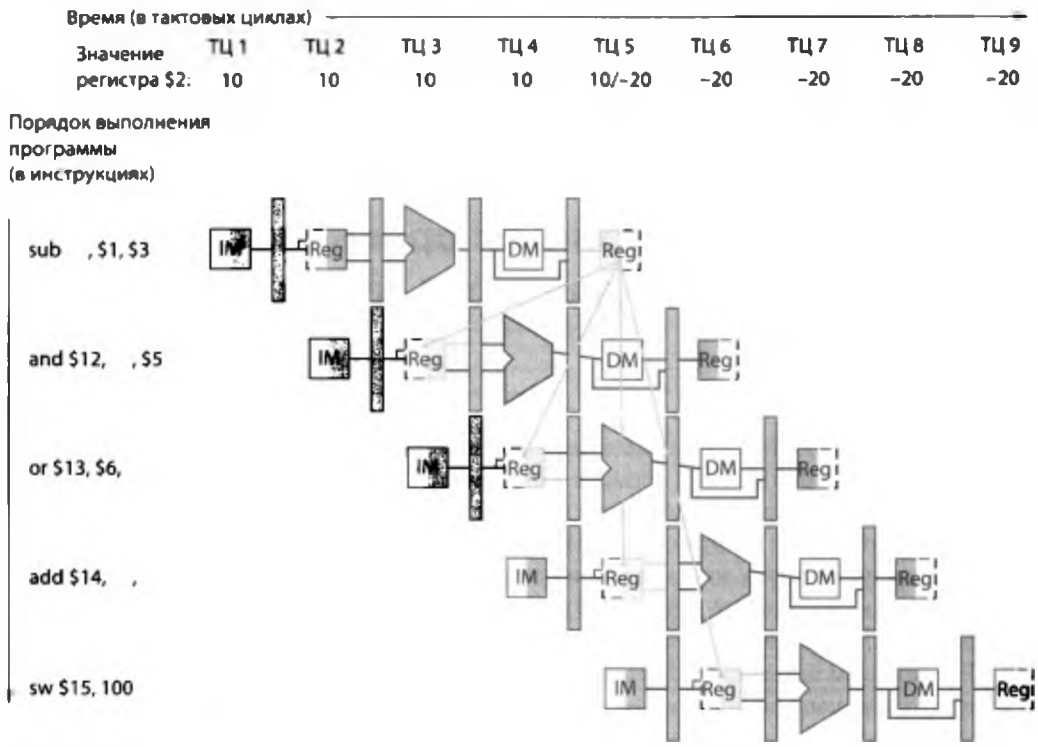


Рис. 4.46. Конвейерные зависимости в последовательности из пяти инструкций при использовании упрощенных операционных блоков для демонстрации этих зависимостей. Все зависимые действия показаны светло-серым цветом, а «ТЦ 1» в верхней части рисунка означает тактовый цикл 1. Первая инструкция ведет запись в регистр $2, а все последующие инструкции считывают значение $2. Запись в этот регистр ведется в тактовом цикле 5, поэтому нужное значение доя наступления тактового цикла 5 недоступно. (Чтение регистра в течение тактового цикла возвращает значение, записанное в конце первой половины цикла, когда и происходит эта запись.) Все зависимости показаны линиями серого цвета, которые направлены от верхнего операционного блока к нижним. Те из них, которые должны идти по времени в обратную сторону, являются конвейерными конфликтами данных

На рис. 4.46 показано, что значения, считанные из регистра $2, не будут результатом выполнения инструкции sub до тех пор, пока чтение не будет вестись во

время пятого и всех последующих циклов. Таким образом, инструкциями, которые получат правильное значение −20, будут add и sw; а инструкции AND и OR получат неправильное значение 10! Использование данного способа графического представления позволяет высветить подобные проблемы, когда линия зависимости может вспять во времени.

Как уже упоминалось в разделе 4.5, нужный результат доступен в конце стадии EX или тактового цикла 3. Когда возникает потребность в данных для инструкций AND и OR? В начале стадии EX, или тактовых циклов 4 и 5 соответственно. Таким образом, мы сможем выполнить этот кодовый фрагмент без задержек, если просто организуем препровождение данных, как только они станут доступны, в любые блоки, которые в них нуждаются, до того, как они станут доступны для чтения из файла регистров.

Как работает препровождение данных? Чтобы не усложнять ситуацию, во всей остальной части раздела мы рассмотрим проблему препровождения данных только в операциях, выполняемых на стадии EX, которые могут быть либо операциями АЛУ, либо операциями вычисления действительного адреса. Если инструкция попытается использовать регистр на EX-стадии, когда предыдущая инструкция (на WB-стадии) использует тот же регистр, нам понадобятся входные значения на АЛУ.

Точнее обозначить зависимости позволяет система записи, использующая названия регистров конвейера. Например, запись «ID/EX.RegisterRs» ссылается на номер одного из регистров, чье значение можно найти в регистре конвейера ID/EX. Первая часть имени слева от точки является именем регистра конвейера, а вторая часть — именем поля этого регистра. При использовании данной системы записи две пары условий конфликта приобретают следующий вид:

```
1a. EX/MEM.RegisterRd = ID/EX.RegisterRs
1b. EX/MEM.RegisterRd = ID/EX.RegisterRt
2a. MEM/WB.RegisterRd = ID/EX.RegisterRs
2b. MEM/WB.RegisterRd = ID/EX.RegisterRt
```

Первый конфликт в последовательности, показанной в начале раздела (регистр $2), возникает между результатом выполнения инструкции sub $2,$1,$3 и первым считываемым операндом инструкции and $12,$2,$5. Этот конфликт может быть обнаружен, когда инструкция and находится на стадии EX, а предыдущая инструкция находится на стадии MEM, то есть это соответствует конфликту 1a:

```
EX/MEM.RegisterRd = ID/EX.RegisterRs = $2
```

Упражнение

Выявление зависимости

Классифицируйте зависимости в последовательности, приведенной в начале раздела:

```
sub   $2, $1,$3          # В регистр $2 ведется запись инструкцией sub
and   $12,$2,$5          # Первый операнд ($2) зависит от инструкции sub
or    $13,$6,$2          # Второй операнд ($2) зависит от инструкции sub
add   $14,$2,$2          # Первый ($2) и второй ($2) операнды зависят от sub
sw    $15,100($2)                # База ($2) зависит от инструкции sub
```

Ответ

Как уже упоминалось, зависимость sub-and относится к конфликту типа 1a. Все остальные конфликты классифицируются следующим образом:

- Зависимость sub-or относится к конфликту типа 2б:

  MEM/WB.RegisterRd = ID/EX.RegisterRt = $2.

- Две зависимости sub-add не являются конфликтами, поскольку на ID-стадии выполнения инструкции add файл регистров предоставляет верные данные.
- Между инструкциями sub и sw конфликт данных не возникает, потому что sw считывает значение регистра $2 в тактовом цикле, который следует после того, как инструкция sub записывает данные в регистр $2.

Поскольку некоторые инструкции не ведут запись в регистры, эта методика не совсем точна; иногда препровождение данных будет производиться, когда в нем нет необходимости. Одно из решений заключается в простой проверке, позволяющей определить, будет ли выставлен сигнал RegWrite: за счет изучения поля управления WB регистра конвейера на стадиях EX и MEM определяется, выставлен или нет сигнал RegWrite. Вспомним, что архитектура MIPS требует, чтобы каждое использование регистра $0 в качестве операнда выдавало значение операнда, равное нулю. В том случае, когда инструкция в конвейере имеет в качестве регистра-получателя $0 (например, sll $0, $1, 2), нам нужно избежать препровождения данных, являющихся, возможно, ненулевым результирующим значением. Отказ от препровождения результатов, предназначенных для регистра $0, освобождает программиста, использующего ассемблер и компилятор, от любых требований избегать использования $0 в качестве получателя данных. Вышеперечисленные условия работают должным образом в том случае, если мы добавим EX/MEM.RegisterRd ≠ 0 к условию первого конфликта и MEM/WB.RegisterRd ≠ 0 к условиям второго конфликта.

Получив возможность обнаружения конфликтов, мы решаем половину проблемы, но по-прежнему остается потребность в препровождении требуемых данных.

На рис. 4.47 показаны зависимости между регистрами конвейера и входами АЛУ для той же кодовой последовательности, показанной на рис. 4.46. Изменение заключается в том, что зависимость начинается с регистра конвейера, а не с ожидания, пока на стадии WB будет произведена запись в файл регистров. Таким образом, присутствуют требуемые данные для более поздних инструкций, а данные, предназначенные для препровождения, хранятся в регистрах конвейера.

Если входные данные для АЛУ можно взять из любого регистра конвейера, а не только из ID/EX, тогда мы можем препроводить требуемые данные. Добавляя к входу АЛУ мультиплексоры и подавая соответствующие сигналы управления, мы можем запустить конвейер на полную скорость при наличии этих зависимостей от данных.

На данный момент мы предположим, что единственными инструкциями, нуждающимися в препровождении данных, являются четыре инструкции R-формата: add, sub, AND и OR. На рис. 4.48 показан крупный план АЛУ и регистров конвейера до

после добавления препровождения данных. В табл. 4.7 показаны значения сигналов на линиях управления для мультиплексоров АЛУ, которые выбирают либо значения из файла регистров, либо одно из препровожденных значений.

Время (в тактовых циклах)	ТЦ 1	ТЦ 2	ТЦ 3	ТЦ 4	ТЦ 5	ТЦ 6	ТЦ 7	ТЦ 8	ТЦ 9
ение регистра $2:	10	10	10	10	10/–20	–20	–20	–20	–20
ачение EX/MEM:	X	X	X	20	X	X	X	X	X
ачение MEM/WB:	X	X	X	X	–20	X	X	X	X

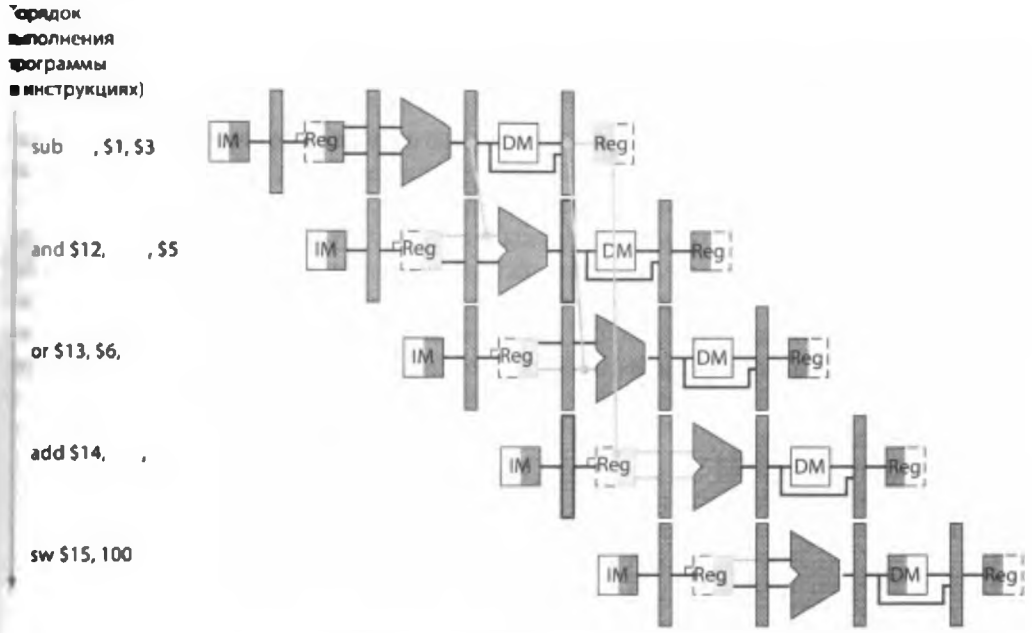


Рис. 4.47. Зависимости между регистрами конвейера перемещаются вперед по шкале времени, поэтому появляется возможность предоставления входных данных АЛУ, необходимых для инструкции AND и инструкции OR, путем препровождения результатов, найденных в регистрах конвейера. Значения в регистрах конвейера показывают, что нужное значение доступно еще до его записи в файл регистров. Мы предполагаем, что файл регистров препровождает значения, которые считываются и записываются во время одного и того же тактового цикла, поэтому выполнение инструкции add происходит без задержки, но значения приходят из файла регистров, а не из регистра конвейера. Файл регистров «препровождает» данные, то есть чтение предоставляет значение записи в данном тактовом цикле, — именно поэтому в тактовом цикле 5 регистр $2 показан как имеющий значение 10 в начале и –20 в конце цикла. Как и во всей остальной части данного раздела, мы рассматриваем все случаи препровождения данных, за исключением того значения, которое должно быть сохранено при выполнении инструкции сохранения

Управление препровождением будет осуществляться на стадии EX, потому что мультиплексоры препровождения данных в АЛУ встречаются именно на этой стадии. Таким образом, мы должны передать номера регистров-операндов из стадии ID через регистр конвейера ID/EX, чтобы определить, препровождать значения

или нет. У нас уже есть поле rt (разряды 20–16). Перед препровождением в регистр ID/EX не нужно было включать пространство для хранения поля rs. Следовательно, поле rs (разряды 25–21) добавлено к ID/EX.

Теперь давайте запишем оба условия для определения конфликтов и выставления управляющих сигналов для их разрешения:

1. EX *конфликт*:

```
если (EX/MEM.RegWrite
и (EX/MEM.RegisterRd ≠ 0)
и (EX/MEM.RegisterRd = ID/EX.RegisterRs)) ForwardA = 10

если (EX/MEM.RegWrite
и (EX/MEM.RegisterRd ≠ 0)
и (EX/MEM.RegisterRd = ID/EX.RegisterRt)) ForwardB = 10
```

Следует заметить, что поле EX/MEM.RegisterRd содержит номер регистра-получателя или для АЛУ-инструкции (который берется из поля Rd инструкции), или для инструкции загрузки (который берется из поля Rt).

В данном случае результат из предыдущей инструкции препровождается на любой вход АЛУ. Если предыдущая инструкция собирается вести запись в файл регистров и номер записываемого регистра соответствует номеру регистра, считываемого во входы A или в АЛУ, при условии, что это не регистр 0, то мультиплексору предписывается забрать имеющееся значение, а не брать его из регистра конвейера EX/MEM.

2. MEM *конфликт*:

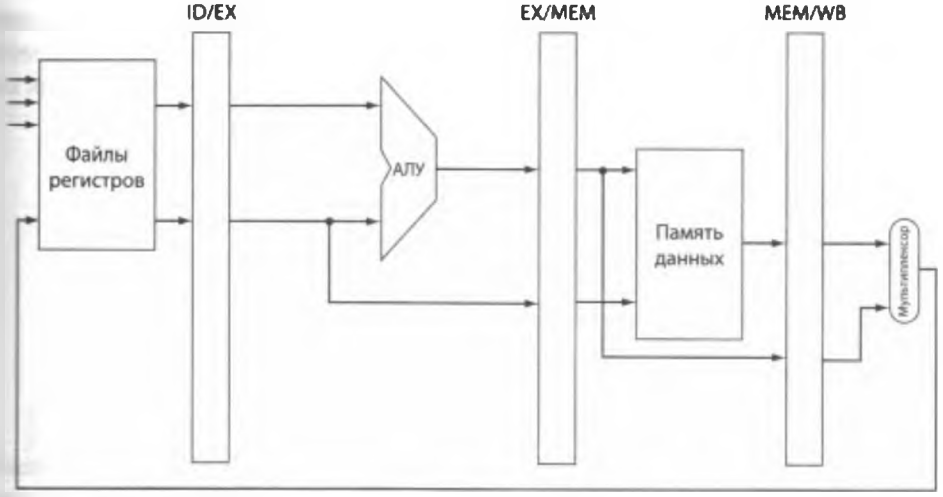
```
если (MEM/WB.RegWrite
и (MEM/WB.RegisterRd ≠ 0)
и (MEM/WB.RegisterRd = ID/EX.RegisterRs)) ForwardA = 01

если (MEM/WB.RegWrite
и (MEM/WB.RegisterRd ≠ 0)
и (MEM/WB.RegisterRd = ID/EX.RegisterRt)) ForwardB = 01
```

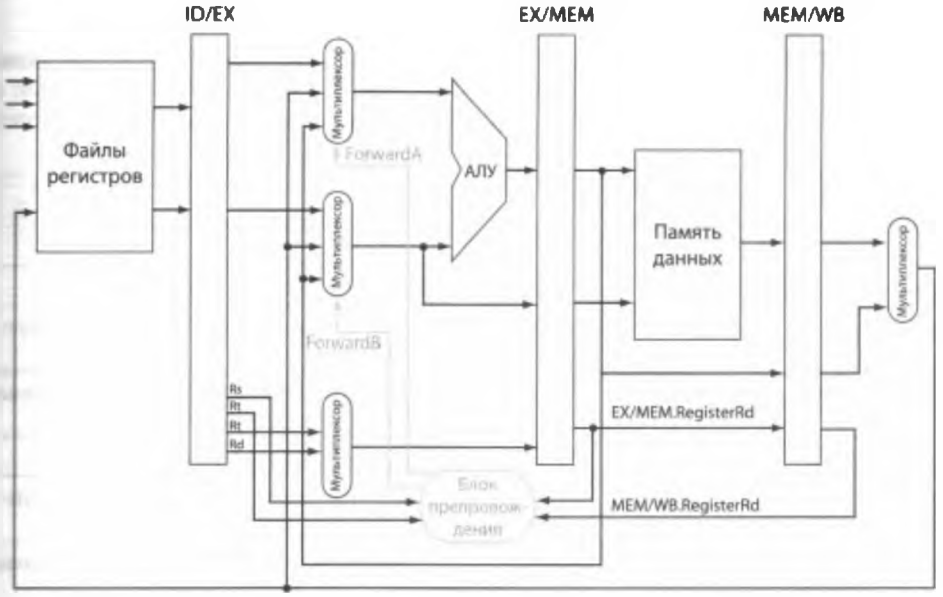
Ранее уже упоминалось, что на стадии WB конфликты не возникают, поскольку мы предполагаем, что файл регистров предоставляет приемлемый результат, если инструкция на стадии ID считывает тот же самый регистр, который записывается инструкцией на стадии WB. Таким образом, файл регистров выполняет еще один вид препровождения данных, но на сей раз все происходит внутри самого файла регистров.

Но возникает одно осложнение в виде потенциального конфликта данных между результатом выполнения инструкции на стадии WB, результатом выполнения инструкции на стадии MEM и операндом-источником инструкции на стадии работы АЛУ. Например, когда в одном и том же регистре суммируется вектор чисел, последовательность инструкций будет заниматься чтением и записью в один и тот же регистр:

```
add $1,$1,$2
add $1,$1,$3
add $1,$1,$4
```



а) без препровождения



б) с препровождением

Рис. 4.48. В верхней части показаны АЛУ и регистры конвейера до добавления препровождения данных. В нижней части мультиплексоры были расширены для добавления всех маршрутов препровождения данных, и мы показали блок препровождения. Все новое оборудование выделено серым цветом. Этот рисунок является условным изображением, поскольку в нем опущены такие подробности полноценного операционного блока, как оборудование для расширения знака. Обратите внимание на то, что поле ID/EX.RegisterRt показано дважды: один раз для подключения к мультиплексору (mux), а второй раз для подключения к блоку препровождения, но на самом деле оно представляет один и тот же сигнал. Как и раньше, здесь проигнорировано препровождение сохраняемого значения для инструкции сохранения. Следует также заметить, что данный механизм работает и для инструкции slt

В этом случае результат препровождается из стадии MEM, поскольку результат в стадии MEM является самым последним результатом. Таким образом, управление для разрешения MEM-конфликта должно иметь следующий вид (дополнения выделены жирным шрифтом):

```
если (MEM/WB.RegWrite
и (MEM/WB.RegisterRd ≠ 0)
и не(EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
        и (EX/MEM.RegisterRd ≠ ID/EX.RegisterRs)
и (MEM/WB.RegisterRd = ID/EX.RegisterRs)) ForwardA = 01

если (MEM/WB.RegWrite
и (MEM/WB.RegisterRd ≠ 0)
и не(EX/MEM.RegWrite и (EX/MEM.RegisterRd ≠ 0)
        и (EX/MEM.RegisterRd ≠ ID/EX.RegisterRt)
и (MEM/WB.RegisterRd = ID/EX.RegisterRt)) ForwardB = 01
```

На рис. 4.49 показано оборудование, необходимое для поддержки препровождения для операций, использующих результаты на стадии EX. Обратите внимание на то, что в поле EX/MEM.RegisterRd сохраняется регистр-получатель либо для АЛУ-инструкции (значение которого берется из поля Rd инструкции), либо для инструкции загрузки (значение которого берется из поля Rt).

Таблица 4.7. Значения сигналов управления для мультиплексоров препровождения, показанных на рис. 4.48. Непосредственное значение со знаком, являющееся еще одним входным значением АЛУ, рассмотрено в «Уточнении»

Управление мультиплексором (mux)	Источник	Объяснение
ForwardA = 00	ID/EX	Первый операнд АЛУ поступает из файла регистров
ForwardA = 10	EX/MEM	Первый операнд АЛУ препровождается из предшествующего результата АЛУ
ForwardA = 01	MEM/WB	Первый операнд АЛУ препровождается из памяти данных или из предшествующего результата АЛУ
ForwardB = 00	ID/EX	Второй операнд АЛУ поступает из файла регистров
ForwardB = 10	EX/MEM	Второй операнд АЛУ препровождается из предшествующего результата АЛУ
ForwardB = 01	MEM/WB	Второй операнд АЛУ препровождается из памяти данных или из предшествующего результата АЛУ

Уточнение. Препровождение может также помочь разрешить конфликты в том случае, когда выполнение инструкций сохранения зависит от других инструкций. Поскольку ими на стадии MEM используется только одно значение данных, препровождение дается относительно легко. Но загрузки, сразу же за которыми следуют сохранения, применяются в MIPS-архитектуре при осуществлении копирования из памяти в память. Поскольку копирование осуществляется довольно часто, нам нужно для ускорения его выполнения добавить дополнительное оборудование препровождения данных. Если бы нам пришлось перерисовывать рис. 4.47, заменяя инструкции sub и AND инструкциями lw и sw, то мы бы увидели, что есть возможность избежать задержки, поскольку

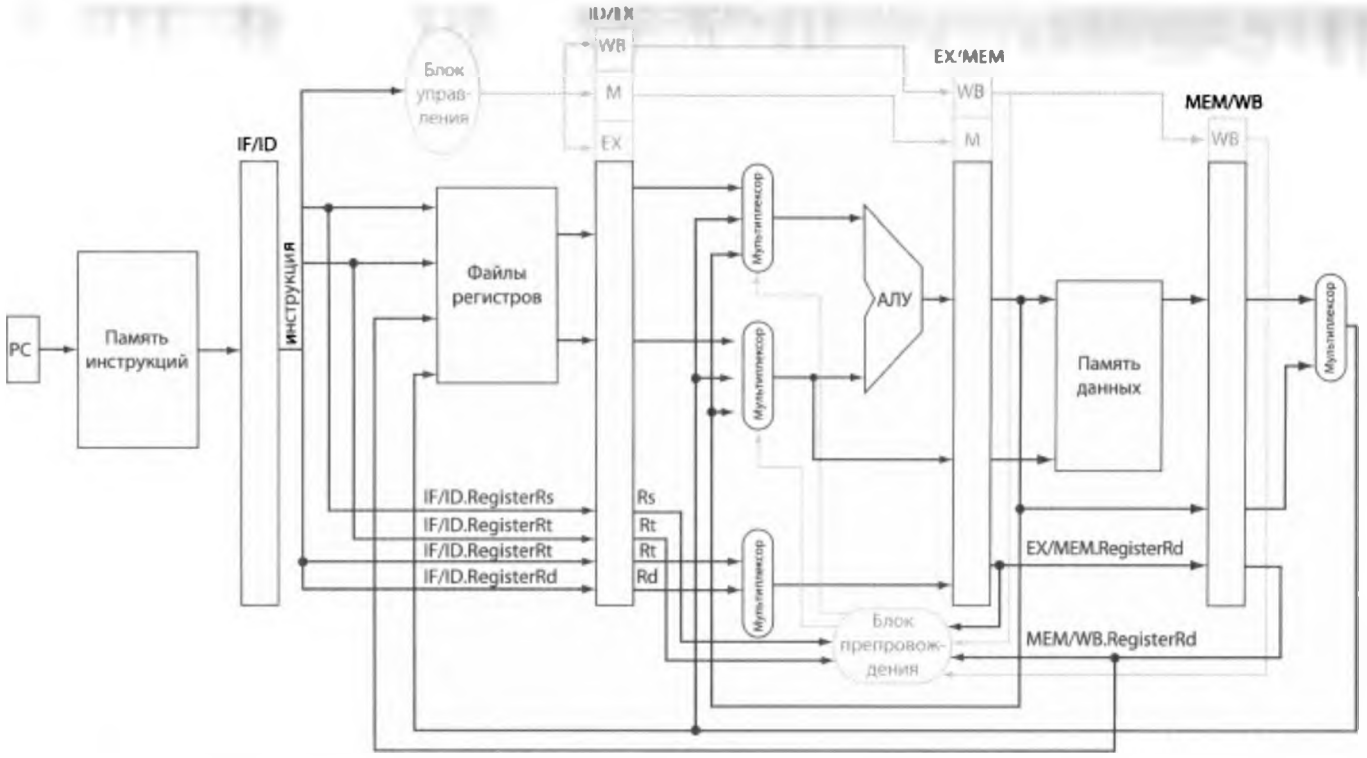


Рис. 4.49. Операционный блок, модифицированный для разрешения конфликтов за счет препровождения данных. По сравнению с операционным блоком, показанным на рис. 4.45, здесь добавлены мультиплексоры на входах АЛУ. Но этот рисунок является еще более условным изображением, поскольку в нем опущены такие подробности полноценного операционного блока, как оборудование для условных переходов и оборудование для расширения знака

данные имеются в регистре MEM/WB инструкции загрузки в то время, как их нужно использовать на стадии MEM инструкции сохранения. Но чтобы их выбрать, нужно на стадии обращения к памяти добавить препровождение данных. Эту модификацию мы оставим в качестве упражнения для читателей.

Помимо всего прочего, операционному блоку, показанному на рис. 4.49, не хватает ввода в АЛУ непосредственного значения со знаком, необходимого для загрузок и сохранений. Поскольку основной системой управления принимается решение, что именно направлять, регистр или непосредственное значение, и поскольку блок препровождения выбирает регистр конвейера для регистрового ввода в АЛУ, наипростейшим решением будет добавить мультиплексор 2:1, выбирающий между выходом мультиплексора, управляемого сигналом ForwardB, и непосредственным значением со знаком. Соответствующие дополнения показаны на рис. 4.50.

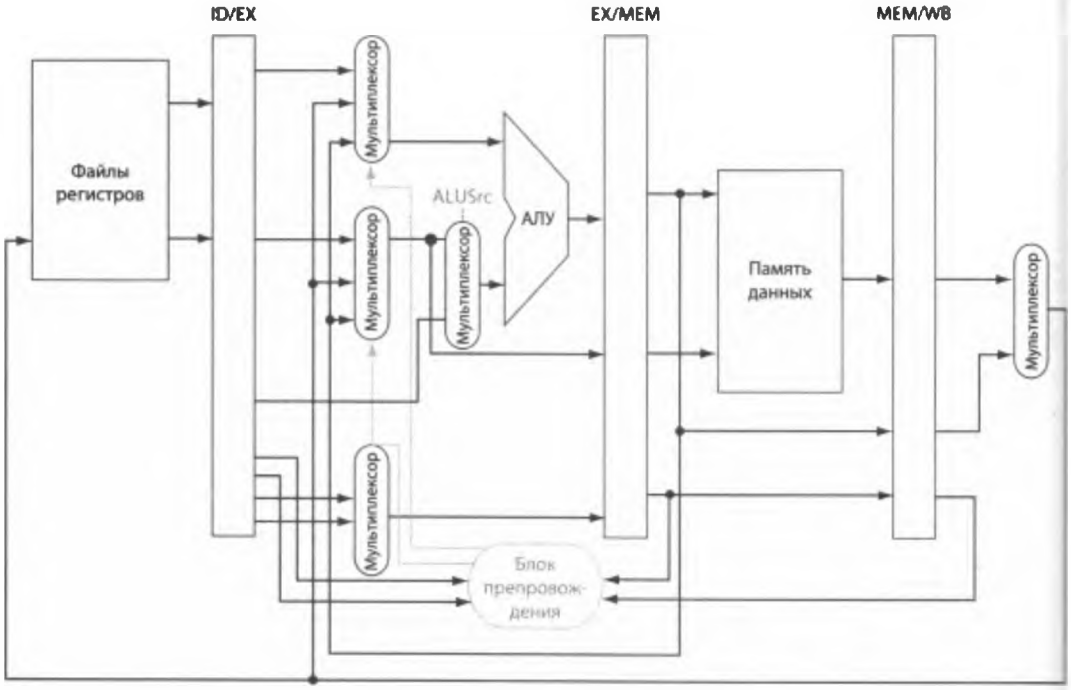


Рис. 4.50. Крупный план операционного блока, изображенного на рис. 4.48, показывающий мультиплексор 2:1, добавленный для выбора непосредственного значения со знаком в качестве входа АЛУ

Конфликты данных и задержки

> Если на первый взгляд ничего не вышло, переопределите критерий успеха.
>
> *Неизвестный автор*

Как говорилось в разделе 4.5, препровождение данных не поможет выправить ситуацию, когда инструкция, которая пытается прочитать содержимое регистра,

следует сразу же за инструкцией загрузки, ведущей запись в этот же самый регистр. Эта проблема показана на рис. 4.51. Данные еще только считываются из памяти во время тактового цикла 4, а АЛУ уже выполняет операцию согласно следующей инструкции. Нужно как-то задержать конвейер для комбинации загрузки, за которой сразу же следует инструкция, считывающая ее результат.

Следовательно, кроме блока препровождения нам нужен блок обнаружения конфликта. Он работает на стадии ID, следовательно, имеет возможность вставить задержку между загрузкой и использованием ее результата. Отслеживая инструкции загрузки, система управления для блока обнаружения конфликта руководствуется следующим единственным условием:

```
если (ID/EX.MemRead и
   ((ID/EX.RegisterRt = IF/ID.RegisterRs) или
     (ID/EX.RegisterRt = IF/ID.RegisterRt)))
      приостановить конвейер
```

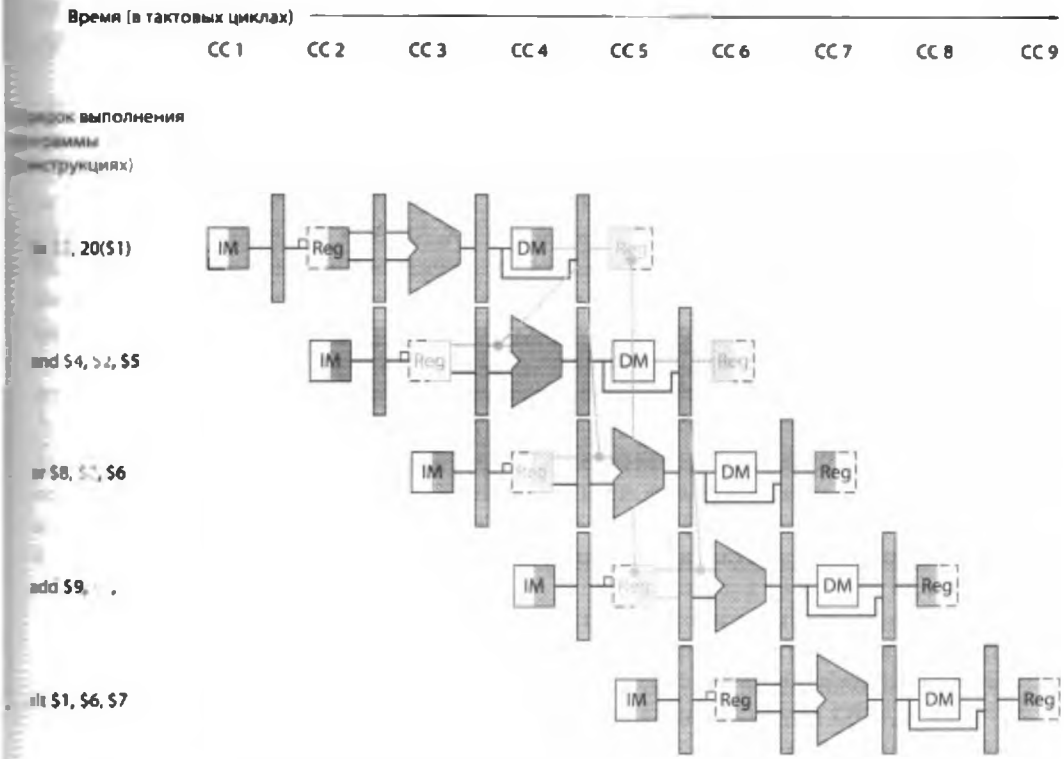


Рис. 4.51. Конвейерное выполнение последовательности инструкций. Поскольку зависимость между загрузкой и следующей инструкцией (and) направлена против хода времени, этот конфликт не может быть разрешен путем препровождения данных. Следовательно, данная комбинация должна вызвать задержку, инициируемую блоком обнаружения конфликта

В первой строке проверяется, является ли инструкцией загрузки данная инструкция. В следующих двух строках проверяется соответствие поля регистра-получателя инструкции загрузки на стадии EX любому регистру-источнику ин-

струкции на стадии ID. Если условие соблюдается, инструкция задерживается на один тактовый цикл. После этой задержки в один цикл логика препровождения сможет справиться с зависимостью, и выполнение продолжится. (Если препровождение не состоится, инструкциям на рис. 4.51 понадобится задержка еще на один тактовый цикл.)

Если инструкция задерживается на стадии ID, то должна быть задержана и инструкция на стадии IF; в противном случае мы потеряем извлеченную инструкцию. Предотвращение выполнения этих двух инструкций достигается простой отменой изменений регистра счетчика (PC) и регистра конвейера IF/ID. Если эти регистры сохранят свои значения, на стадии IF инструкция будет продолжать считывание, используя то же самое значение PC, а на стадии ID будет продолжать считывание регистров с использованием тех же полей в регистре конвейера IF/ID. Возвращаясь к нашей любимой аналогии, это похоже на перезапуск стирки той же самой партии белья и согласие с тем, что сушилка будет работать вхолостую. Разумеется, как и сушилка, вся остальная половина конвейера, начиная со стадии EX, должна чем-то заниматься, и она занимается тем, что выполняет абсолютно ничего не делающие инструкции **nop**.

Но как добавить эти инструкции `nop`, действующие в качестве пузырей, в конвейер? На рис. 4.43 мы видим, что невыставление всех девяти сигналов управления (установка для них значения 0) на стадиях EX, MEM и WB создаст «бездействующую», или `nop`, инструкцию. Обнаружив конфликт на стадии ID, мы можем вставить пузырь, установив поля управления EX, MEM и WB в регистре конвейера ID/EX в нуль. Эти инертные управляющие значения проходят вперед с каждым тактовым циклом с соответствующим эффектом: никакие регистры или ячейки памяти не подвергаются записи, если все значения сигналов управления равны нулю.

На рис. 4.52 показано, что происходит в оборудовании на самом деле: Место в конвейере для выполнения инструкции `AND` превращается в `nop`, и все инструкции, начиная с инструкции `AND`, задерживаются на один цикл. Подобно воздушному пузырю в трубе с водой, пузырь задержки задерживает все, что стоит за ним, и продвигает конвейер инструкций вниз на одну стадию при каждом цикле. В данном примере конфликт заставляет инструкции `AND` и `OR` повторить в тактовом цикле 4 то, что они делали в тактовом цикле 3: инструкция `AND` считывает регистры и проводит декодирование, а инструкция `OR` снова извлекается из памяти инструкций. Задержка выглядит как повторяющаяся работа, но ее действие заключается во времени выполнения инструкций `AND` и `OR` и во времени извлечения инструкции `add`.

На рис. 4.53 выделены все соединительные линии конвейера, как для блока обнаружения конфликта, так и для блока препровождения данных. Как и ранее, блок препровождения управляет мультиплексорами АЛУ для замены значения из регистра общего назначения значением нужного регистра конвейера. Блок обнаружения конфликта управляет записью в регистры PC и IF/ID, а также управляет мультиплексором, осуществляющим выбор между настоящими и нулевыми управляющими сигналами. Блок обнаружения конфликта осуществляет задержку и сброс полей управления,

nop
Инструкция, которая не делает ничего для изменения состояния (no operation).

если тест на наличие загрузки значения с его немедленным использованием будет положительным.

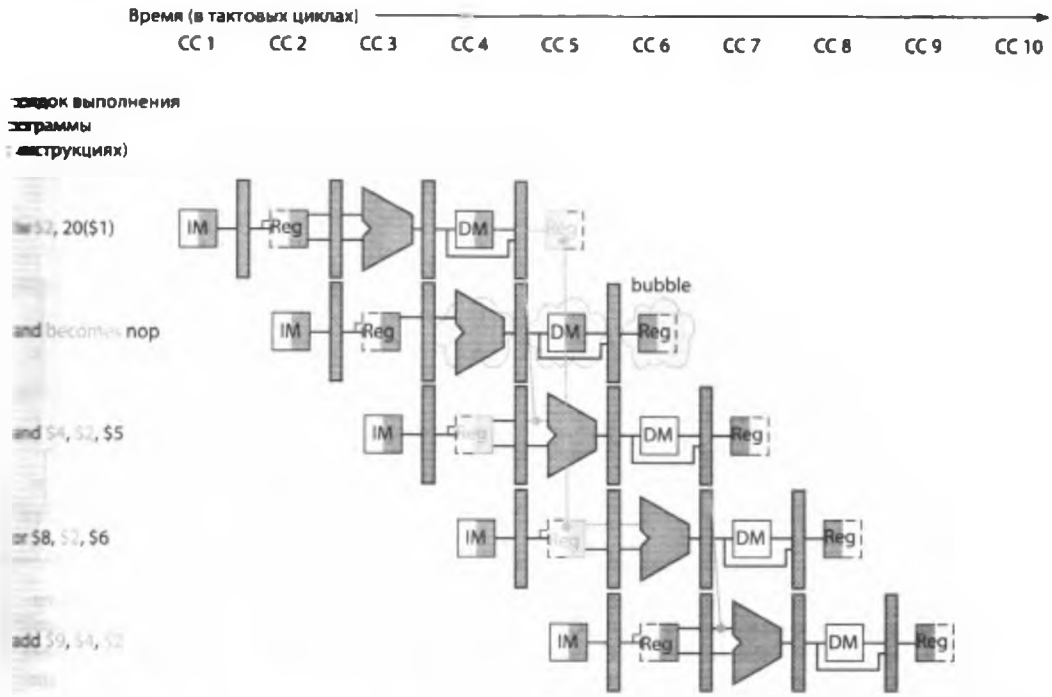


Рис. 4.52. Способ вставки задержки в конвейер. Пузырь добавляется, начиная с тактового цикла 4, путем изменения инструкции and на инструкцию nop. Обратите внимание на то, что инструкция and реально извлекается и декодируется в течение тактовых циклов 2 и 3, но ее стадия EX задерживается до наступления тактового цикла 5 (по сравнению с позицией без задержки, изображенной в тактовом цикле 4). Подобным образом, инструкция OR извлекается в течение тактового цикла 3, но ее стадия ID задерживается до наступления тактового цикла 5 (по сравнению с позицией без задержки в тактовом цикле 4). После добавления пузыря все зависимости смещаются по времени вперед, и в последующем конфликты не возникают

Общее представление

Хотя компилятор для разрешения конфликтов в основном полагается на оборудование, обеспечивая, таким образом, правильное выполнение, для достижения наилучшей производительности он должен понимать тонкости работы конвейера. В противном случае неожиданные задержки будут снижать производительность скомпилированного кода.

Уточнение. Относительно ранее сделанного замечания об обнулении сигналов на линиях управления с целью предотвращения записи в регистры или в ячейки памяти: обнулены должны быть только сигналы RegWrite и MemWrite, а о состоянии других сигналов управления можно не беспокоиться.

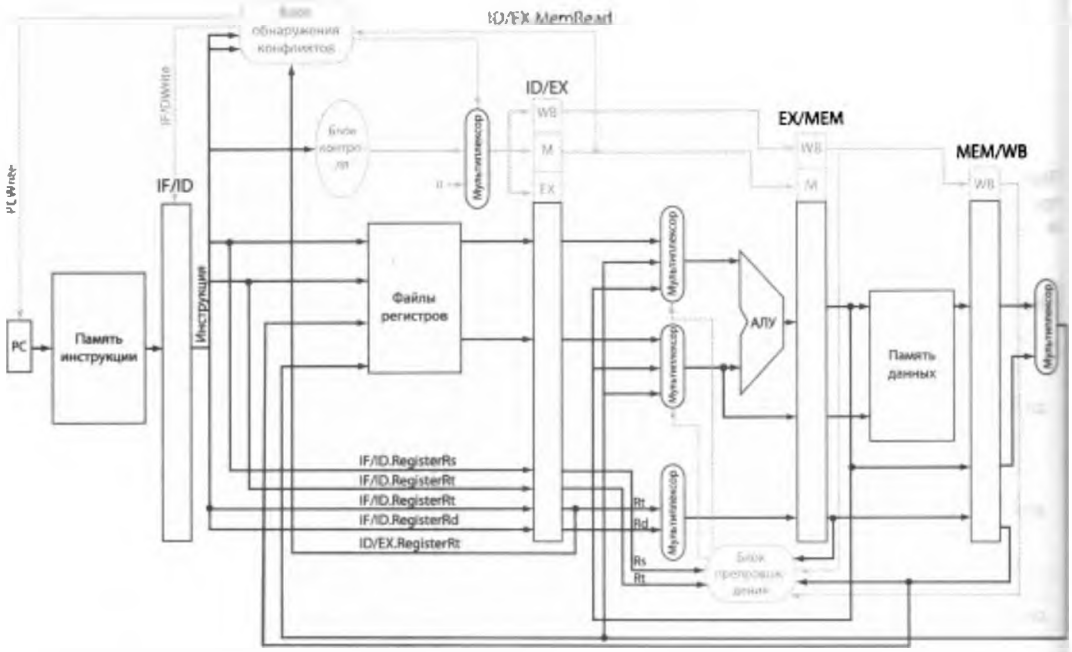


Рис. 4.53. Общий вид конвейеризированного управления, показывающий два мультиплексора для препровождения данных, блок обнаружения конфликтов и блок препровождения данных. Хотя стадии ID и EX были упрощены — здесь не показана логика расширения знака непосредственного значения и условных переходов это изображение отражает суть требований к оборудованию препровождения данных

4.8. Конфликты управления

На один удар по истинным корням зла приходится тысяча охотников обрубать его ветви[1].

Генри Дэвид Торо

До сих пор мы ограничивали наш интерес конфликтами, касающимися арифметических операций и переноса данных. Но, как было показано в разделе 4.5, существуют также конфликты конвейера, касающиеся условных переходов. На рис. 4.54 показана последовательность инструкций и момент выполнения в конвейере условного перехода. Чтобы поддерживался режим работы конвейера, инструкция должна извлекаться с каждым тактовым циклом, в то время как в нашей конструкции решение об условном переходе не принимается до наступления стадии конвейера MEM. Как говорилось в разделе 4.5, эта задержка в определении нужной инструкции для извлечения, в отличие от только что изученного конфликта данных, называется *конфликтом управления*, или *конфликтом условного перехода*.

[1] Перевод З. Е. Александровой.

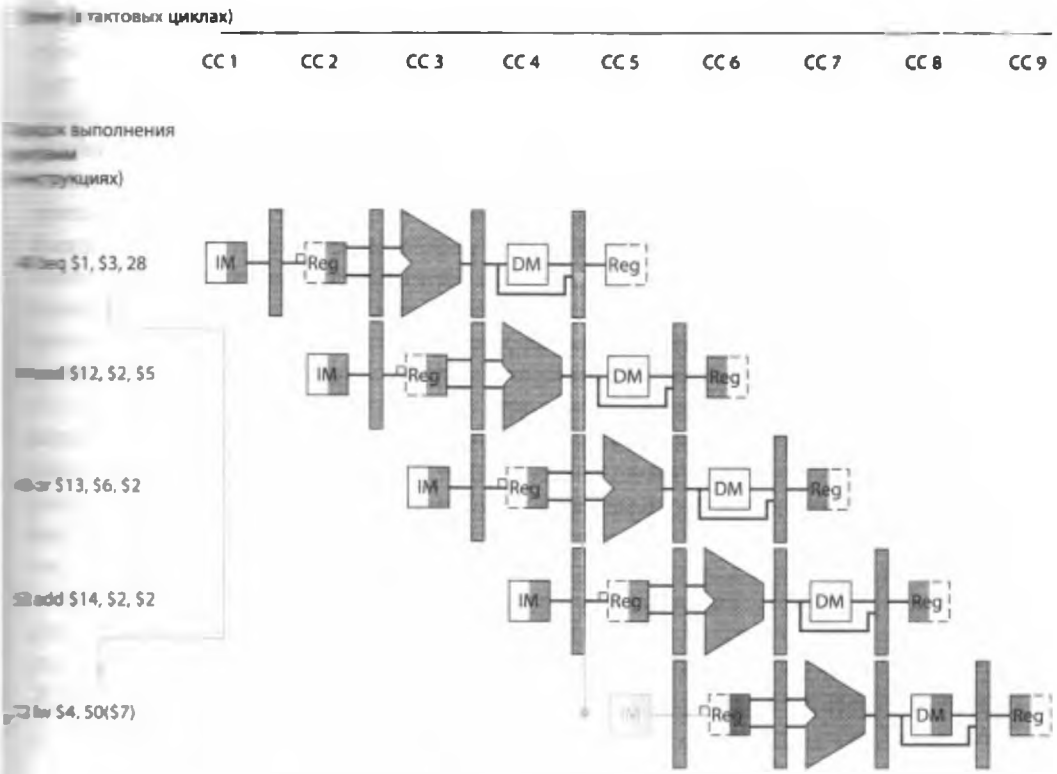


Рис. 4.54. Влияние конвейера на работу инструкции условного перехода. Числа 40, 44, ... — это адреса инструкций. Поскольку инструкция условного перехода решает, нужен ли переход только на стадии MEM — тактовый цикл 4 для показанной выше инструкции beq, — три следующие за условным переходом инструкции будут извлечены и начнут выполняться. Если не вмешаться, эти три следующие инструкции начнут выполняться до условного перехода, инициируемого инструкцией beq на инструкцию lw по адресу 72. (На рис. 4.27 для сокращения конфликта управления до одного тактового цикла предусматривается использование дополнительного оборудования; на данном рисунке используется неоптимизированный операционный блок.)

Этот раздел, посвященный конфликтам управления, короче предыдущих, в которых рассматривались конфликты данных. Причина в том, что конфликты управления относительно проще для понимания, они возникают реже, чем конфликты данных, и для разрешения конфликтов управления не существует столь же эффективных средств как препровождение для разрешения конфликта данных. Следовательно, используются более простые схемы. Будут рассмотрены две схемы разрешения конфликтов управления и оптимизация, позволяющая улучшить эти схемы.

Предположение о том, что условный переход не состоится

Как было показано в разделе 4.5, задержка до тех пор, пока выполнение инструкции условного перехода не завершится, приводит к слишком медленной работе. Простое улучшение по сравнению с задержкой условного перехода заключается в предположении о том, что условный переход не состоится, и, таким образом, в продолжении дальнейшего выполнения потока инструкций. Если условный переход состоится, извлеченные и декодированные к тому времени инструкции должны быть сброшены. Выполнение продолжается с целевого адреса условного перехода. Если условные переходы не предпринимаются в половине случаев и если сброс инструкций обходится относительно дешево, эта оптимизация наполовину снижает издержки конфликтов управления.

Для сброса инструкций мы просто обнуляем исходные значения сигналов управления, почти так же, как это делалось для задержки при конфликте данных, связанном с необходимостью немедленного использования только что загруженных данных. Разница состоит в том, что нам нужно также изменить три инструкции на стадиях IF, ID и EX, когда инструкция условного перехода дойдет до стадии MEM; для задержек, связанных с необходимостью немедленного использования только что загруженных данных, мы всего лишь обнуляли сигналы управления на стадии ID и позволяли им пройти по конвейеру. А **сброс** инструкций означает, что мы должны иметь возможность сбросить их на стадиях конвейера IF, ID и EX.

Сокращение задержки условных переходов

Один из способов улучшения производительности условных переходов состоит в уменьшении издержек состоявшегося условного перехода. До сих пор мы предполагали, что следующее значение счетчика команд (PC) выбирается на стадии MEM, но если мы переместим выполнение условного перехода на более ранние стадии конвейера, то в сбросе будет нуждаться меньшее количество инструкций. Архитектура MIPS была разработана для поддержки быстрых однотактных условных переходов, которые могут быть конвейеризированы с незначительными издержками. Конструкторы подметили, что многие условные переходы зависят только от простых тестов (к примеру, на равенство или на знак) и что такие тесты не требуют полноценного проведения операции АЛУ и могут быть выполнены самое большое с помощью всего нескольких логических элементов. Когда для условного перехода требуется более сложное решение, для выполнения сравнения нужна отдельная инструкция, использующая АЛУ, — эта ситуация похожа на использование для условных переходов кодов условий (см. главу 2).

Для перемещения решения на условный переход вверх нужно, чтобы до этого момента совершились два действия: вычисление целевого адреса перехода и вычисление решения о переходе. Легче всего в этом изменении переместить вверх вычисление целевого адреса перехода. У нас уже есть значение PC и поля непосредственного значения в ре-

Сброс
Сброс инструкций в конвейере, чаще всего из-за неожиданного события.

тре конвейера IF/ID, поэтому мы просто перемещаем сумматор перехода из адии EX в стадию ID; разумеется, вычисление целевого адреса перехода будет ществлено для всех инструкций, но использоваться будет только по необходи-сти. Труднее дается решение о переходе как таковое. Для условного перехода равенству нужно сравнить считанные значения двух регистров на стадии ID, бы определить, равны ли они друг другу. Равенство может быть протестировано ем проведения операции исключающего ИЛИ к их соответствующим разрядам следующим применением операции ИЛИ ко всем результатам.

Перемещение теста условия перехода на стадию ID подразумевает наличие до-лнительного оборудования препровождения данных и обнаружения конфликта, скольку условный переход, зависящий от результата, все еще находящегося конвейере, должен по-прежнему правильно работать после этой оптимизации. Например, для реализации условного перехода по равенству (и его инверсии) нам надобится передать значения логическому устройству, проверяющему равенство тадия ID). Есть два фактора, усложняющих ситуацию:

1. На стадии ID мы должны декодировать инструкцию, решить, нужно ли организовать обход данных для блока проверки равенства, и завершить сравнение на равенство, чтобы при выполнении инструкции условного перехода мы могли бы присвоить PC значение целевого адреса перехода. Препровождение данных для операндов условного перехода прежде управлялось логикой препровождения АЛУ, но введение в ID блока теста на равенство потребует новой логики препровождения. Обратите внимание на то, что препровожденные исходные операнды условного перехода могут поступать либо из регистра-фиксатора конвейера ALU/MEM, либо из регистра-фиксатора конвейера MEM/WB.
2. Поскольку значения, сравниваемые при условном переходе, нужны на стадии ID, но могут быть выданы позже по времени, существует вероятность возникновения конфликта данных и может понадобиться задержка. Например, если АЛУ-инструкция, непосредственно предшествующая условному переходу, производит один из операндов для сравнения в условном переходе, понадобится задержка, поскольку стадия EX для АЛУ-инструкции наступит после стадии ID условного перехода. Далее, если непосредственно за загрузкой следует условный переход по результату загрузки, понадобятся два цикла задержки, поскольку результат загрузки появляется в конце стадии MEM, но потребность в нем возникает в начале стадии ID условного перехода.

Несмотря на эти сложности, перемещение выполнения условного перехода на стадию ID является усовершенствованием, поскольку оно сокращает издержки условного перехода до всего лишь одной инструкции, если условный переход состоится, а именно до той, которая только что была извлечена. В упражнениях исследуются подробности реализации маршрутов препровождения и определения конфликта.

Для сброса инструкций на стадии IF мы добавляем линию управления, которая называется `IF.Flush`, обнуляющую поле инструкции регистра конвейера IF/ID. Очистка регистров превращает извлеченную инструкцию в `nop`, — инструкцию, которая не производит никаких действий и не изменяет никаких состояний.

Упражнение

Конвейеризированный условный переход

Покажите, что произойдет, когда условный переход предпринимается в представленной последовательности инструкций, при условии, что конвейер оптимизирован для несостоявшихся условных переходов и что мы переместили выполнение условного перехода на стадию ID:

```
36 sub $10, $4, $8
40 beq  $1, $3, 7      # переход относительно PC на 40 + 4 + 7 * 4 = 72
44 and $12, $2, $5
48 or  $13, $2, $6
52 add $14, $4, $2
56 slt $15, $6, $7
   ...
72 lw  $4,  50($7)
```

Ответ

На рис. 4.55 показано, что произойдет, когда условный переход состоится. В отличие от показанного на рис. 4.54, здесь всего лишь один конвейерный пузырь при состоявшемся условном переходе.

Динамическое прогнозирование условного перехода

Предположение о том, что переход не состоится, является одной из простейших форм прогнозирования условного перехода. В этом случае мы прогнозируем, что перехода не будет, сбрасывая конвейер, когда прогноз оказывается ошибочным. Для простого конвейера из пяти стадий такой подход, возможно, в совокупности с прогнозом компилятора, наверное, вполне адекватен. Для более глубоких конвейеров издержки условных переходов, измеренные в тактовых циклах, увеличиваются. Аналогично этому, при запуске сразу нескольких инструкций (см. раздел 4.10) растут издержки перехода, измеряющиеся в потерянных инструкциях. Это сочетание означает, что в интенсивном конвейере простая статическая схема прогнозирования будет, наверное, слишком сильно и неоправданно снижать производительность. Как уже говорилось в разделе 4.5, при наличии более совершенного оборудования можно попытаться спрогнозировать исход уловного перехода в процессе выполнения программы.

Один из подходов состоит в поиске адреса инструкции с целью определения, был ли осуществлен условный переход, когда инструкция выполнялась в последний раз, и если был, приступать к извлечению новой инструкции из того же самого места, что и в последний раз. Такая технология называется **динамическим прогнозированием условного перехода**.

Динамическое прогнозирование условного перехода
Прогнозирование перехода в процессе выполнения программы с использованием информации, добытой в этом же процессе

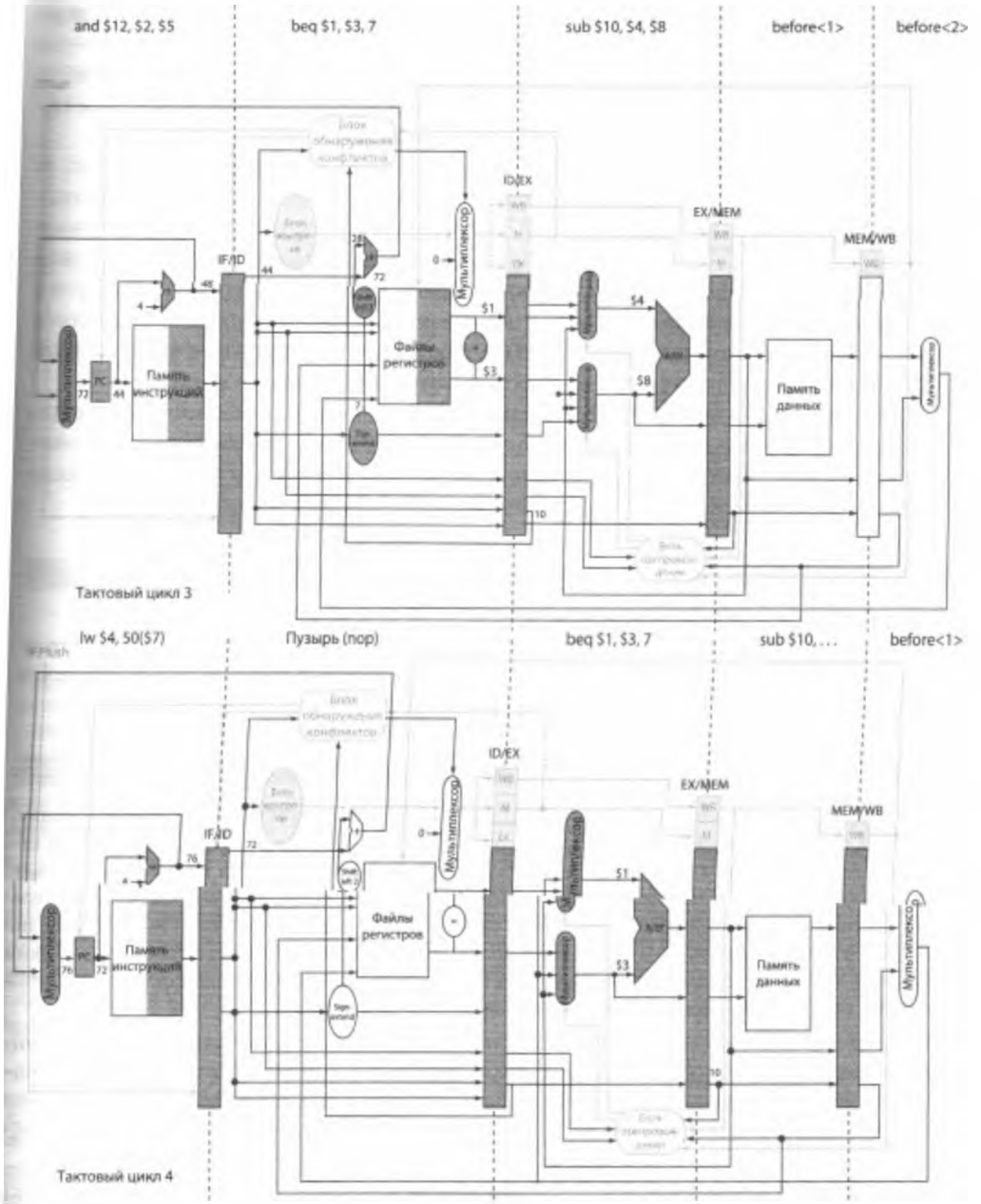


Рис. 4.55. На стадии ID тактового цикла 3 определяется, что условный переход должен состояться, поэтому в качестве следующего адреса PC выбирается значение 72 и обнуляется инструкция, извлеченная для следующего тактового цикла. На тактовом цикле 4 в качестве результата состоявшегося условного перехода показана извлеченная инструкция по адресу 72 и единственный пузырь, или инструкция nop, в конвейере. (Поскольку на самом деле инструкция nop является инструкцией sll $0, $0, 0, вопрос о том, должна быть выделена стадия ID в тактовом цикле 4 или нет, остается спорным.)

Одной из реализаций этого подхода является **буфер прогнозирования перехода, или таблица истории переходов**. Буфер прогнозирования перехода — это небольшой блок памяти, проиндексированный с использованием младшей части адреса инструкции условного перехода. Память содержит разряд, свидетельствующий о том, был недавно предпринят переход или нет.

Это простейший вид буфера; на самом деле мы не знаем, верен ли прогноз, он может быть помещен туда другим условным переходом, имеющим такие же младшие разряды адреса. Но это не влияет на точность. Прогнозирование — это всего лишь подсказка, на правильность которой питаются надежды и по которой извлечение инструкции начинается в спрогнозированном направлении. Если окажется, что подсказка неверна, неправильно спрогнозированные инструкции удаляются, разряд прогнозирования инвертируется и сохраняется на прежнем месте, а затем извлекается и выполняется нужная последовательность инструкций.

Простая одноразрядная схема прогнозирования не дает заметного улучшения производительности: даже если переход происходит почти всегда, мы можем ошибиться в прогнозировании дважды, а не один раз, как в случае отсутствия перехода. Подобное затруднение демонстрируется в следующем упражнении.

Упражнение

Циклы и прогнозирование

Рассмотрим условный переход в цикле, который состоялся девять раз подряд, после чего не состоялся один раз. Какова точность прогнозирования для этого перехода, при условии, что разряд прогнозирования для него находится в буфере прогнозирования?

Ответ

Установившееся состояние прогнозирования приведет к неправильному прогнозированию первой и последней итерации цикла. Неверный прогноз последней итерации является неизбежным. Разряд прогнозирования будет показывать состоявшийся переход, поскольку переход на тот момент осуществлялся девять раз подряд. Неверный прогноз первой итерации дается из-за того, что разряд поменял свое значение при предыдущем выполнении последней итерации цикла, ведь при итерации с выходом из цикла переход не состоялся. Таким образом, точность прогнозирования того, что этот переход состоится, составляет в 90% случаев только 80% (два неверных прогноза при восьми верных).

В идеале для таких весьма регулярных условных переходов точность предсказателя соответствовала бы частоте этих переходов. Чтобы избавиться от этого недостатка, часто используются схемы двухразрядных предсказателей. В двухразрядной схеме предсказатель должен ошибиться дважды, прежде чем его значение изменится. На рис. 4.56 показан конечный автомат для двухразрядной схемы предсказателя.

Буфер прогнозирования перехода
Также называется таблицей истории переходов. Небольшой блок памяти, проиндексированный с использованием младшей части адреса инструкции условного перехода и содержащий один или несколько разрядов, свидетельствующих о том, состоялся переход в последний раз или нет.

Буфер предсказателя условного перехода может быть реализован в виде небольшого специального буфера, доступного вместе с адресом инструкции на стадии конвейера IF. Если действие инструкции предсказывается состоявшимся, извлечение начинается с целевого адреса, как только он будет известен счетчику команд; как уже ранее упоминалось, может произойти уже на стадии ID. В

произойдет извлечение следующей по порядку инструкции и ее выполнение. Если предсказание окажется неверным, разряды предсказателя изменят свое значение, как показано на рис. 4.56.

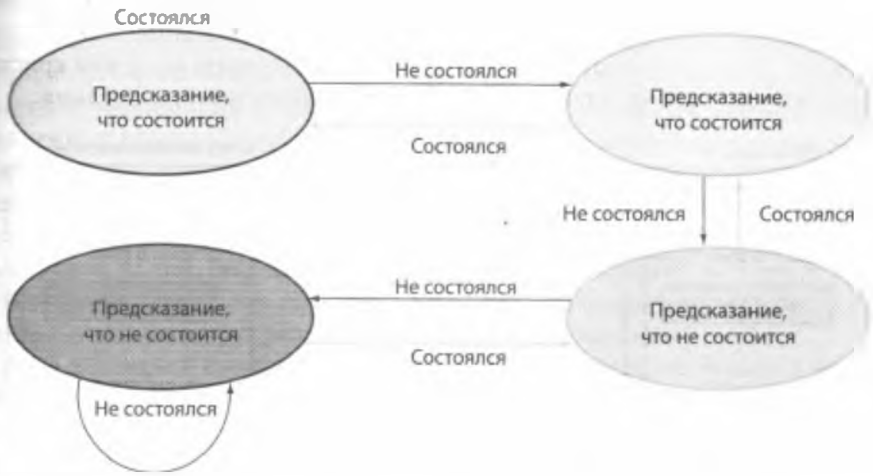


Рис. 4.56. Состояния в двухразрядной схеме прогнозирования. За счет использования двух разрядов вместо одного переход, который чаще всего является состоявшимся или несостоявшимся, — что и происходит со многими переходами, — будет неправильно спрогнозирован только один раз. Двa разряда используются для кодирования четырех состояний системы. Двухразрядная схема служит общим примером блока прогнозирования, основанного на работе счетчика, получающего приращение, когда прогноз точен, и получающего отрицательное приращение, когда он неточен, и использующего среднее состояние своего диапазона чисел в качестве расхождения между прогнозированием состоявшегося и несостоявшегося перехода

Уточнение. Как говорилось в разделе 4.5, в конвейере, состоящем из пяти стадий, переопределяя условный переход, можно вызвать конфликт управления. Отложенный переход всегда приводит к выполнению следующей инструкции, но вторая инструкция, следующая за инструкцией условного перехода, будет подвержена влиянию этого перехода.

Компиляторы и ассемблеры пытаются поместить всегда выполняемые после перехода инструкции в **слот задержки перехода**. Задача программного обеспечения состоит в том, чтобы сделать последующие инструкции допустимыми и полезными. На рис. 4.57 показаны три способа диспетчеризации слота задержки перехода.

Ограничения, накладываемые диспетчеризацией отложенного перехода, возникают из-за: 1) ограничений, накладываемых на инструкции, вносимые в слоты задержки, и 2) нашей возможности прогнозирования в процессе компиляции, будет ли, скорее всего, предпринят переход или нет.

Слот задержки перехода
Слот, следующий сразу же за отложенной инструкцией условного перехода, который в MIPS архитектуре заполняется инструкциями, не влияющими на переход.

Отложенный условный переход был простым и эффективным решением для конвейера, состоящего из пяти стадий и выполняющего по одной инструкции за каждый тактовый цикл. По мере того как процессоры переходят как на более длинные конвейеры, так и на выполнение сразу нескольких инструкций за один тактовый цикл (см. раздел 4.10), задержка условного перехода становится продолжительнее, а единственный слот задержки становится недостаточным. Следовательно, отложенный условный переход теряет популярность по сравнению с более дорогостоящими, но и более гибкими динамическими подходами.

Наряду с этим рост количества доступных транзисторов на одном кристалле привел к относительному удешевлению динамического прогнозирования.

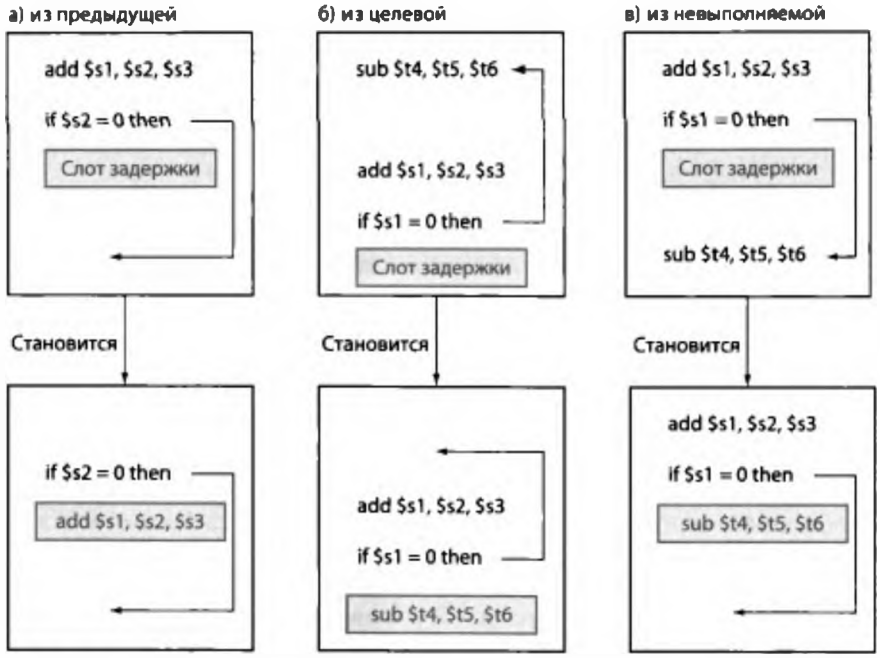


Рис. 4.57. Составление слота задержки перехода. Верхние блоки в каждой паре показывают код до составления, а нижние — после составления. В примере «а» слот задержки составлен из независимой инструкции, которая находилась до инструкции перехода. Это наилучший выход. Подходы «б» и «в» используются в том случае, когда подход «а» невозможен. В кодовой последовательности для примеров «б» и «в» использование $s1 в условии перехода не дает инструкции add (чьим регистром-получателем является $s1) переместиться в слот задержки перехода. В кодовой последовательности для примера «б» слот задержки перехода составляется из инструкции, которая служит целью перехода; обычно целевая инструкция будет нуждаться в копировании, потому что по-другому добраться до нее невозможно. Подход «б» является более предпочтительным, когда условный переход может состояться с высокой долей вероятности: например, если это переход при выполнении цикла. И наконец, слот задержки перехода может быть составлен из инструкции, не выполняемой при выходе из цикла, как в примере «в». Чтобы данная оптимизация было допустима для примера «б» или «в», должно быть вполне приемлемо выполнение инструкции sub, когда переход осуществляется в неожиданном направлении. Под приемлемостью мы понимаем, что работа будет проделана впустую, но на правильность выполнения программы это не повлияет. Так может получиться, к примеру, если на момент осуществления перехода в неожиданном направлении $t4 был неиспользованным временным регистром

Уточнение. Блок прогнозирования перехода сообщает о том, состоится этот переход или нет, но он все еще требует вычисления целевого адреса перехода. В конвейере из пяти стадий это вычисление занимает один цикл, а это значит, что состоявшиеся переходы будут иметь издержку в один цикл. Одним из подходов, избавляющим от этой задержки, является отложенный переход. Еще одним подходом является использование кэш-памяти для хранения целевого адреса для счетчика команд или целевой инструкции, то есть использования **буфера адресов перехода**.

Схема двухразрядного динамического прогнозирования использует только информацию о конкретном переходе. Исследователи заметили, что использование совместной информации о локальном переходе и о глобальном поведении недавно выполненных переходов приводит к более точным прогнозам при таком же количестве разрядов прогнозирования. Такие предсказатели называются **сопоставляющими блоками прогнозирования**. Типичный блок может иметь два двухразрядных предсказателя для каждого условного перехода, а выбор предсказателя может быть основан на том, был или не был предпринят последний условный переход. Таким образом, глобальное поведение может рассматриваться как добавление дополнительных индексных разрядов для поиска предсказания.

Самой последней новинкой в прогнозировании переходов является использование **соревновательных блоков прогнозирования**. В них используется несколько предсказателей, и для каждого перехода отслеживается, какой из предсказателей выдал наилучшие результаты. Типичный соревновательный блок прогнозирования может содержать два предсказателя для каждого индекса условного перехода: один на основе локальной информации и один на основе глобального поведения условного перехода. Селектор выберет, какой предсказатель применить для каждого отдельно взятого блока соревновательного прогнозирования. Аналогичным образом селектор может работать с одно- или двухразрядными предсказателями, выбирая тот из предсказателей, который был более точен. Подобные довольно сложные блоки прогнозирования используются в некоторых самых последних микропроцессорах.

Буфер адресов перехода
Структура, которая кэширует целевой адрес перехода для счетчика команд или целевую инструкцию перехода. Обычно буфер организован в виде кэша с тегами, что делает его более дорогостоящим по сравнению с простым буфером прогнозирования.

Сопоставляющий блок прогнозирования
Предсказатель перехода, комбинирующий локальное поведение конкретного перехода и глобальную информацию о поведении нескольких последних выполненных инструкций условных переходов.

Соревновательный блок прогнозирования
Блок прогнозирования перехода с несколькими предсказателями для каждого условного перехода и механизмом выбора предсказателя, допустимого для заданного перехода.

Уточнение. Один из способов сокращения количества условных переходов заключается в добавлении *условных инструкций перемещения*. Вместо изменения значения счетчика команд (PC) в результате условного перехода инструкция, сообразуясь с условием, изменяет значение регистра-получателя перемещаемых данных. Если условие не соблюдается, перемещение действует как пустая инструкция (nop). Например, в одной из версий архитектуры набора инструкций MIPS имеются две новые инструкции с именами movn (перемещение, если не равно нулю) и movz (перемещение, если равно нулю). Соответственно

инструкция `movn $8, $11, $4` копирует содержимое регистра 11 в регистр 8, при условии, что значение в регистре 4 не равно нулю; в противном случае эта инструкция бездействует. Набор инструкций ARM имеет в большинстве инструкций поле условия.

Следовательно, ARM-программы могут содержать меньше условных переходов, чем MIPS-программы.

Краткие выводы по конвейеризации

Рассмотрение этого вопроса началось в прачечной, где были показаны принципы конвейеризации в обычной жизни. Используя эту аналогию в качестве учебного пособия, мы шаг за шагом объясняли конвейерное выполнение инструкций, начиная с однотактного операционного блока, затем добавляя регистры конвейера, пути препровождения данных, определение конфликта данных, прогнозирование условных переходов и сброс инструкций при исключительных ситуациях. На рис. 4.58 показывается окончательный вид операционного блока и блока управления. Теперь мы готовы к рассмотрению еще одного конфликта управления.

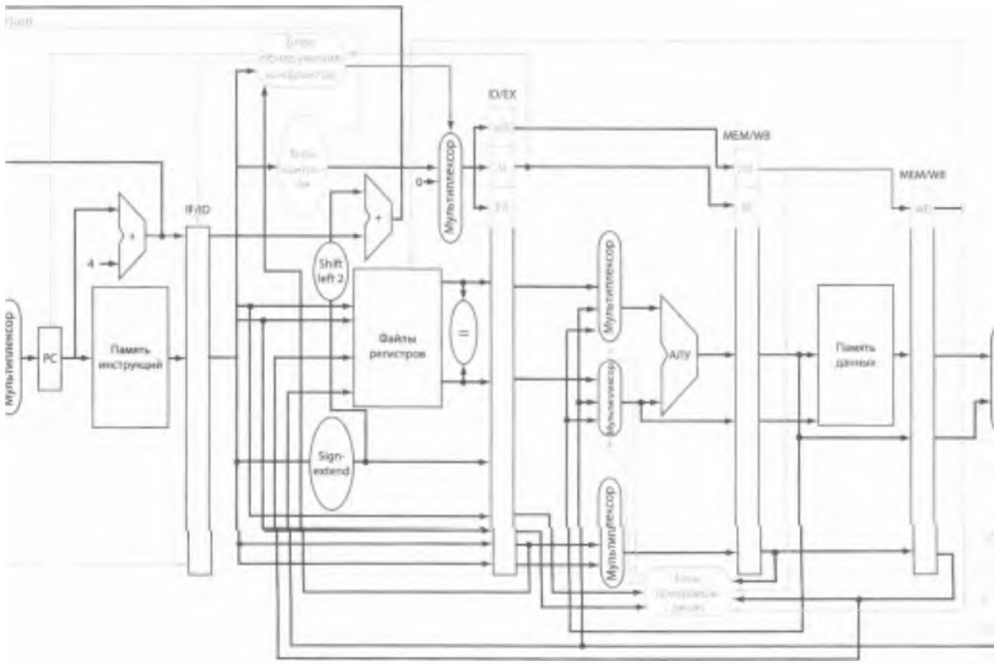


Рис. 4.58. Окончательный вид операционного блока и блока управления. Следует учесть, что это условное, а не подробное изображение операционного блока, поэтому здесь опущен мультиплексор, управляемый сигналом ALUSrc, показанный на рис. 4.50, и сигналы управления мультиплексорами, показанные на рис. 4.45

Самопроверка

Рассмотрите три схемы прогнозирования условного перехода: прогнозирование того, что переход не состоится, того, что он состоится, и динамическое прогнозирование. Предположим, что все эти схемы свободны от издержек при удачном прогнозировании и имеют издержку в два цикла при неудачном прогнозировании. Предположим, что средняя точность прогнозирования динамического предсказания составляет 90%. Какой из блоков прогнозирования станет лучшим выбором для следующих условных переходов?

1. Условный переход, который выполняется с частотой 5%.
2. Условный переход, который выполняется с частотой 95%.
3. Условный переход, который выполняется с частотой 70%.

4.9. Исключения

> Создание компьютера с возможностями автоматического [последовательного] программного прерывания не было легкой задачей, поскольку количество инструкций, находящихся на разных стадиях процесса выполнения при поступлении сигнала прерывания, может быть довольно большим.
>
> *Фрэд Брукс-младший*

Управление — наиболее сложный аспект конструирования процессора: в нем сложнее всего правильно разобраться и не менее сложно добиться от него быстрой работы. Одной из наиболее трудных составляющих системы управления является реализация **исключений** и **прерываний** — событий, которые, помимо условных или безусловных переходов, изменяют нормальное течение выполнения инструкций. Первоначально они создавались для обработки неожиданных событий внутри процессора, например арифметического переполнения. Тот же базовый механизм, как будет показано в главе 6, был расширен для устройств ввода-вывода с целью их связи с процессором.

Многие архитектуры и авторы не делают различий между прерываниями и исключениями, часто используя более раннее название *прерывание* для ссылки на оба типа событий. Например, в Intel x86 используется термин «прерывание». Мы же, следуя принятому в MIPS соглашению, используем термин *исключение* для ссылки на *любое* неожиданное изменение в алгоритме управления, не различая при этом внутреннюю или внешнюю причину его возникновения; термин *прерывание* используется, только когда событие вызвано

Исключение
Также называется прерыванием. Незапланированное событие, прерывающее выполнение программы; используется при обнаружении переполнения.

Прерывание
Исключение, которое исходит не из процессора. (В некоторых архитектурах термин «прерывание» используется для всех исключений.)

внешними причинами. Ниже приводятся пять примеров, показывающих создание ситуации как процессором, так и внешними факторами:

Тип события	Источник происхождения	Терминология MIPS
Запрос устройства ввода-вывода	Внешний	Прерывание
Вызов операционной системы из пользовательской программы	Внутренний	Исключение
Арифметическое переполнение	Внутренний	Исключение
Использование неопределенной инструкции	Внутренний	Исключение
Отказы оборудования	Как внутренний, так и внешний	Исключение или прерывание

Многие из требований по поддержке исключений исходят из определенной ситуации, являющейся причиной исключения. Соответственно мы вернемся к этой теме в главе 5, где будут рассматриваться иерархии памяти, и в главе 6, где будет рассматриваться ввод-вывод, и тогда мы лучше поймем мотивацию создания дополнительных возможностей в механизме исключений. В этом разделе рассматривается реализация управления для обнаружения двух видов исключений, вызываемых уже рассмотренной частью набора инструкций.

Обнаружение исключительных условий и выполнение соответствующих действий часто находится на критическом пути синхронизации процессора, который определяет продолжительность тактового цикла, а следовательно, и производительность. Без должного внимания к исключениям на этапе конструирования блока управления попытка добавить исключения к сложной реализации может существенно снизить производительность, а также усложнить задачу получения правильной конструкции.

Порядок обработки исключений в архитектуре MIPS

К двум типам исключений, которые могут быть сгенерированы нашей текущей реализацией, относятся выполнение неопределенной инструкции и арифметическое переполнение. В качестве примера исключения на следующих нескольких страницах мы воспользуемся арифметическим переполнением в инструкции `add $1, $2, $1`. Основным действием, которое должно быть выполнено процессором при возникновении исключения, является сохранение адреса проблемной инструкции в *счетчике команд исключения* — *EPC* (exception program counter) и передача управления операционной системе на некий заданный адрес.

Затем операционная система может предпринять соответствующее действие, которое может включать в себя предоставление пользовательской программе некой услуги, выполнение некоторого предопределенного действия в ответ на переполнение или останов выполнения программы и выдачу сообщения об ошибке.

выполнения какого-либо действия, вызванного исключением, операционная система может прекратить выполнение программы или может продолжить ее выполнение, используя EPC для определения места возобновления выполнения программы. В главе 5 вопрос возобновления выполнения будет рассмотрено более подробно.

Чтобы операционная система обработала исключение, ей, кроме вызвавшей его инструкции, должна быть известна причина исключения. Для сообщения о причине исключения существует два основных метода. Метод, используемый в архитектуре MIPS, заключается во включении регистра состояния (который называется *регистром причины* — Cause register), который содержит поле, показывающее причину исключения.

Второй метод заключается в использовании **векторных прерываний**. В векторном прерывании адрес, на который передается управление, определяется причиной исключения. Например, для обслуживания двух вышеперечисленных типов исключения мы можем определить следующие два адреса векторов исключений:

Тип исключения	Адрес вектора исключения (в шестнадцатеричном виде)
Неопределенная инструкция	$8000\ 0000_{16}$
Арифметическое переполнение	$8000\ 0180_{16}$

Операционная система знает причину исключения по адресу своей инициализации. Адреса разделены 32 байтами или восемью инструкциями, и операционная система в эту последовательность должна записать причину исключения и произвести некоторую ограниченную обработку. Когда исключение не является векторным, для всех исключений может быть использована одна и та же точка входа, и для определения причины операционная система декодирует регистр состояния.

Мы можем выполнить обработку, требуемую для исключений, путем добавления к нашей базовой реализации нескольких дополнительных регистров и сигналов управления и небольшого расширения системы управления. Давайте предположим, что нами реализуется система исключений, используемая в архитектуре MIPS, с точкой входа по адресу $8000\ 0180_{16}$. (Реализация векторных исключений дается не намного сложнее.) К реализации MIPS нужно добавить два дополнительных регистра:

- *EPC*: 32-разрядный регистр, используемый для хранения адреса инструкции, вызвавшей исключение. (Такой регистр нужен, даже когда исключения векторные.)
- *Cause*: Регистр, используемый для записи причины исключения. В архитектуре MIPS этот регистр состоит из 32 разрядов, хотя часть разрядов в настоящее время не используется.

Векторное прерывание
Прерывание, для которого адрес, куда будет передано управление, определяется причиной прерывания.

Исключения в конвейеризированной реализации

При конвейеризированной реализации исключения рассматриваются как еще одна форма конфликта управления. Предположим, к примеру, что произошло арифметическое переполнение при выполнении инструкции сложения — add. Точно так же как это делалось в предыдущем разделе для состоявшегося условного перехода, мы должны сбросить с конвейера инструкции, следующие за инструкцией сложения, и приступить к извлечению инструкции с нового адреса. Будет использоваться тот же механизм, который использовался для состоявшихся условных переходов, но на этот раз причиной сброса сигналов на линиях управления станет исключение.

Когда мы имели дело с неправильно спрогнозированным условным переходом, мы видели, как сбросить инструкцию на стадии IF путем превращения ее в пустую инструкцию — nop. Для сброса инструкций на стадии ID мы используем уже имеющийся на этой стадии мультиплексор, который обнуляет сигналы управления для задержки.

Новый сигнал управления, который называется ID.Flush, подвергается операции ИЛИ вместе с сигналом задержки из блока обнаружения конфликта для сброса инструкции на стадии ID. Для сброса инструкции на стадии EX мы используем новый сигнал, который называется EX.Flush, чтобы заставить новые мультиплексоры обнулить сигналы на линиях управления. Для начала извлечения инструкций с адреса $8000\ 0180_{16}$, который является в MIPS адресом исключений, мы просто добавляем еще один вход в мультиплексор PC, передающий в PC адрес $8000\ 0180_{16}$. Эти изменения показаны на рис. 4.59.

Этот пример вскрывает проблему, связанную с исключениями: если мы не остановим выполнение на середине инструкции, программист не сможет увидеть исходное значение регистра $1, который помог привести к переполнению, поскольку оно будет затерто из-за того, что этот регистр в инструкции сложения является регистром-получателем. Благодаря тщательно спланированным действиям исключение, связанное с переполнением будет обнаружено на стадии EX; следовательно, сигнал EX.Flush может быть использован для того, чтобы инструкция, находящаяся на стадии EX, не могла записывать свои результаты на стадии WB. Многие исключения требуют, чтобы мы, в конечном счете, завершили выполнение инструкции, вызвавшей исключение, как будто она выполнялась нормально. Простейший способ реализации этого замысла заключается в сбросе инструкции и ее повторном запуске с самого начала после того, как исключение будет обработано.

Завершающим шагом станет сохранение адреса инструкции, вызвавшей исключение в счетчике команд исключений (EPC). В действительности сохраняется адрес + 4, поэтому подпрограмма обработки исключения должная сначала уменьшить сохраненное значение на 4. На рис. 4.59 показана условная версия операционного блока, включающая оборудование для условных переходов и необходимые приспособления для обработки исключений.

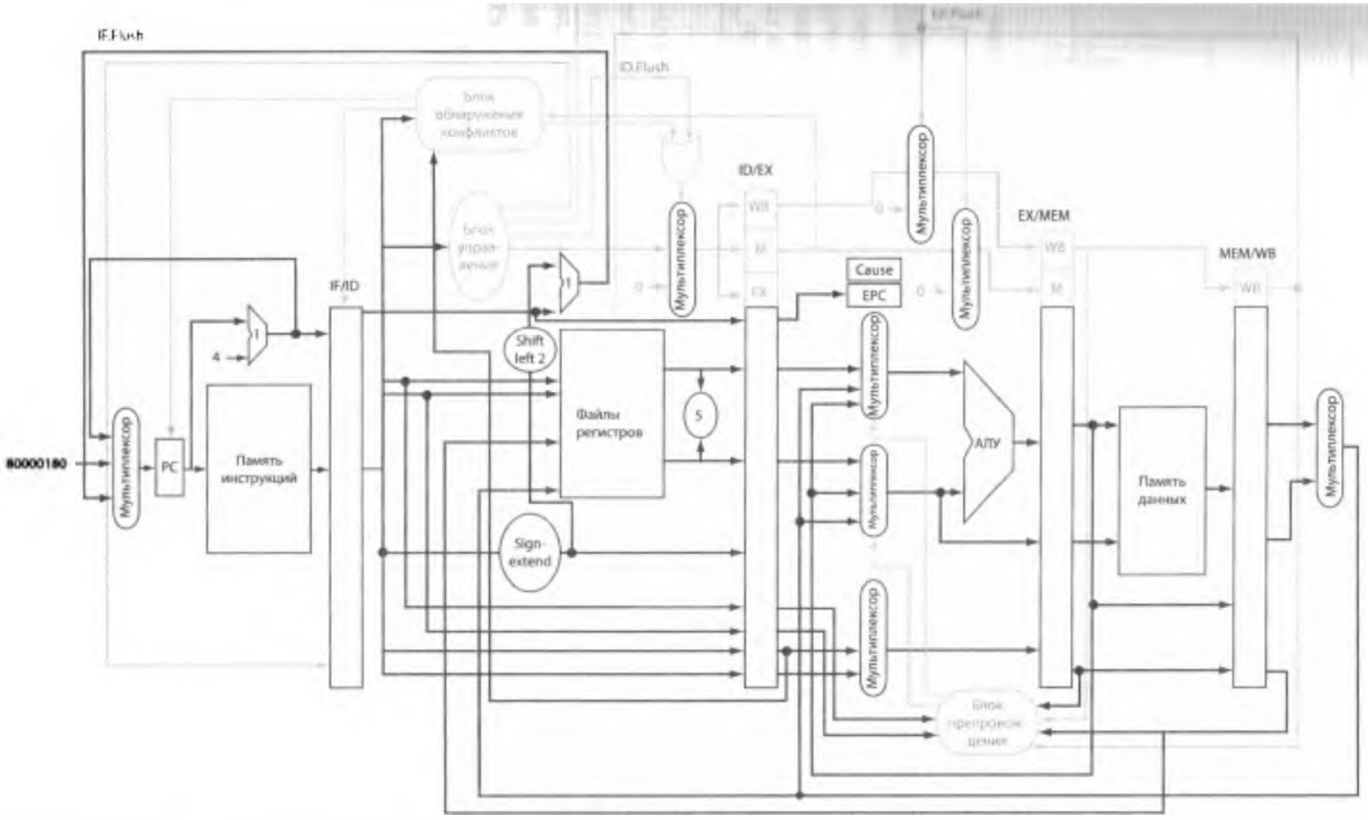


Рис. 4.59. Операционный блок с системой управления для обработки исключений. Ключевые дополнения состоят из нового входа в мультиплексор со значением 8000 0180_{16}, который предоставляет новое значение для счетчика команд (PC), регистра для записи причины исключения (Cause), и регистра счетчика команд исключения (EPC) для сохранения адреса инструкции, вызвавшей исключение. Вход в мультиплексор 8000 0180_{16} является исходным адресом для начала извлечения инструкций в случае исключения. Здесь еще не показан в качестве входа в блок управления сигнал переполнения АЛУ

Упражнение

Исключение в конвейеризированном компьютере

Предположим, что для последовательности инструкций

```
4016  sub  $11, $2, $4
4416  and  $12, $2, $5
4816  or   $13, $2, $6
4C16  add  $1,  $2, $1
5016  slt  $15, $6, $7
5416  lw   $16, 50($7)
```

инструкции, которые будут вызваны исключением:

```
8000018016 sw $25, 1000($0)
8000018416 sw $26, 1004($0)
```

Покажите, что произойдет в конвейере, если случится исключение, связанное с переполнением в инструкции сложения (add).

Ответ

На рис. 4.60 показаны события начиная с выполнения инструкции add на стадии EX. Переполнение обнаруживается на этой стадии, и в счетчик команд принудительно вводится значение $8000\ 0180_{16}$. На тактовом цикле 7 показывается, что add и следующие инструкции сбрасываются и извлекается первая инструкция кода исключения. Обратите внимание на то, что сохраняется адрес той инструкции, которая *следует* за инструкцией add: $4C_{16} + 4 = 50_{16}$.

В начале раздела 4.9 упоминалось о пяти примерах исключений, все остальные исключения будут рассмотрены в главах 5 и 6. Когда в любом тактовом цикле имеется пять активных инструкций, связать исключение с соответствующей инструкцией является непростой задачей. Более того, в одном тактовом цикле могут случиться сразу несколько исключений. Решение заключается в установке приоритетности исключений, чтобы было проще определить, какое из них следует обслуживать в первую очередь. В большинстве MIPS-реализаций оборудование сортирует исключения таким образом, чтобы прерывалась самая ранняя из инструкций.

Запросы, поступающие от устройств ввода-вывода, и отказы оборудования не связаны с конкретной инструкцией, поэтому реализация обладает некоторой гибкостью в выборе момента прерывания работы конвейера. Поэтому механизм, используемый для других исключений, работает вполне удовлетворительно.

Регистр EPC захватывает адрес прерванных инструкций, а MIPS-регистр Cause записывает все возможные исключения, возникшие на протяжении тактового цикла, поэтому программное обеспечение исключения должно соответствовать исключению, возникшему при выполнении данной инструкции. Очень важно знать, на какой стадии конвейера может произойти то или иное исключение. Например, неопределенная инструкция выявляется на стадии ID, а вызов операционной системы происходит на стадии EX. Исключения собираются в регистре Cause в поле исключений, ждущих обработки, чтобы оборудование могло прервать свою работу на основе более поздних исключений, как только будет обслужено самое раннее исключение.

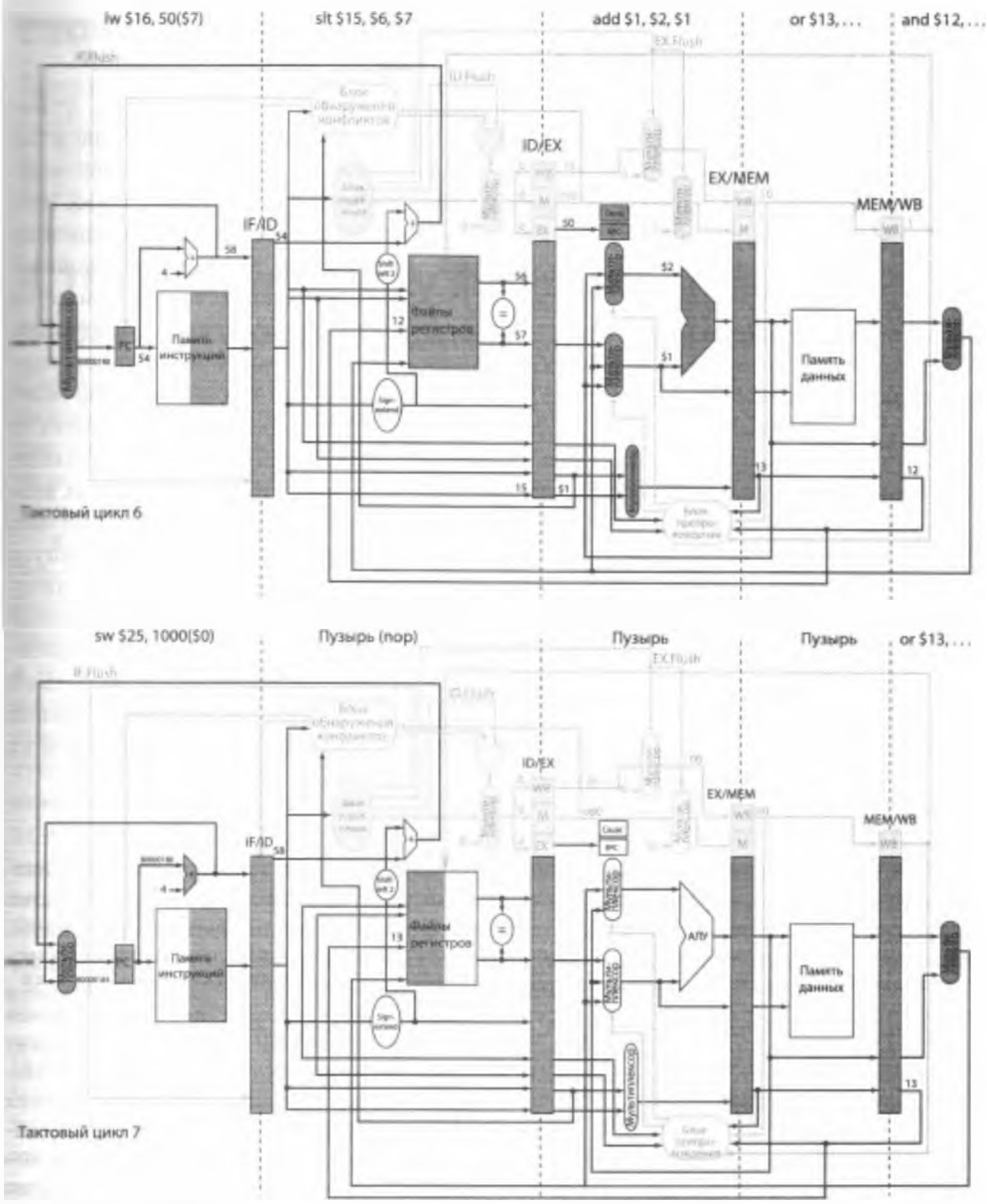


Рис. 4.60. Результат исключения, возникшего из-за арифметического переполнения при выполнении инструкции add. Переполнение было выявлено на стадии EX тактового цикла 6, при этом в регистре EPC был сохранен адрес инструкции, следующей за инструкцией add (4C + $4 = 50_{16}$). Переполнение вызвало ближе к концу этого тактового цикла выставление всех сигналов сброса — Flush, обнуляя для инструкции add значения сигналов управления. Тактовый цикл 7 показывает, что все инструкции в конвейере превращены в пузыри, и, кроме этого, он показывает извлечение первой инструкции подпрограммы обработки исключения — sw $25,1000($0) — из места в памяти инструкций $8000\ 0180_{16}$. Обратите внимание на то, что инструкции AND и OR, предшествующие инструкции add, все же завершаются. Хотя это и не показано, на вход блока управления поступает сигнал переполнения АЛУ (ALU overflow)

Интерфейс аппаратного и программного обеспечения

Оборудование и операционная система должны работать согласованно, чтобы исключения вели себя вполне ожидаемо. На оборудование обычно возлагается остановка проблемной инструкции в ходе ее выполнения, предоставление возможности для завершения выполнения всех предшествующих инструкций, сброс всех последующих инструкций, установка значения регистра, показывающего причину исключения, сохранение адреса проблемной инструкции, а затем безусловный переход на заранее подготовленный адрес. На операционную систему возлагается задача посмотреть на исключение и действовать соответствующим образом. Для неопределенной инструкции, сбоя оборудования или арифметического переполнения операционная система обычно прекращает работу программы и возвращает информацию, показывающую причину ее действий. Для запросов, поступающих от устройств ввода-вывода или сервисного вызова операционной системы, операционная система сохраняет состояние программы, выполняет нужную задачу и, впоследствии, возвращает управление программе для продолжения ее выполнения. В случае запросов устройств ввода-вывода мы можем зачастую выбрать запуск другой задачи до того, как возобновим выполнение задачи, запросившей ввод-вывод, поскольку эта задача не может выполняться до тех пор, пока не завершится ввод-вывод. Именно поэтому важно иметь возможность сохранения и восстановления состояния любой задачи. Одним из наиболее важных и частых применений исключений является обработка исключений, связанных с отсутствием нужной страницы и нужной записи в буфере TLB. Более подробно эти исключения и их обработка рассматриваются в главе 5.

Уточнение. Сложность постоянной связи соответствующего исключения с соответствующей инструкцией в конвейеризированных компьютерах заставила некоторых разработчиков компьютеров ослабить эти требования в некритических ситуациях. Про такие процессоры говорят, что у них **неточные прерывания** или **неточные исключения**. В рассмотренном выше примере счетчик команд (PC) имел значение 58_{16} в начале тактового цикла после обнаружения исключения, даже если бы проблемная инструкция находилась по адресу $4C_{16}$. Процессор с неточными исключениями может поместить в EPC адрес 58_{16} и возложить на операционную систему определение инструкции, вызвавшей проблему. MIPS и подавляющее большинство сегодняшних компьютеров поддерживают **точные прерывания** или **точные исключения**. (Одной из причин является поддержка виртуальной памяти, которая будет рассматриваться в главе 5.)

Неточное прерывание
Также называется неточным исключением. Прерывания или исключения в конвейеризированных компьютерах, не связанные с конкретной инструкцией, ставшей причиной прерывания или исключения.

Точное прерывание
Также называется точным исключением. Прерывание или исключение, которое всегда связано с соответствующей инструкцией в конвейеризированных компьютерах.

Уточнение. Хотя почти для всех исключений в MIPS используется адрес входа $8000\ 0180_{16}$, для повышения производительности в этой системе в качестве адреса обработчика исключения, связанного с отсутствием нужного адреса в TLB, используется адрес $8000\ 0000_{16}$ (см. главу 5).

Самопроверка

Какое исключение должно быть обнаружено первым в данной последовательности?

```
1. add $1, $2, $1    # арифметическое переполнение
2. lw  $1, $2, $1    # неопределенная инструкция
3. sub $1, $2, $1    # ошибка оборудования
```

4.10. Параллелизм и расширенный параллелизм на уровне инструкций

Следует предупредить, что данный раздел представляет собой краткий обзор очень интересных, но расширенных тем. Если нужно получить более подробные сведения, следует обратиться к нашей книге «Computer Architecture: A Quantitative Approach», четвертое издание, где материал, рассматриваемый на десятке следующих страниц, расширен почти до 200 страниц (включая приложения)!

Конвейеризация использует потенциальный параллелизм среди инструкций. Этот параллелизм называется **параллелизмом на уровне инструкций** (instruction-level parallelism — ILP). Существуют два основных метода повышения потенциального объема параллелизма на уровне инструкций. Первый из них заключается в углублении конвейера для выполнения одновременно большего количества инструкций. Используя нашу аналогию с прачечной, при условии, что цикл стирки продолжается дольше всех остальных, мы можем распределить функцию одной стиральной машины на три машины, выполняющие такие шаги традиционной стиральной машины, как стирка, полоскание и отжим. Тогда мы сможем перейти от конвейера с четырьмя стадиями к конвейеру с шестью стадиями. Для получения полного ускорения нам нужно перебалансировать оставшиеся шаги, чтобы они имели одинаковую продолжительность в процессорах (или в прачечной). Объем используемого параллелизма становится выше, поскольку одновременно выполняется большее количество инструкций. Производительность становится потенциально выше, поскольку тактовый цикл может быть короче.

Второй подход заключается в дублировании внутренних компонентов компьютера, позволяющем запускать сразу несколько инструкций на каждой стадии конвейера. Обобщенное название этой технологии — **параллельный запуск**. В прачечной, использующей параллельный запуск, нужно заменить наши бытовые стиральную и сушильную машины, скажем, тремя стиральными и тремя сушильными машинами. Нужно будет также нанять больше помощников для складывания белья и его транспортировки к месту хранения за прежнее время. Недостатком этого подхода является дополнительная работа по загрузке всех машин и по переносу партий белья на следующую стадию конвейера.

Запуск сразу нескольких инструкций на одной стадии позволяет частоте выполнения инструкций превысить тактовую частоту или, иначе говоря, достичь показателя CPI, меньше-

Параллелизм на уровне инструкций
Параллелизм инструкций.

Параллельный запуск
Схема, при которой в одном тактовом цикле запускается сразу несколько инструкций.

го единицы. Иногда полезно поменять местами составляющие показателя и воспользоваться *IPC*, или *количеством инструкций за один тактовый цикл*. Исходя из этого, микропроцессор с тактовой частотой 4 ГГц с четырехкратным параллельным запуском инструкций может на пиковой скорости выполнять по 16 млрд инструкций в секунду и иметь наивысший показатель CPI, равный 0,25, или IPC, равный 4. Если представить, что у этого микропроцессора конвейер имеет пять стадий, то в каждый момент времени на нем будут выполняться 20 инструкций. Современные высокопроизводительные микропроцессоры запускают за каждый тактовый цикла от трех до шести инструкций. Но на типы одновременно выполняемых инструкций и на события, происходящие в случаях возникновения взаимозависимостей, обычно накладывается множество ограничений.

Для реализации процессора с параллельным запуском инструкций используются два способа, основное отличие которых друг от друга заключается в распределении работы между компилятором и оборудованием. Поскольку распределение работ диктуется теми решениями, которые были приняты в статическом (то есть во время компиляции) или в динамическом режиме (то есть во время выполнения), подходы к нему иногда называют **статическим параллельным запуском** и **динамическим параллельным запуском**. Мы еще увидим, что у обоих подходов есть и другие, более распространенные, но менее точные или более ограниченные названия.

Есть две прямые и конкретные обязанности, с которыми должен справиться конвейер с параллельным запуском:

1. Составление пакета инструкций в **слотах запуска**. Как процессор определяет, сколько и каких инструкций может быть запущено в заданном тактовом цикле? Во многих процессорах со статическим параллельным запуском этот процесс, по крайней мере частично, возлагается на компилятор, а в конструкциях динамического параллельного запуска решением этого вопроса обычно занимается процессор в ходе выполнения программы, хотя компилятор зачастую будет пытаться повысить частоту запуска инструкций, размещая их в более рациональном порядке.
2. Разрешение конфликтов данных и управления: в случае применения процессоров со статическим параллельным запуском некоторые или даже все последствия конфликтов данных и управления устраняются статически компилятором. В отличие от этого многие процессоры с динамическим параллельным запуском пытаются снизить последствия по крайней мере некоторых классов конфликтов, используя аппаратные технологии, задействованные во время выполнения программы.

Статический параллельный запуск
Подход к реализации процессора с параллельным запуском инструкций, при котором многие решения принимаются компилятором еще до выполнения программы.

Динамический параллельный запуск
Подход к реализации процессора с параллельным запуском инструкций, при котором многие решения принимаются процессором в ходе выполнения программы.

Слоты запуска
Позиции, с которых инструкции могут запускаться в заданный тактовый цикл; если проводить аналогию, то они соответствуют позициям стартовых колодок для спринтерского бега.

Хотя все это в наших описаниях преподносится в виде отдельных подходов, на самом деле методики, применяемые в одном подходе, часто

ствуются другим подходом, и нельзя утверждать об абсолютной чистоте того ного подхода.

нятие предположения

им из наиболее важных методов получения и использования большей степе- параллелизма на уровне инструкций является выстраивание предположения. **страивание предположение** — это подход, позволяющий компилятору или цессору «догадаться» о свойствах инструкции, чтобы можно было приступить полнению других инструкций, которые могут зависеть от той инструкции, ношении которой выстраивалось предположение. Например, мы можем выстра- ть предположение относительно исхода условного перехода, позволяя заранее олнить инструкции, следующие за этим переходом. Можно также, к примеру, едположить, что сохранение, предшествующее загрузке, не ссылается на тот же рес, что позволит загрузке быть выполненной до сохранения. Трудности, воз- кающие при использовании предположения, заключаются в том, что оно может ть неверным. Поэтому любой механизм предположений должен включать как тод проверки того, была ли догадка верна, так и метод устранения последствий инструкций, выполненных на основе предположения. Реализация таких возмож- стей по возвращению к исходным позициям усложняет конструкцию.

Предположение может выстраиваться компилятором или оборудованием. На- имер, компилятор может использовать предположение для изменения порядка следования инструкций, перемещения инструкции через инструкцию условного ерехода или перемещения инструкции загрузки через инструкцию сохранения. Оборудование процессора может осуществлять такие же преобразования в ходе ыполнения программы, используя технологии, которые еще будут рассмотрены анном разделе.

Механизмы восстановления, используемые для неправильных предположений, тличаются разнообразием. В том случае, когда предположение выстраивается рограммным обеспечением, компилятор обычно вставляет дополнительные ин- струкции, проверяющие точность предположения, и предоставляет подпрограмму справления ситуации, использующуюся в случае несостоятельности предполо- ения. В том случае, когда предположение выстраивается оборудованием, про- ессор обычно помещает в буфер результаты действий, предпринятых на основе редположения, до тех пор, пока не убедится в том, что они больше не являются редположительными. Если предположение было правильным, выполнение ин- струкций завершается за счет того, что содержимому буферов позволяется быть записанным в регистры или в память. Если предположение было неверным, обо- рудование сбрасывает буферы и выполняет правильную последовательность ин- струкций.

Выстраивание предположения становится причиной возникновения еще одной возмож- ной проблемы: предположение в отношении определенных инструкций может вызвать ис- ключения, которых раньше не было. Допустим,

Выстраивание предположения
Подход, благодаря которому компилятор или процессор догадывается о последствиях выполнения инструкции для их устранения в случае, если они оказывают влияние на выполнение других инструкций.

к примеру, что инструкция загрузки была перемещена на основе предположения, но если предположение ошибочно, используемый этой инструкцией адрес является недопустимым. Результатом может стать возникновение такого исключения, которое не должно было произойти. Проблема усложняется еще и тем фактом, что если предположение не выстраивалось в отношении инструкции загрузки, то может возникнуть исключение! В предположении, выстроенном компилятором, подобные проблемы обходятся за счет добавления специальной поддержки предположений, позволяющей подобным исключениям быть проигнорированными до тех пор, пока не станет ясно, что они на самом деле должны произойти. В предположении, выстроенном оборудованием, исключения просто буферизуются до тех пор, пока не станет ясно, что вызвавшая их инструкция не является больше предположительной и готова к завершению; в этот момент и вызывается исключение и происходит его обычная обработка.

Поскольку верное выстраивание предположений может повысить, а неверное снизить производительность, для решения, когда нужно выстраивать предположение, прилагаются значительные усилия. Далее в этом разделе будут исследованы как статические, так и динамические технологии выстраивания предположений.

Статический параллельный запуск

Все процессоры со статическим параллельным запуском используют компилятор для помощи в составлении пакетов инструкций и разрешении конфликтов. При использовании процессора со статическим параллельным запуском можно говорить о наборе инструкций, запущенных в заданном тактовом цикле, как о **пакете запуска** в виде одной большой инструкции с несколькими операциями. Это представление превыше всяких аналогий. Поскольку процессор со статическим параллельным запуском обычно накладывает ограничения на то, какое сочетание инструкций может быть запущено в отдельно взятом тактовом цикле, пакет запуска полезнее всего представлять в виде одной инструкции, разрешающей в конкретных предопределенных полях проведение сразу нескольких операций. Этот взгляд приводит к своеобразному названию такого подхода: **слово инструкции очень большой длины** — Very Long Instruction Word (**VLIW**).

Пакет запуска
Набор инструкций, запускаемых вместе в одном тактовом цикле; пакет может быть подобран статически компилятором или динамически процессором.

Слово инструкции очень большой длины (VLIW)
Название набора инструкций, запускающего множество операций, определенных независимыми друг от друга, и рассматриваемого в виде одной расширенной инструкции, которая обычно имеет множество отдельных полей opcode.

Большинство процессоров со статическим параллельным запуском также полагаются на то, что компилятор возьмет на себя некоторую ответственность за разрешение конфликтов данных и управления. Ответственность компиляторов может включать в себя статическое прогнозирование условных переходов и диспетчеризацию кода с целью сокращения или предотвращения всех конфликтов. Перед тем как рассматривать описание использования этих технологий на более активных процессорах, давайте посмотрим на простую статическую версию MIPS-процессора.

Пример: статический параллельный запуск с применением архитектуры набора инструкций MIPS

Чтобы показать, что такое статический параллельный запуск, мы рассмотрим простой MIPS-процессор с двумя параллельно запускаемыми инструкциями, где одна из инструкций может быть целочисленной АЛУ-операцией или условным переходом, а другая может быть инструкцией загрузки или сохранения. Такая конструкция похожа на используемую в некоторых встроенных MIPS-процессорах. Запуск двух инструкций за один тактовый цикл потребует извлечения и декодирования 64 разрядов инструкций. Во многих процессорах со статическим параллельным запуском и практически во всех VLIW-процессорах формат одновременно запускаемых инструкций ограничен с целью упрощения их декодирования и запуска. Поэтому нам потребуется, чтобы инструкции были сдвоены и выровнены по 64-разрядной границе и чтобы первыми следовали инструкции выполнения АЛУ-операций или условных переходов. Более того, если одна инструкция в паре не может быть использована, ее нужно заменить пустой инструкцией — nop. Таким образом, инструкции всегда запускаются парами, при этом в одном из слотов может быть инструкция nop. Поступление инструкций на конвейер попарно показано в табл. 4.8.

Таблица 4.8. Конвейер со статическим параллельным запуском двух инструкций в работе. Одновременный запуск инструкций АЛУ-операции и переноса данных. В данном случае предполагается использование такого же конвейера из пяти стадий, который использовался при последовательном запуске инструкций. Хотя в этом и нет особой необходимости, он обладает рядом преимуществ. В частности, то, что в нем по-прежнему запись в регистр ведется в конце конвейера, упрощает обработку исключений и поддержку модели точных исключений, которая в процессорах с параллельным запуском инструкций заметно усложняется

Тип инструкции	Стадии конвейера							
Инструкция АЛУ-операции или условного перехода	IF	ID	EX	MEM	WB			
Инструкция загрузки или сохранения	IF	ID	EX	MEM	WB			
Инструкция АЛУ-операции или условного перехода		IF	ID	EX	MEM	WB		
Инструкция загрузки или сохранения		IF	ID	EX	MEM	WB		
Инструкция АЛУ-операции или условного перехода			IF	ID	EX	MEM	WB	
Инструкция загрузки или сохранения			IF	ID	EX	MEM	WB	
Инструкция АЛУ-операции или условного перехода				IF	ID	EX	MEM	WB
Инструкция загрузки или сохранения				IF	ID	EX	MEM	WB

Процессоры со статическим параллельным запуском отличаются друг от друга тем, как они справляются с потенциальными конфликтами данных и управления. В некоторых конструкциях компилятор берет на себя всю ответственность за устранение *всех* конфликтов, диспетчеризацию кода и вставку пустых инструкций, чтобы при выполнении кода не требовалось обнаружение конфликтов или аппаратное генерирование задержек. В других конструкциях оборудование обнаруживает конфликты данных и генерирует задержки между двумя пакетами запуска, по-прежнему требуя, чтобы компилятор избегал всех зависимостей внутри пар инструкций. Но даже при этих условиях конфликт, как правило, заставляет задерживать весь пакет запуска, в котором содержится зависимая инструкция. Появление больших единых инструкций, состоящих из нескольких операций, обостряет вопрос относительно того, должно ли программное обеспечение справляться со всеми конфликтами или оно должно лишь пытаться сократить долю конфликтов между отдельными пакетами запуска. Для данного примера мы предположим, что применяется второй подход.

Для осуществления параллельного запуска операций АЛУ и переноса данных в первую очередь требуется наличие дополнительного оборудования сверх обычного определения конфликтов и логики задержек — дополнительных портов в файле регистров (рис. 4.61). В течение одного тактового цикла может понадобиться чтение двух регистров для операции АЛУ и чтение еще двух регистров для операции сохранения, а также один порт записи для операции АЛУ и один порт записи для операции загрузки. Поскольку АЛУ связано операцией АЛУ, нам также понадобится отдельный сумматор на вычисляющий эффективный адрес для переноса данных. Без этих дополнительных ресурсов наш конвейер с параллельным запуском двух инструкций будет блокироваться структурными конфликтами.

Вполне очевидно, что этот процессор с параллельным запуском двух инструкций может повысить производительность почти в два раза. Но при этом нужно, чтобы одновременно выполнялось вдвое больше инструкций, и это дополнительное перекрытие увеличивает относительные потери производительности, обусловленные конфликтами данных и управления. Например, в нашем простом конвейере, имеющем пять стадий, загрузка имеет **задержку использования** в один тактовый цикл, которая не дает инструкции использовать результаты без задержки. В конвейере с параллельным запуском двух инструкций и пятью стадиями результат инструкции загрузки не может быть использован в следующем тактовом цикле. Это означает, что следующие две инструкции не могут использовать результаты загрузки без задержки. Более того, АЛУ-инструкции, которые не использовали задержку в простом конвейере из пяти стадий, теперь имеют задержку использования в одну инструкцию, поскольку результаты не могут быть использованы в парных инструкциях загрузки или сохранения. Для эффективного использования параллелизма, доступного в процессорах с параллельным запуском инструкций, необходимы более серьезные компиляторы или технологии аппаратной диспетчеризации, а статический параллельный запуск инструкций требует, чтобы эта роль возлагалась на компилятор.

Задержка использования
Количество тактовых циклов между инструкцией загрузки и инструкцией, которая может использовать результат загрузки без холостой работы конвейера.

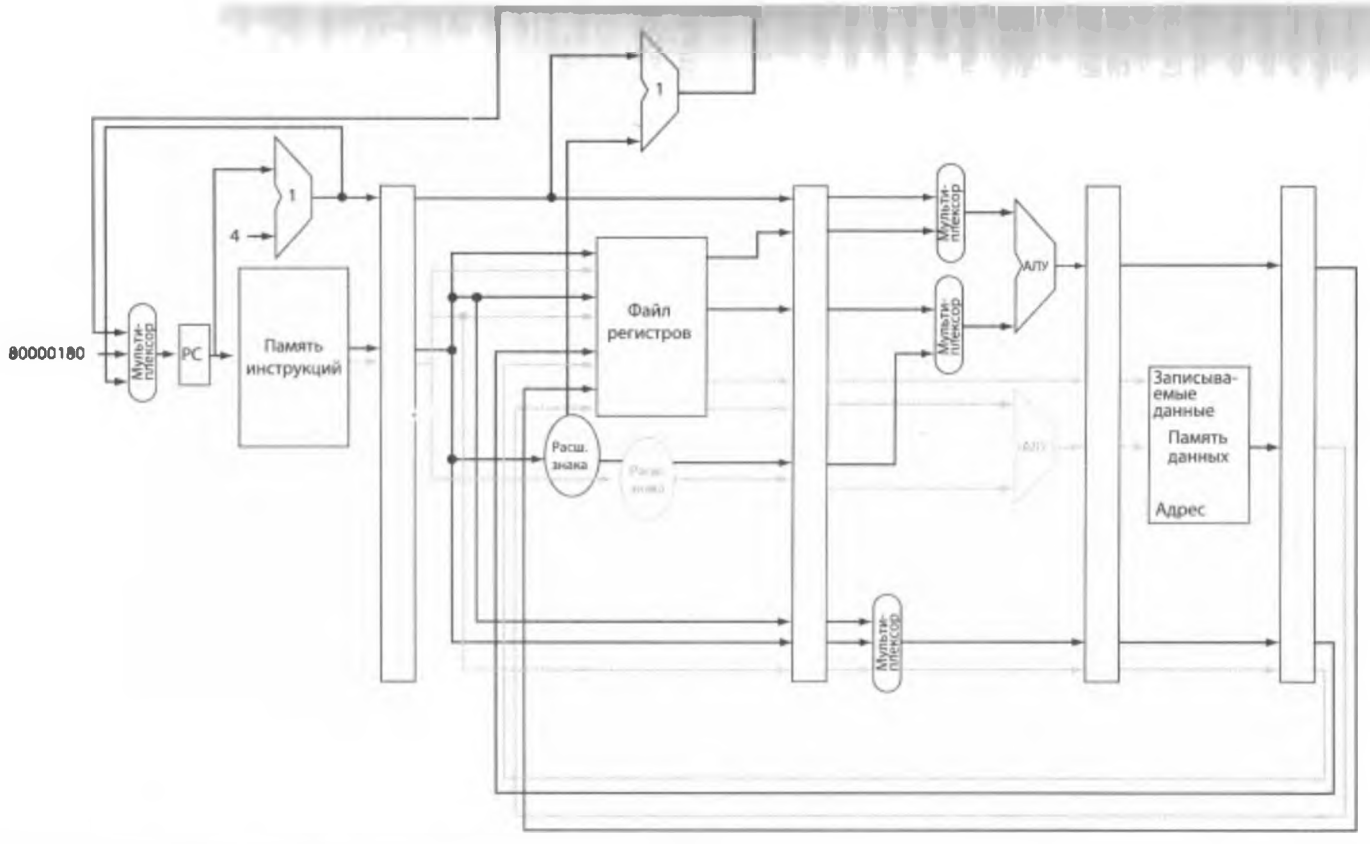


Рис. 4.61. Операционный блок для статического параллельного запуска двух инструкций. На рисунке выделены дополнительные элементы оборудования, необходимые для параллельного запуска двух инструкций: дополнительные 32 разряда из памяти инструкций, два дополнительных порта чтения и один дополнительный порт записи в файле регистров и еще одно АЛУ. Предполагается, что нижнее АЛУ занимается вычислением адресов для инструкций переноса данных, а верхнее АЛУ занимается всем остальным

Упражнение

Простая диспетчеризация кода при параллельном запуске инструкций
Как можно было бы провести диспетчеризацию следующего цикла для статического конвейера с параллельным запуском двух MIPS-инструкций?

```
Loop:  lw      $t0, 0($s1)     # $t0=элемент массива
       addu    $t0,$t0,$s2     # прибавление скаляра из $s2
       sw      $t0, 0($s1)     # сохранение результата
       addi    $s1,$s1,-4      # декремент указателя
       bne     $s1,$zero,Loop  # условный переход, если $s1!=0
```

Измените порядок следования инструкций, чтобы свести к минимуму холостую работу конвейера, при условии, что условные переходы спрогнозированы и с конфликтами управления справляется оборудование.

Ответ

В первых трех инструкциях, как и в последних двух, имеется зависимость от данных. В табл. 4.9 показан лучший план выполнения этих инструкций. Следует заметить, что только одна пара инструкций задействует оба слота запуска. Каждая итерация программного цикла занимает четыре тактовых цикла; при четырех тактовых циклах на выполнение пяти инструкций мы получаем обескураживающий показатель CPI, равный 0,8, по сравнению с наилучшим из возможных, равным 0,5, или показатель IPC, равный 1,25, по сравнению с 2,0. Учтите, что при вычислении CPI или IPC мы не брали в расчет пустые инструкции nop, выполняемые в качестве полезных. Если бы это было сделано, то улучшился бы показатель CPI, но не производительность!

Таблица 4.9. Код после диспетчеризации, как он мог бы выглядеть в MIPS-конвейере с параллельным запуском двух инструкций. Пустые слоты представляют собой инструкции nop

	Инструкция АЛУ или условного перехода	Инструкция переноса данных	Тактовый цикл
Loop:		lw $t0, 0($s1)	1
	addi $s1,$s1,–4		2
	addu $t0,$t0,$s2		3
	bne $s1,$zero,Loop	sw $t0, 4($s1)	4

Для достижения более высокой производительности при выполнении циклов компиляторами применяется весьма ценная технология, называемая **развертыванием цикла**, предусматривающая множественное копирование тела цикла. После развертывания можно достичь более высокой степени параллелизма на уровне инструкций за счет одновременного выполнения инструкций из разных итераций.

Развертывание цикла
Технология достижения более высокой производительности от циклов, используемых для обращения к массивам, предусматривающая создание множественных копий тела цикла и общую диспетчеризацию инструкций из разных итераций.

Упражнение

Развертывание циклов для конвейеров с параллельным запуском нескольких инструкций
Посмотрим, насколько хорошо работают развертывание цикла и диспетчеризация применительно к ранее показанному примеру. Для простоты представим, что индекс цикла кратен четырем.

Ответ

Получается, что для диспетчеризации цикла без задержек нам нужно сделать четыре копии тела цикла. После развертывания и исключения ненужных инструкций, составляющих издержки цикла, все, что останется от цикла, будет представлять собой четыре копии каждой из инструкций lw, add и sw плюс одну инструкцию addi и одну инструкцию bne. Развернутый код, прошедший диспетчеризацию, показан в табл. 4.10.

В процессе развертывания компилятор вводит в действие дополнительные регистры ($t1, $t2, $t3). Целью этого процесса, который называется **переименованием регистров**, является исключение зависимостей, не являющихся настоящими зависимостями данных, но способных либо привести к потенциальным конфликтам, либо помешать компилятору выполнить правильную диспетчеризацию кода. Посмотрим, как выглядел бы развернутый код, если бы использовался только регистр $t0. Он содержал бы повторяющиеся экземпляры инструкций lw $t0,0($s1), addu $t0,$t0,$s2, за которыми следовала бы инструкция sw t0,4($s1), но эти последовательности, если не брать в расчет использование регистра $t0, в действительности совершенно независимы друг от друга — между одной парой этих инструкций и следующей такой парой нет никакого потока данных. Такое положение называется **антизависимостью** или зависимостью **от имени**, то есть предопределенностью, явно вызванной повторным использованием имени, а не реальной зависимостью от данных, которая еще называется настоящей зависимостью.

Переименование регистров в ходе процесса развертывания позволяет компилятору переместить эти независимые инструкции, сообразуясь с наилучшей диспетчеризацией кода. Процесс переименования исключает зависимости от имени, сохраняя при этом настоящие зависимости.

Обратите внимание на то, как 12 из 14 инструкций в цикле выполняются парами. На четыре итерации цикла затрачивается восемь тактовых циклов, или по два цикла на итерацию, что дает значение CPI, равное 8/14 = 0,57. Развертывание цикла и диспетчеризация при параллельном запуске двух инструкций дают нам фактор улучшения, приближающийся к 2, частично за счет сокращения инструкций управления циклом и частично за счет выполнения кода с параллельным запуском двух инструкций. Ценой такого улучшения производительности стало использование четырех временных регистров вместо одного, а также существенное увеличение размера кода.

Переименование регистров
Переименование регистров компилятором или оборудованием для устранения антизависимости.

Антизависимость
Также называемая зависимостью от имени. Предопределенность, вызванная повторным использованием имени, обычно регистра, в отличие от настоящей зависимости, переносящей значение между двумя инструкциями.

Таблица 4.10. Код, показанный в табл. 4.9, прошедший развертывание и диспетчеризацию и представленный в том виде, в котором он был бы использован в MIPS-конвейере с параллельным запуском двух инструкций. Пустые слоты являются пустыми инструкциями — nop. Поскольку в первой инструкции цикла значение $s1 уменьшается на 16, исходное значение адреса загрузок $s1, затем этот адрес последовательно уменьшается на 4, 8 и 12

	Инструкция АЛУ или условного перехода	Инструкция переноса данных	Тактовый цикл
Loop:	addi $s1,$s1,-16	lw $t0, 0($s1)	1
		lw $t1,12($s1)	2

продолжение

Таблица 4.10 (*продолжение*)

	Инструкция АЛУ или условного перехода	Инструкция переноса данных	Тактовый цикл
	addu $t0,$t0,$s2	lw $t2, 8($s1)	3
	addu $t1,$t1,$s2	lw $t3, 4($s1)	4
	addu $t2,$t2,$s2	sw $t0, 16($s1)	5
	addu $t3,$t3,$s2	sw $t1,12($s1)	6
		sw $t2, 8($s1)	7
	bne $s1,$zero,Loop	sw $t3, 4($s1)	8

Процессоры с динамическим параллельным запуском

Процессоры с динамическим параллельным запуском также известны как **суперскалярные** процессоры, или простые суперскаляры. В самых простых суперскалярных процессорах инструкции запускаются по порядку, и процессор решает, сколько инструкций: ноль, одна или более — могут быть запущены в отдельно взятый тактовый цикл. Вполне очевидно, что достижение хорошей производительности на таком процессоре по-прежнему требует от компилятора диспетчеризации инструкций для обособленного перемещения зависимостей и повышения, таким образом, показателя запуска инструкций. Но даже при такой диспетчеризации, проводимой компилятором, между эти простым суперскаляром и VLIW-процессором имеется весьма существенная разница: оборудование гарантирует правильное выполнение кода независимо от того, прошел он диспетчеризацию или нет. Более того, скомпилированный код будет всегда работать правильно, независимо от показателя параллельности запуска или от конвейерной структуры процессора. В некоторых VLIW-конструкциях такое не случалось, и при перемещении между различными моделями процессора требовалась повторная компиляция; в других процессорах со статическим параллельным запуском код в различных реализациях работал бы правильно, но зачастую настолько плохо, что требовалась бы более эффективная компиляция.

Суперскаляр
Усовершенствованная технология конвейеризации, позволяющая процессору выполнять более одной инструкции за тактовый цикл путем выбора инструкций в процессе выполнения.

Динамическая диспетчеризация конвейера
Аппаратная поддержка, предназначенная для изменения порядка выполнения инструкций, чтобы избегать холостой работы.

Многие суперскаляры расширяют базовую структуру решений по динамическому параллельному запуску, чтобы включить **динамическую диспетчеризацию конвейера**. Эта диспетчеризация выбирает, какие именно инструкции будут выполняться в отдельно взятом тактовом цикле, пытаясь избежать конфликтов и холостой работы. Начнем с простого примера устранения конфликта данных. Рассмотрим кодовую последовательность:

```
lw      $t0, 20($s2)
addu    $t1, $t0, $t2
sub     $s4, $s4, $t3
slti    $t5, $s4, 20
```

Даже если инструкция sub готова к выполнению, она сначала должна дождаться завершения инструкции lw и addu, что может при медленной памяти занять множество тактовых циклов. (В главе 5 рассматриваются неудачные обращения кэш-памяти, которые иногда становятся причиной очень медленного обращения к памяти.) Динамическая диспетчеризация конвейера позволяет частично или полностью избегать возникновения подобных конфликтов.

Динамическая диспетчеризация конвейера

При динамической диспетчеризации конвейера происходит выбор следующих выполняемых инструкций с возможной их перестановкой для устранения холостой работы. В таких процессорах конвейер поделен на три основных блока: блок извлечения и запуска инструкций, несколько функциональных блоков (число которых в высокопроизводительных конструкциях 2008 года достигало одного или нескольких десятков) и **блока передачи**. Модель такой конструкции показана на рис. 4.62. Первый блок извлекает инструкции, декодирует их и отправляет каждую инструкцию соответствующему функциональному блоку для ее выполнения. У каждого функционального блока имеются буферы, называемые **резервациями**, хранящие операнды и операции. (В следующем разделе будет рассмотрена альтернатива резервациям, используемая многими современными процессорами.) Как только в буфере будут содержаться все его операнды и функциональный блок будет готов к выполнению инструкции, происходит вычисление результата. Когда будет получен результат, он отправляется в любую резервацию, ожидающую именно этот результат, а также в блок передачи, который сохраняет результат в буфер до тех пор, пока не возникнут условия для безопасного помещения результата в файл регистров или для сохранения его в памяти. Буфер в блоке передачи, который часто называется **буфером восстановления последовательности**, также используется для предоставления операндов, во многом так же, как это делалось логикой препровождения в конвейерах со статической диспетчеризацией. Как только результат отправляется в файл регистров, он должен быть извлечен именно оттуда, как в обычном конвейере.

Блок передачи
Блок в динамическом конвейере, не придерживающемся порядка выполнения инструкций, который решает, когда будет безопасно выпустить результат операции в находящиеся в поле зрения программиста регистры и память.

Резервация
Буфер внутри функционального блока, хранящий операнды и операцию.

Буфер восстановления последовательности
Буфер, хранящий результаты в процессоре с динамической диспетчеризацией до тех пор, пока не появится возможность сохранить эти результаты в памяти или в регистре.

Комбинация буферируемых в резервации операндов и результатов в буфере восстановления последовательности обеспечивает разновидность переименования регистров, почти так же, как это использовалось компилятором в нашем раннем примере по развертыванию цикла. Давайте рассмотрим, как это работает на уровне понятий:

1. При извлечении инструкции она копируется в резервацию соответствующего функционального блока. Также туда немедленно копируются любые операнды, доступные в файле регистров или в буфере восстановления последовательности. Инструкции буферизуются в резервации до тех пор, пока не станут доступны все операнды и функциональный блок. Для запущенной инструкции регистровая копия операнда уже не нужна, и если произойдет запись в этот регистр, значение может быть переписано.
2. Если операнд не находится в файле регистров или в буфере восстановления последовательности, нужно ждать, пока он не будет произведен функциональным блоком. При этом отслеживается имя функционального блока, который произведет результат. Когда в конечном итоге этот блок произведет результат, этот результат будет скопирован непосредственно в ожидающую его резервацию из функционального блока, минуя регистры.

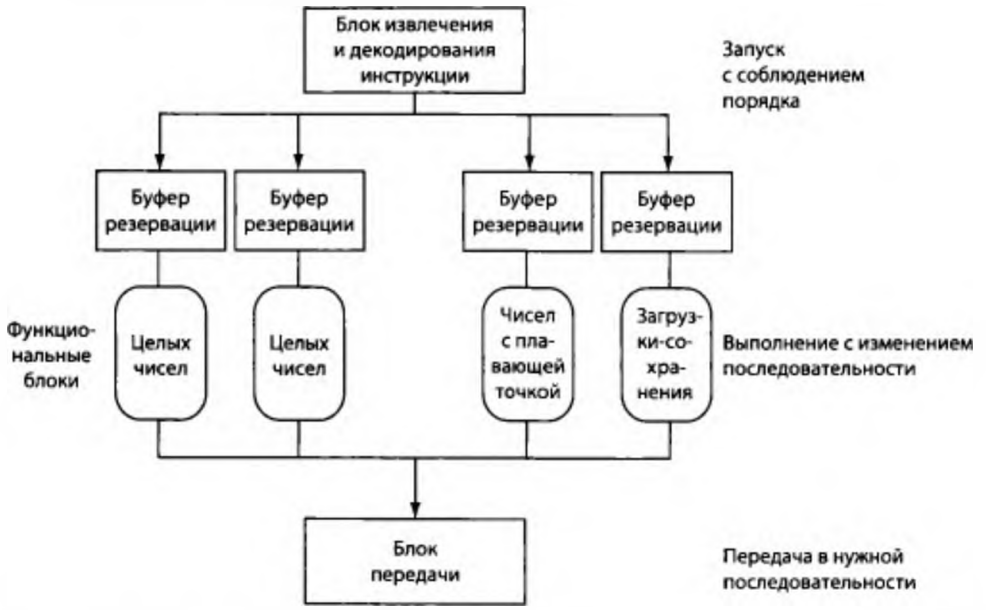


Рис. 4.62. Три основных блока в конвейере с динамической диспетчеризацией. Последний шаг обновления состояния также называется выводом результатов, или выпуском

На этих шагах для реализации переименования регистров эффективно используется буфер восстановления последовательности и резервации.

Концептуально конвейер с динамической диспетчеризацией можно представить анализатором структуры потока данных программы. Затем процессор выполняет инструкции в некотором порядке, сохраняющем структуру потока данных программы. Этот стиль выполнения называется **выполнением с изменением после-**

Выполнение с изменением последовательности
Ситуация в конвейеризированном выполнении, когда инструкция, чье выполнение заблокировано, не становится причиной ожидания для следующих инструкций.

вательности, поскольку инструкции могут быть выполнены не в том порядке, котором были извлечены.

Чтобы заставить программы вести себя так же, как и при их выполнении на простом конвейере без изменения последовательности, блок извлечения и декодирования должен запускать инструкции в том порядке, который позволяет отслеживать зависимости, а блок передачи должен записывать результаты в регистры и в память в порядке извлечения кода программы. Этот консервативный режим называется **передачей в нужном порядке**. Следовательно, если возникнет исключение, компьютер может указать на последнюю выполненную инструкцию, и обновляться будут только те регистры, запись в которые будет вестись инструкциями, предшествующими инструкции, вызвавшей исключение. Хотя на входе (извлечение и запуск) и на выходе (передача) конвейер работает упорядоченно, функциональные блоки вольны инициировать выполнение, как только будут доступны необходимые им данные. В настоящее время все конвейеры с динамической диспетчеризацией используют передачу в нужном порядке.

Динамическая диспетчеризация часто расширяется за счет включения аппаратно реализованных предположений, особенно относительно результатов условных переходов. Прогнозируя направление условного перехода, процессор с динамической диспетчеризацией может продолжить извлечение и выполнение инструкций в соответствии с предсказанным путем. Поскольку результаты выполнения инструкций передаются в нужном порядке, мы узнаем, был ли правильно предсказан условный переход, еще до того, как будет передан результат любой инструкции из предсказанного пути. Конвейер, в котором реализованы предположения и динамическая диспетчеризация, может также поддерживать предположения относительно адресов загрузки, позволяя изменять порядок выполнения инструкций загрузки-сохранения и используя блок передачи для отмены неверных предположений. В следующем разделе будет рассмотрено использование динамической диспетчеризации с предположениями в конструкции микропроцессора AMD Opteron X4 (Barcelona).

Представление о производительности программ

Учитывая, что компиляторы могут также заниматься перепланировкой выполнения кода для обхода зависимостей от данных, можно спросить, зачем суперскалярным процессорам использовать динамическую диспетчеризацию. На то есть три главные причины. Во-первых, не все холостые задержки предсказуемы. В частности, отсутствие нужных данных в кэш-памяти (см. главу 5) приводит к непредсказуемым задержкам. Динамическая диспетчеризация позволяет процессору скрывать некоторые из этих задержек за счет продолжения выполнения инструкций в период ожидания окончания задержки.

Во-вторых, если процессор предполагает, какими будут результаты условных переходов, используя их динамическое прогнозирование, он не может знать конкретный порядок следования инструкций во время компиляции, поскольку он зависит от предсказанного и реаль-

Передача в нужном порядке
Передача, при которой результаты конвейеризированного выполнения записываются в структуры, видимые программисту в том же порядке, в каком извлекались инструкции.

ного поведения условных переходов. Включение динамических предположений для достижения более высокого параллелизма на уровне инструкций (ILP) без включения динамической диспетчеризации существенно ограничит преимущества, получаемые от выдвижения предположений.

В-третьих, поскольку задержка конвейера и ширина запуска инструкций от реализации к реализации изменяются, лучший способ компиляции кодовой последовательности также изменяется. Например, как спланировать кодовую последовательность, состоящую из взаимозависимых инструкций, с учетом влияния как ширины запуска, так и задержки конвейера? Структура конвейера влияет и на количество развертываний цикла с целью устранения задержек, и на процесс по переименованию регистров, проводимый компилятором. Динамическая диспетчеризация позволяет оборудованию скрывать большинство таких подробностей. Таким образом, пользователи и распространители программного обеспечения не должны волноваться о том, что им понадобятся несколько версий программы для различных реализаций одного и того же набора инструкций. Наряду с этим старый унаследованный код получит от новой реализации наибольшие преимущества без необходимости его перекомпиляции.

Общее представление

Конвейеризация и выполнение с параллельным запуском инструкций повышают пиковую пропускную способность инструкций и пытаются использовать параллелизм на уровне инструкций (ILP). Но взаимозависимости, возникающие при доступе к данным и при управлении процессом выполнения программы, назначают верхний предел устойчивой производительности, поскольку процессору приходится иногда ожидать разрешения взаимозависимостей. Подходы, основанные главным образом на программных методах использования ILP, базируются на способности компилятора обнаруживать и сокращать влияние подобных взаимозависимостей, а подходы, основанные главным образом на работе оборудования, зависят от расширений конвейера и механизмов запуска инструкций. Предположения, выдаваемые компилятором или оборудованием, могут повысить уровень потенциальной возможности использования ILP, хотя их нужно применять с оглядкой, поскольку неверные предположения чаще всего приводят к снижению производительности.

Интерфейс аппаратного и программного обеспечения

Современные высокопроизводительные микропроцессоры способны запускать сразу несколько инструкций в один тактовый цикл; к сожалению, обеспечение этого показателя запуска инструкций является очень сложной задачей. Например, несмотря на существование процессоров, параллельно запускающих от четырех до шести инструкций в один тактовый цикл, весьма немногие приложения могут поддержать параллельный запуск более двух инструкций за один такт. Это объясняется двумя основными причинами.

Во-первых, внутри конвейера основные узкие места производительности возникают из-за зависимостей, которые не могут быть устранены, снижая, таким образом, параллелизм инструкций и поддерживаемую норму их запуска. Хотя с настоящими зависимостями от данных справиться довольно трудно, компилятор или оборудование не знают точно, существует такая зависимость или нет, и поэтому должны консервативно считать зависимость существующей. Например, код, в котором используются указатели, особенно такими способами, которые могут привести к использованию псевдонимов, приведет к более скрытым потенциальным зависимостям. И наоборот, более систематическое обращение к массивам позволяет компилятору делать вывод об отсутствии зависимостей. Точно так же условные переходы, которые не могут быть точно предсказаны в процессе выполнения программы или в ходе ее компиляции, ограничат возможности использования ILP. Часто имеется возможность дополнительного увеличения уровня ILP, но возможности компилятора или оборудования по поиску ILP, который мог бы в значительной степени быть обособлен (иногда путем выполнения нескольких тысяч инструкций), ограниченны.

Во-вторых, неудачные обращения к системам памяти (которые раскрываются в главе 5) также ограничивают возможности по поддержанию полной загруженности конвейера. Некоторые задержки, связанные с системой памяти, могут быть скрыты, но ограничения уровня ILP также ограничивают и пространства, в которых могут быть скрыты подобные задержки.

Энергоэффективность и усовершенствованная конвейеризация

Отрицательным фактором повышения использования параллелизма на уровне инструкций за счет динамического параллельного запуска и выстраивания предположений является снижение энергоэффективности. Каждое нововведение могло привлекать к работе большее количество транзисторов, что приводило к существенному снижению энергоэффективности. И сейчас, когда мы уперлись в стену теплового барьера, рассматриваются конструкции, содержащие сразу несколько процессоров на одном кристалле, где процессоры не настолько глубоко конвейеризированы и не настолько агрессивны в плане выстраивания предположений, как их предшественники.

Существует представление о том, что более простые процессоры не настолько быстры, как их более сложные собратья, но при этом они выдают лучший показатель производительности в пересчете на ватт потребляемой энергии, поэтому они могут выдать более высокую производительность на один кристалл, когда конструкции испытывают ограничения, связанные больше с выделяемой тепловой мощностью, чем с количеством транзисторов.

В табл. 4.11 показано количество стадий конвейера, ширина запуска, уровень выстраивания предположений, тактовая частота, количество ядер на кристалл и потребляемая мощность некоторых микропроцессоров. Обратите внимание на

снижение количества стадий конвейера и потребляемой мощности при переходе компаний на многоядерные конструкции.

Таблица 4.11. Данные компаний Intel и Sun Microprocessors. В стадии конвейера Pentium 4 не включены стадии передачи. Если они были бы включены, то конвейеры Pentium 4 стали бы еще глубже

Микропроцессор	Год	Тактовая частота	Количество стадий конвейера	Ширина запуска	Изменение последовательности и предположения	Количество ядер или чипов	Мощность
Intel 486	1989	25 МГц	5	1	Нет	1	5 Вт
Intel Pentium	1993	66 МГц	5	2	Нет	1	10 Вт
Intel Pentium Pro	1997	200 МГц	10	3	Да	1	29 Вт
Intel Pentium 4 Willamette	2001	2000 МГц	22	3	Да	1	75 Вт
Intel Pentium 4 Prescott	2004	3600 МГц	31	3	Да	1	103 Вт
Intel Core	2006	2930 МГц	14	4	Да	2	75 Вт
Sun UltraSPARC III	2003	1950 МГц	14	4	Нет	1	90 Вт
Sun UltraSPARC T1 (Niagara)	2005	1200 МГц	6	1	Нет	8	70 Вт

Уточнение. Блок передачи управляет обновлениями файла регистров *и* памяти. Некоторые процессоры с динамической диспетчеризацией обновляют файл регистров немедленно в ходе выполнения инструкции, используя дополнительные регистры для реализации функции переименования и сохраняя старую копию регистра до тех пор, пока инструкция, обновляющая регистр, больше не относится к числу предполагаемых к выполнению. Другие процессоры помещают результат в буфер, обычно в структуру, которая называется буфером восстановления последовательности, и настоящее обновление файла регистров происходит позже, являясь частью передачи. Данные, сохраняемые в памяти, должны храниться в буфере до тех пор, пока не наступит время передачи, причем это может быть либо *буфер сохранения* (см. главу 5), либо буфер восстановления последовательности. Блок передачи позволяет инструкции сохранения вести запись в память из буфера, когда у буфера имеются нужный адрес и нужные данные и когда инструкция сохранения больше уже не зависит от спрогнозированных условных переходов.

Уточнение. Обращение к памяти извлекает пользу от *неблокируемых кэшей*, которые продолжают обслуживание обращения к кэш-памяти в период отсутствия в кэше нужных данных (см. главу 5). Процессоры с изменением последовательности выполнения инструкций нуждаются в таком устройстве кэш-памяти, которое позволяет инструкциям выполняться во время отсутствия в ней нужных данных.

Самопроверка

Определите, имеют ли следующие технологии или компоненты отношение в основном к программным или к аппаратным подходам использования ILP. В некоторых случаях ответ может относиться к обоим подходам.

1. Прогнозирование условного перехода.
2. Параллельный запуск инструкций.
3. Слово инструкций очень большой длины – VLIW.
4. Суперскаляр
5. Динамическая диспетчеризация.
6. Выполнение с изменением последовательности инструкций.
7. Выстраивание предположений.
8. Буфер восстановления последовательности.
9. Переименование регистров.

4.11. Реальное оборудование: конвейер AMD Opteron X4 (Barcelona)

Как и в большинстве современных компьютеров, в микропроцессорах x86 используются довольно сложные подходы к конвейеризации. Но эти процессоры по-прежнему сталкиваются с трудностями реализации сложного набора инструкций x86, рассмотренными в главе 2. Изделия компаний AMD и Intel извлекают инструкции x86 и транслируют их во внутренние MIPS-подобные инструкции, которые компания AMD называет *RISC-операциями* (*Rops*), а компания Intel — *микрооперациями*. Затем RISC-операции выполняются современным конвейером с динамической диспетчеризацией и с выстраиванием предположений, способным обеспечить в AMD Opteron X4 (Barcelona) норму выполнения, равную трем RISC-операциям за один тактовый цикл. Этот раздел посвящен конвейеру RISC-операций.

Когда мы рассматриваем сложные процессоры с динамической диспетчеризацией, конструкция системы управления функциональными блоками, кэшем и файлом регистров, запуском инструкций и конвейером в целом становится очень запутанной, затрудняя отделение операционного блока от конвейера. По этой причине многие инженеры и исследователи для ссылки на подробную внутреннюю **архитектуру** взяли на вооружение термин **микроархитектура**. На рис. 4.63 показана микроархитектура X4, сфокусированная на структурах для выполнения RISC-операций.

Другой взгляд на X4 заключается в рассмотрении стадий конвейера, через которые проходит типовая инструкция. На рис. 4.64 показана структура конвейера и типичное количество затрачиваемых тактовых циклов, которое,

Микроархитектура
Организация процессора, включая основные функциональные блоки, их взаимосвязи и управление.

конечно же, варьируется сообразно природе динамической диспетчеризации, а также согласно требованиям отдельных RISC-операций.

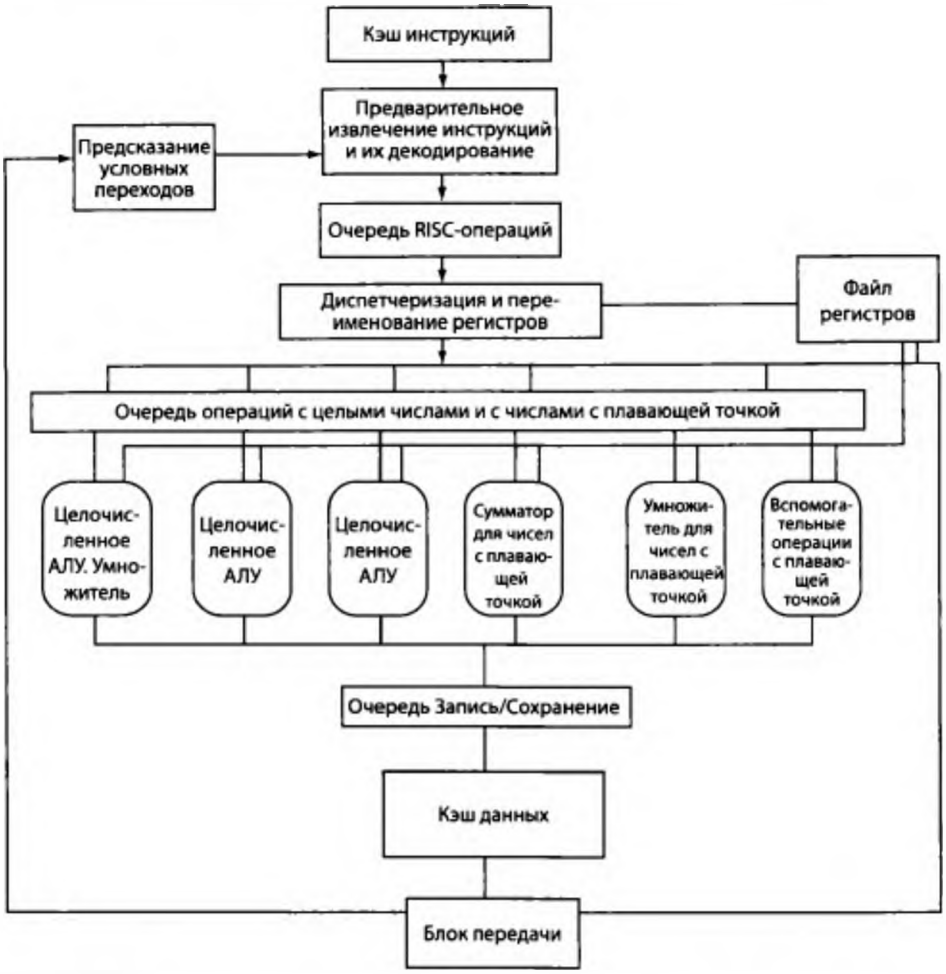


Рис. 4.63. Микроархитектура AMD Opteron X4. Большие по емкости очереди позволяют оставлять невыполненными до 106 RISC-операций, включая до 24 целочисленных операций, до 36 операций с плавающей точкой, или SSE-операций, и до 44 операций загрузки и сохранения. Блоки загрузки и сохранения в действительности разделены на две части, первая из которых занимается вычислением адреса в целочисленных блоках АЛУ, а вторая отвечает непосредственно за обращение к памяти. Между функциональными блоками существует разветвленная сеть параллельных соединений; поскольку конвейер имеет динамическую, а не статическую природу, обходы осуществляются за счет маркировки (тегирования) результатов и отслеживания исходных операндов, позволяющих получать соответствие в том случае, когда результат произведен для инструкции в одной из нуждающихся в этом результате очередей

Рис. 4.64. Изображение на конвейере Opteron X4 продвижения типовой инструкции и количества тактовых циклов для основных шагов в конвейере, имеющем 12 стадий для целочисленных RISC-операций. Очередь выполнения инструкций для чисел с плавающей точкой состоит из 17 стадий. На рисунке также показаны буферы, где ожидают своего выполнения RISC-операции

Уточнение. В Opteron X4 используется схема для разрешения антизависимостей и неверных предположений, которая использует буфер восстановления последовательности вместе с переименованием регистров. Это переименование имеет прямое отношение к переименованию **архитектурных регистров**, имеющихся в процессоре (16 в случае использования 64-разрядной версии архитектуры x86) в большое количество физических регистров (72 в X4). Переименование регистров используется в X4 для устранения антизависимостей. Оно требует от процессора поддержки отображения между архитектурными и физическими регистрами, показывающей, какой из физических регистров является наиболее актуальной копией архитектурного регистра. За счет отслеживания происходящих переименований переименование регистров предлагает еще один подход восстановления в случае неверного предположения: простой откат отображений, происшедших со времени начала выполнения инструкции, в отношении которой было сделано неверное предположение. Это заставит состояние процессора вернуться к последней правильно выполненной инструкции, сохраняя при этом правильное отображение между архитектурными и физическими регистрами.

Самопроверка

Верны ли следующие утверждения:

1. Конвейер Opteron X4, обладающий свойством параллельного запуска, выполняет непосредственно x86-инструкции.
2. В X4 используется динамическая диспетчеризация, но без выстраивания предположений.
3. В микроархитектуре X4 имеется намного больше регистров, чем требует x86.
4. В X4 используется менее половины стадий конвейеров ранее выпускавшегося микропроцессора Pentium 4 Prescott (см. табл. 4.11).

Представление о производительности программ

В Opteron X4 для достижения высокой производительности сочетаются конвейер, состоящий из 12 стадий, и агрессивный параллельный запуск. За счет поддержания низких значений периодов ожидания следующих друг за другом операций сокращается влияние зависимостей от данных. Каковы наиболее важные

Архитектурные регистры
Регистры процессора, видимые в наборе инструкций; например, в MIPS существует 32 целочисленных регистра и 16 регистров с плавающей точкой.

потенциально узкие места производительности программ, запущенных на этом процессоре? В следующий список включены некоторые потенциальные проблемы производительности, включая и последние три, которые могут быть отнесены в той или иной форме к любому высокопроизводительному процессору, имеющему конвейер.

- Использование x86-инструкций, которые не могут быть отображены на несколько простых RISC-операций.
- Условные переходы, трудно поддающиеся прогнозированию, становящиеся причиной задержек из-за неверных предсказаний и перезапуска при несостоявшихся предположениях.
- Протяженные взаимозависимости, которые обычно вызываются наличием долго работающих инструкций или отсутствием нужных данных в кэше и приводят к задержкам.
- Холостая работа, возникающая при обращениях к памяти (см. главу 5) и приводящая к задержкам в работе процессора.

4.12. Заблуждения и недоразумения

Заблуждение. *Конвейеризация — это просто.*

Наши книги свидетельствуют о том, что правильное конвейерное выполнение является настоящим искусством. В первом издании нашей расширенной книги в описании конвейера была допущена ошибка, несмотря на то что ее просмотрело более ста человек и ее качество было проверено в 18 университетах. Ошибка была выявлена, только когда была предпринята попытка создать компьютер по описаниям, представленным в книге. Тот факт, что описание конвейера, подобного тому, который имеется в Opteron X4, будет занимать на языке Verilog тысячи строк, свидетельствует о его сложности. Поэтому нужно быть осторожнее в своих оценках!

Заблуждение. *Идеи конвейеризации могут быть реализованы независимо от технологии.*

Когда наилучшим решением благодаря количеству транзисторов на одном кристалле и их быстродействию стал конвейер, имеющий пять стадий, то простым разрешением конфликтов управления был отложенный условный переход (см. первое «Уточнение» в разделе «Динамическое прогнозирование условного перехода»). Теперь уже избыточными являются более длинные конвейеры, суперскалярное выполнение и динамическое прогнозирование условного перехода. В начале 1990-х динамическая диспетчеризация конвейера отнимала слишком много ресурсов и для достижения высокой производительности не требовалась, но по мере того как удвоение количества транзисторов продолжалось, а быстродействие логических схем существенно опережало быстродействие памяти, возрос и смысл использования нескольких функциональных блоков и динамической конвейеризации. Сегодня озабоченность, связанная с ростом энергопотребления, привела к появлению менее агрессивных конструкций.

Недоразумение. *Недооценка особенностей набора инструкций может отрицательно сказаться на конвейеризации.*

Многие трудности конвейеризации возникают из-за сложностей устройства набора инструкций. Приведем ряд примеров:

- Широко варьируемые длины инструкций и периоды их выполнения могут привести к диспропорции стадий конвейера и к существенному усложнению обнаружения конфликтов в конструкции, где конвейеризация осуществляется на уровне набора инструкций. Эта проблема была преодолена сначала на машине DEC VAX 8500 в конце 1980-х за счет использования схемы с микроконвейером, которая сегодня используется на Opteron X4. Разумеется, остаются издержки трансляции и обеспечения соответствия между микрооперациями и фактическими инструкциями.
- Сложные режимы адресации могут стать причиной возникновения различных проблем. Режимы адресации, обновляющие значения регистров усложняют обнаружение конфликтов. Другие режимы адресации, которые требуют множественного обращения к памяти, существенно усложняют управление конвейером и затрудняют его равномерную работу.

Наверное, лучшим примером могут послужить компьютеры DEC Alpha и DEC NVAX. При сопоставимой технологии более новая архитектура набора инструкций компьютера Alpha позволяет создать реализацию, которая более чем вдвое производительнее компьютера NVAX. Другой пример, приводимый Бхандаркаром (Bhandarkar) и Кларком (Clark), касается сравнения компьютеров MIPS M/2000 и DEC VAX 8700 путем подсчета тактовых циклов при выполнении контрольных задач SPEC; они пришли к выводу, что, несмотря на то что компьютер MIPS M/2000 выполняет больше инструкций, компьютер VAX в среднем выполняет в 2,7 раза больше тактовых циклов, следовательно, компьютер MIPS работает быстрее.

4.13. Заключительные комментарии

> Девять десятых мудрости заключается в том, чтобы проявить эту мудрость своевременно.
>
> *Американская поговорка*

Как было показано в данной главе, и операционный блок, и система управления процессора могут строиться на базе архитектуры набора инструкций и осмысления базовых характеристик технологии. В разделе 4.3 было показано, как операционный блок для процессора MIPS может быть сконструирован на основе архитектуры и решения о построении однотактной реализации. Разумеется, положенная в основу технология также оказывает влияние на многие конструкторские решения путем нанизывания состава компонентов, которые могут использоваться в операционном блоке, а также путем постановки вопроса о целесообразности применения однотактной реализации.

Конвейеризация улучшает пропускную способность, но не присущее той или иной инструкции время выполнения, или **латентность инструкции**; некоторые инструкции имеют такой же показатель латентности, как и при однотактном подходе. Параллельный запуск инструкций влечет за собой применение дополнительного оборудования операционного блока, позволяющего при каждом тактовом цикле приступать к выполнению сразу нескольких инструкций, но делать это за счет увеличения эффективного показателя латентности. Конвейеризация была призвана сократить продолжительность тактового цикла простого однотактного операционного блока. По сравнению с этим параллельный запуск инструкций явно сфокусирован на задаче сокращения количества тактовых циклов, приходящихся на одну инструкцию (CPI).

Как конвейеризация, так и параллельный запуск являются попыткой использования параллелизма на уровне инструкций. Наличие взаимозависимостей данных и управляющих структур, которые могут превратиться в конфликты, становится главным фактором ограничения степени используемого параллелизма. Диспетчеризация и выстраивание предположений как в аппаратной, так и в программной сфере являются первостепенными технологиями, используемыми для ослабления влияния взаимозависимостей на производительность.

Переход на более длинные конвейеры, параллельный запуск инструкций и динамическую диспетчеризацию, состоявшийся в середине 1990-х годов, помог выдержать 60% ежегодный прирост производительности процессоров, взявший старт в начале 1980-х годов. Как уже упоминалось в главе 1, микропроцессоры придерживались последовательной модели программирования, но со временем они столкнулись с тепловым барьером. Поэтому промышленность была вынуждена разработать многопроцессорные системы, использующие параллелизм на более жестких уровнях (эта тема будет рассмотрена в главе 7). Эта тенденция также заставила разработчиков пересмотреть с середины 1990-х годов значение некоторых изобретений, касающихся потребляемой мощности и производительности, что выразилось в упрощении конвейеров в самых последних версиях микроархитектур.

Чтобы поддержать улучшения в производительности обработки данных с помощью параллельных процессоров, закон Амдала намекает на то, что теперь узким местом системы станет другая ее часть. Это узкое место, а именно система памяти, и станет темой следующей главы.

4.14. Упражнения

Предоставлены Милошем Првуловичем (Milos Prvulovic) из Технологического института Джорджии.

Упражнение 4.1

Латентность инструкции
Время выполнения, присущее той или иной инструкции.

В базовой однотактной реализации разными инструкциями используются разные аппаратные блоки. Следующие три задачи в этом упражнении используют инструкции:

	Инструкция	Интерпретация
а	add Rd,Rs,Rt	Per[Rd]=Per[Rs]+Per[Rt]
б	lw Rt,Offs(Rs)	Per[Rt]=Память[Per[Rs]+Смещ]

4.1.1 [5] <4.1> Какими будут для этих инструкций значения сигналов управления, генерируемых блоком управления, показанном на рис. 4.2?

4.1.2 [5] <4.1> Какие ресурсы (блоки) выполняют полезную функцию для данной инструкции?

4.1.3 [10] <4.1> Какие ресурсы (блоки) производят данные на выходе, которые используются для этих инструкций? Какие ресурсы не производят данных на выходе для этой инструкции?

Разные блоки выполнения и блоки цифровой логики имеют разные показатели латентности (времени, необходимого им для выполнения работы). На рис. 4.2 показано семь разновидностей основных блоков. Латентности блоков согласно критическому (самому продолжительному) пути для инструкции определяют минимальную латентность этой инструкции. Для остальных трех задач этого упражнения предположим следующие показатели латентности ресурсов:

	Память инструкций (I-Mem)	Сумматор (Add)	Мультиплексор (Mux)	АЛУ (ALU)	Файл регистров (Regs)	Память данных (D-Mem)	Блок управления (Control)
а	400 пс	100 пс	30 пс	120 пс	200 пс	350 пс	100 пс
б	500 пс	150 пс	100 пс	180 пс	220 пс	1000 пс	65 пс

4.1.4 [5] <4.1> Каков критический путь для MIPS-инструкции AND?

4.1.5 [5] <4.1> Каков критический путь для MIPS-инструкции загрузки (LD)?

4.1.6 [10] <4.1> Каков критический путь для MIPS-инструкции BEQ?

Упражнение 4.2

Базовая однотактная MIPS-реализация на рис. 4.2 способна выполнять лишь некоторые инструкции. К существующей архитектуре набора инструкций могут быть добавлены новые инструкции, но решение о целесообразности таких действий, кроме всего прочего, зависит от стоимости и сложности подобных добавлений, вносимых в операционный блок и в блок управления процессора. Первые три задачи данного упражнения ссылаются на следующую новую инструкцию:

	Инструкция	Интерпретация
а	add3 Rd Rs,Rt,Rx	Per[Rd]=Per[Rs]+Per[Rt]+Per[Rx]
б	sll Rt,Rd,Shift	Per[Rd]= Per[Rt] << Shift (сдвиг влево на Shift разрядов)

4.2.1 [10] <4.1> Какие из имеющихся блоков (если есть таковые) могут быть использованы для этой инструкции?

4.2.2 [10] <4.1> Какие новые функциональные блоки (если в них есть потребность) нужны для этой инструкции?

4.2.3 [10] <4.1> Какие новые сигналы нужны (и нужны ли они) от блока управления для поддержки этой инструкции?

Когда разработчики процессора рассматривают возможное усовершенствование операционного блока процессора, решения обычно зависят от соотношения стоимости и производительности. В следующих трех задачах предполагается, что мы начинаем с использования операционного блока, изображенного на рис. 4.2, где блоки памяти инструкций (I-Mem), сумматора (Add), мультиплексора (Mux), АЛУ (ALU), файла регистров (Regs), памяти данных (D-Mem) и управления (Control) имеют показатели латентности, равные соответственно 400 пс, 100 пс, 30 пс, 120 пс, 200 пс, 350 пс и 100пс, а показатели стоимости 1000, 30, 10, 100, 200, 2000 и 500 соответственно. Остальные три задачи этого упражнения ссылаются на следующее усовершенствование процессора:

	Усовершенствование	Латентность	Стоимость	Преимущество
а	Более быстродействующий сумматор	–20 пс для блоков сумматоров	+20 для блоков сумматоров	Замена существующих блоков сумматоров более быстродействующими
б	Расширенный файл регистров	+100 пс для файла регистров	+200 для файла регистров	Для сохранения и восстановления значений регистров понадобится меньшее количество загрузок и сохранений. Это выразится в пятипроцентном сокращении количества инструкций

4.2.4 [10] <4.1> Какой будет продолжительность тактового цикла при использовании данного усовершенствования и без него?

4.2.5 [10] <4.1> Какое ускорение будет достигнуто при добавлении этого усовершенствования?

4.2.6 [10] <4.1> Сравните соотношение стоимости и производительности при использовании данного усовершенствования и без него.

Упражнение 4.3

Задачи данного упражнения ссылаются на следующий логический блок:

	Логический блок
а	Небольшая память инструкций из четырех восьмиразрядных слов
б	Небольшой блок файла регистров из двух восьмиразрядных регистров

4.3.1 [5] <4.1, 4.2> Содержит ли этот логический блок одни только логические схемы, одни только триггеры или имеет и то и другое?

4.3.2 [20] <4.1, 4.2> Покажите, как может быть реализован этот блок. Используйте только И, ИЛИ, НЕ и D-элементы.

4.3.3 [10] <4.1, 4.2> Выполните упражнение 4.3.2 еще раз, но все используемые вами логические элементы И и ИЛИ должны иметь два входа.

Стоимость и латентность цифровой логики зависят от типов основных доступных логических элементов и свойств этих элементов. Остальные три задачи этого упражнения ссылаются на следующие логические элементы, показатели латентности и стоимости:

	НЕ		Двухвходовый ИЛИ и И		Каждый дополнительный вход для И/ИЛИ		D-элемент	
	Латентность	Стоимость	Латентность	Стоимость	Латентность	Стоимость	Латентность	Стоимость
а	20 пс	1	30 пс	2	+0 пс	+1	40 пс	6
б	50 пс	1	100 пс	2	+40 пс	+1	160 пс	2

4.3.4 [5] <4.1, 4.2> Какова латентность вашей реализации из упражнения 4.3.2?

4.3.5 [5] <4.1, 4.2> Какова стоимость вашей реализации из упражнения 4.3.2?

4.3.6 [20] <4.1, 4.2> Внесите изменения в вашу конструкцию для минимизации латентности, а затем для минимизации стоимости. Сравните стоимость и латентность этих двух оптимизированных конструкций.

Упражнение 4.4

При реализации логического выражения в цифровой логике нужно использовать доступные логические элементы, чтобы реализовать оператор, для которого не приспособлен логический элемент. Задачи в данном упражнении ссылаются на следующие логические выражения:

	Сигнал управления 1	Сигнал управления 2
а	(((A ИЛИ B) ИЛИ C) ИЛИ (A И C)) ИЛИ (A И B)	(A ИЛИ B) ИЛИ C
б	(((A OR B) исключающее ИЛИ B) ИЛИ (A ИЛИ C)) ИЛИ (A И B)	A И B

4.4.1 [5] <4.2> Создайте логическую схему для сигнала управления 1. Ваша схема должна непосредственно реализовать заданное выражение (не следует заниматься реорганизацией выражения с целью его «оптимизации»), используя логические элементы НЕ и двухвходовые логические элементы И, ИЛИ и исключающее ИЛИ.

4.4.2 [10] Если предположить, что у всех логических элементов одинаковая латентность, то какой будет длина (в элементах) критического пути в вашей схеме из упражнения 4.4.1?

4.4.3 [10] <4.2> При реализации нескольких логических выражений можно сократить стоимость за счет использования одних и тех же сигналов в более чем одном выражении. Выполните упражнение 4.4.1 еще раз, но теперь реализуйте оба сигнала управления, 1 и 2, и попытайтесь задействовать «общие» цепи между

выражениями там, где это возможно. Для остальных трех задач этого упражнения представим, что нам доступны показанные ниже основные элементы цифровой логики, обладающие следующими показателями латентности и стоимости:

	НЕ		Двухвходовое И		Двухвходовое ИЛИ		Двухвходовое исключающее ИЛИ	
	Латентность	Стоимость	Латентность	Стоимость	Латентность	Стоимость	Латентность	Стоимость
а	20 пс	1	30 пс	2	34 пс	3	40 пс	6
б	50 пс	1	100 пс	2	120 пс	2	150 пс	2

4.4.4 [10] <4.2> Какова длина критического пути в вашей схеме из упражнения 4.4.3?

4.4.5 [10] <4.2> Какова стоимость вашей схемы из упражнения 4.4.3?

4.4.6 [10] <4.2> Какова доля стоимости, сэкономленная в вашей схеме из упражнения 4.4.3, за счет совместной, а не раздельной реализации этих двух сигналов управления?

Упражнение 4.5

Цель данного упражнения состоит в том, чтобы помочь вам ознакомиться с конструкцией и работой последовательных логических схем. Задачи в данном упражнении ссылаются на следующую операцию АЛУ:

	Операция АЛУ
а	Прибавление единицы (X+1)
б	Сдвиг влево на два разряда (X<<2)

4.5.1 [20] <4.2> Составьте схему с одноразрядными входами данных и одноразрядным выходом данных, которая выполняла бы эту операцию последовательно, начиная с самого младшего разряда. В последовательной реализации схема обрабатывает входные операнды поразрядно, генерируя выходные разряды друг за другом. Например, последовательная схема И представляет собой простой логический элемент И, в цикле N мы даем ему N-ный разряд из каждого операнда и получаем N-ный разряд результата. Кроме входов данных в схеме имеется вход Clk (тактовый импульс) и вход «Start», на котором выставляется 1 только при самом первом цикле операции. В вашей конструкции можно использовать D-элементы и логические элементы НЕ, И, ИЛИ и Исключающее ИЛИ.

4.5.2 [20] <4.2> Выполните упражнение 4.5.1 еще раз, но теперь составьте схему, выполняющую эту операцию сразу для двух разрядов.

Для остальной части данного упражнения представим, что нам доступны показанные ниже основные элементы цифровой логики со следующими показателями латентности и стоимости:

	НЕ		И		ИЛИ		Исключающее ИЛИ		D-элемент	
	Латентность	Стоимость	Латентность	Стоимость	Латентность	Стоимость	Латентность	Стоимость	Латентность	Стоимость
а	20 пс	1	30 пс	2	20 пс	2	30 пс	4	40 пс	6
б	40 пс	1	50 пс	2	60 пс	2	80 пс	3	80 пс	12

Время, указанное для D-элемента, является временем его установки. Вход данных триггера должен иметь нужное единичное значение в установочное время до прохождения фронта синхроимпульса (окончания тактового цикла), что позволит сохранить это значение в триггере.

4.5.3 [10] <4.2> Каким будет продолжительность тактового цикла для составленной вами схемы в упражнении 4.5.1? Сколько времени займет выполнение 32-разрядной операции?

4.5.4 [10] <4.2> Каким будет продолжительность тактового цикла для составленной вами схемы в упражнении 4.5.2? Какое ускорение будет получено за счет использования этой схемы вместо схемы, составленной в упражнении 4.5.1 для 32-разрядной операции?

4.5.5 [10] <4.2> Вычислите стоимость для схемы, составленной в упражнении 4.5.1, а затем для схемы, составленной в упражнении 4.5.2.

4.5.6 [5] <4.2> Сравните соотношение стоимости и производительности для двух схем, составленных в упражнениях 4.5.1 и 4.5.2. Для этих задач производительность схемы является величиной, обратной времени, которое необходимо для проведения 32-разрядной операции.

Упражнение 4.6

Задачи в этом упражнении предполагают, что логические блоки, необходимые для реализации операционного блока процессора, имеют следующие показатели латентности:

	Память инструкций (I-Mem)	Сумматор (Add)	Мультиплексор (Mux)	АЛУ (ALU)	Файл регистров (Regs)	Память данных (D-Mem)	Расширитель знака (Sign-extend)	Блок сдвига влево на два (Shift-left-2)
а	400 пс	100 пс	30 пс	120 пс	200 пс	350 пс	20 пс	2 пс
б	500 пс	150 пс	100 пс	180 пс	220 пс	1000 пс	90 пс	20 пс

4.6.1 [10] <4.3> Каким должен быть период тактового цикла, если в процессоре нужно лишь извлекать идущие друг за другом инструкции (см. рис. 4.6)?

4.6.2 [10] <4.3> Рассмотрим операционный блок, подобный показанному на рис. 4.11, но для процессора, который имеет только один тип инструкций:

безусловный переход, завязанный на значение счетчика команд (PC). Каким будет продолжительность тактового цикла для такого операционного блока?

4.6.3 [10] <4.3> Повторите упражнение 4.6.2, но теперь с необходимостью поддержки только *условных* переходов, завязанных на значение счетчика команд (PC).

Остальные три задачи данного упражнения ссылаются на следующий логический блок (ресурс) операционного блока:

	Ресурс
а	Прибавление 4 (к значению PC)
б	Память данных

4.6.4 [10] <4.3> Для какого типа инструкций требуется данный ресурс?

4.6.5 [20] <4.3> Для какого типа инструкций (если таковой имеется) этот ресурс находится на критическом пути?

4.6.6 [10] <4.3> Предположим, что обеспечивается поддержка только лишь инструкций `beq` и `add`. Дайте объяснение тому, как изменения в заданной латентности этих ресурсов влияют на продолжительность тактового цикла процессора, при условии, что показатели латентности других ресурсов не изменяются.

Упражнение 4.7

В данном упражнении мы изучим, как латентность отдельных компонентов операционного блока влияет на продолжительность тактового цикла всего операционного блока и как эти компоненты используются инструкциями. Для задач этого упражнения будут использоваться следующие показатели латентности логических схем операционного блока:

	Память инструкций (I-Mem)	Сумматор (Add)	Мультиплексор (Mux)	АЛУ (ALU)	Файл регистров (Regs)	Память данных (D-Mem)	Расширитель знака (Sign-extend)	Блок сдвига влево на два (Shift-left-2)
а	400 пс	100 пс	30 пс	120 пс	200 пс	350 пс	20 пс	0 пс
б	500 пс	150 пс	100 пс	180 пс	220 пс	1000 пс	90 пс	20 пс

4.7.1 [10] <4.3> Какой будет продолжительность тактового цикла, если нужно будет поддерживать только один тип инструкций, связанный с работой АЛУ (`add`, `and`, и т.д.)?

4.7.2 [10] <4.3> Какой будет продолжительность тактового цикла, если нужно будет поддерживать только инструкции `lw`?

4.7.3 [20] <4.3> Какой будет продолжительность тактового цикла, если нужно будет поддерживать инструкции `add`, `beq`, `lw` и `sw`?

Для остальных задач этого упражнения предположим, что задержки конвейера отсутствуют, а выполняемые инструкции распределяются следующим образом:

	add	addi	not	beq	lw	sw
a	30%	15%	5%	20%	20%	10%
б	25%	5%	5%	15%	35%	15%

4.7.4 [10] <4.3> В какой доле всех циклов используется память данных?

4.7.5 [10] <4.3> В какой доле всех циклов необходимо использовать ввод в схему расширения знака? Что делает эта схема в циклах, в которых ее ввод не нужен?

4.7.6 [10] <4.3> Если можно улучшить показатель латентности одного из заданных компонентов операционного блока на 10%, то какой из компонентов нужно выбрать? Каким будет ускорение работы от этого улучшения?

Упражнение 4.8

При производстве кремниевых микросхем дефекты в исходных материалах (например, в кремнии) и производственные ошибки могут в итоге привести к созданию дефектных схем. Весьма распространенным является дефект, при котором один проводник оказывает влияние на появление сигнала в другом проводнике. Это называется перекрестной наводкой. К специальному классу перекрестной наводки относится тот случай, когда сигнал связан с проводником, на котором выставлено постоянное логическое значение (например, с питающим проводником). В таком случае мы имеем дело с ошибкой типа постоянного нуля или постоянной единицы, и находящийся под влиянием сигнал имеет соответственно логическое значение 0 или 1.

Следующие задачи ссылаются на показанный ниже сигнал с рис. 4.21:

	Сигнал
а	Память инструкций, выход Инструкция, разряд 7
б	Блок управления, выход MemtoReg

4.8.1 [10] <4.3, 4.4> Предположим, что тестирование процессора произведено путем заполнения счетчика команд (PC), регистров и памяти данных и инструкций некими значениями (по вашему выбору), при этом было позволено выполниться отдельной инструкции, затем произведено чтение счетчика команд, блоков памяти и регистров. Затем эти значения были проверены для обнаружения наличия специфической ошибки. Можете ли вы разработать тест (придумать значения для PC, блоков памяти и регистров), который определил бы наличие ошибки постоянного нуля на том или ином сигнале?

4.8.2 [10] <4.3, 4.4> Повторите упражнение 4.8.1 для ошибки постоянной единицы. Можете ли вы использовать единый тест для обеих ошибок постоянного нуля и постоянной единицы? Если да, то объясните как, если нет, то объясните почему.

4.8.3 [60] <4.3, 4.4> Если нам известно, что процессор имеет для того или иного сигнала ошибку постоянной единицы, может ли процессор по-прежнему считаться пригодным к использованию? Чтобы он был пригоден, нам нужно иметь

возможность конвертировать любую программу, выполняемую на нормальном MIPS-процессоре, в программу, которая работает на этом процессоре. Можно предположить наличие достаточного объема свободной памяти инструкций и памяти данных, позволяющего сделать программу длиннее и сохранять дополнительные данные. *Подсказка*: процессор пригоден к использованию, если каждая инструкция, «разрушенная» этой ошибкой, может быть заменена последовательностью «работающих» инструкций, производящих аналогичный эффект.

Следующие задачи ссылаются на показанную ниже ошибку:

	Ошибка
а	Постоянная единица
б	Появление нуля, если Инструкция [31-26] имеет во всех разрядах нули, и отсутствие ошибки в противном случае

4.8.4 [10] <4.3, 4.4> Повторите упражнение 4.8.1, но теперь попробуйте протестировать наличие этой ошибки в управляющем сигнале MemRead.

4.8.5 [10] <4.3, 4.4> Повторите упражнение 4.8.1, но теперь попробуйте протестировать наличие этой ошибки в управляющем сигнале Jump.

4.8.6 [40] <4.3, 4.4> Используя единый тест, описание которого было дано в упражнении 4.8.1, мы можем протестировать ошибки в нескольких разных сигналах, но обычно не во всех этих сигналах. Дайте описание серии тестов, обнаруживающих эту ошибку на всех выходах мультиплексоров (каждого выходного разряда из каждого из пяти мультиплексоров). Попробуйте, по возможности, сделать это с помощью нескольких тестов, состоящих из одной инструкции.

Упражнение 4.9

В данном упражнении будет изучена работа однотактного операционного блока применительно к одной конкретной инструкции. Задачи данного упражнения ссылаются на следующую MIPS-инструкцию:

	Инструкция
а	lw $1,40($6)
б	Label: bne $1,$2,Label

4.9.1 [10] <4.4> каким будет значение слова инструкции?

4.9.2 [10] <4.4> Каким будет номер регистра, предоставляемый входу файла регистров «Читаемый регистр 1»? Читается ли на самом деле этот регистр? А что можно сказать насчет входа «Читаемый регистр 2»?

4.9.3 [10] <4.4> Каким будет номер регистра, предоставляемый входу файла регистров «Записываемый регистр»? Действительно ли данные записываются в этот регистр?

Разные инструкции требуют выставления в операционном блоке разных сигналов управления. Остальные задачи данного упражнения ссылаются на два сигнала управления с рис. 4.21:

	Сигнал управления 1	Сигнал управления 2
а	RegDst	MemRead
б	RegWrite	MemRead

4.9.4 [20] <4.4> Какие значения имеют эти два сигнала для данной инструкции?

4.9.5 [20] <4.4> Нарисуйте для операционного блока, изображенного на рис. 4.21, логическую диаграмму той части блока управления, которая выставляет только первый сигнал. Предположим, что нам нужно поддерживать только инструкции `lw`, `sw`, `beq`, `add` и `j` (переход).

4.9.6 [20] <4.4> еще раз выполните упражнение 4.9.5, но уже для выставления обоих сигналов.

Упражнение 4.10

В этом упражнении будет изучено, как продолжительность тактового цикла процессора влияет на конструкцию блока управления, и наоборот. В задачах данного упражнения предполагается, что логические блоки, используемые для реализации операционного блока, имеют следующие показатели латентности:

	Память инструкций (I-Mem)	Сумматор (Add)	Мультиплексор (Mux)	АЛУ (ALU)	Файл регистров (Regs)	Память данных (D-Mem)	Расширитель знака (Sign-extend)	Блок сдвига влево на два (Shift-left-2)	Блок управления АЛУ (ALU Ctrl)
а	400 пс	100 пс	30 пс	120 пс	200 пс	350 пс	20 пс	0 пс	50 пс
б	500 пс	150 пс	100 пс	180 пс	220 пс	1000 пс	90 пс	20 пс	55 пс

4.10.1 [10] <4.2, 4.4> Сколько времени должна занять у блока управления генерация сигнала MemWrite, чтобы не был удлинен критический путь операционного блока, показанного на рис. 4.21?

4.10.2 [20] <4.2, 4.4> На генерацию какого управляющего сигнала на рис. 4.21 тратится больше всего времени и каков должен быть этот показатель у блока управления, если нужно избежать присутствия этой генерации в критическом пути?

4.10.3 [20] <4.2, 4.4> Выдача какого из сигналов управления на рис. 4.21 наиболее критична по времени и сколько времени должно уйти у блока управления на его генерацию, чтобы этот процесс не попал в критический путь?

Для остальных задач данного упражнения предполагается, что время, необходимое блоку управления для генерации отдельных сигналов управления, имеет следующие показатели:

	RegDst	Jump	Branch	MemRead	MetoReg	ALUOp	MeWrite	ALUSrc	RegWrite
а	720 пс	730 пс	600 пс	400 пс	700 пс	200 пс	710 пс	200 пс	800 пс
б	1600 пс	1600 пс	1400 пс	500 пс	1400 пс	400 пс	1500 пс	400 пс	1700 пс

4.10.4 [20] <4.4> Какой будет продолжительность тактового цикла процессора?

4.10.5 [20] <4.4> Если есть возможность ускорить генерацию сигналов управления, но стоимость всего процессора возрастает на один доллар при каждом увеличении скорости отдельно взятого сигнала управления на 5 пс, генерацию каких сигналов управления вы будете ускорять для максимизации производительности и на сколько? Какой станет стоимость (в пересчете на один процессор) такого увеличения производительности?

4.10.6 [30] <4.4> Если процессор уже слишком дорог, то вместо платы за ускорение его работы, как это делалось в упражнении 4.10.5, мы хотим минимизировать стоимость без снижения скорости работы процессора. Если можно было бы воспользоваться для реализации сигналов управления более медленной логикой, экономя по одному доллару за добавление к показателю латентности отдельного сигнала управления лишних 5 пс, генерацию каких сигналов управления вы бы замедлили и на сколько снизилась бы стоимость процессора без уменьшения скорости его работы?

Упражнение 4.11

В данном упражнении будет подробно изучено, как выполняется инструкция в однотактном операционном блоке. Задачи данного упражнения ссылаются на тактовый цикл, в котором процессор извлекает следующее слово инструкции:

	Слово инструкции
а	10001100010000110000000000010000
б	00010000001000110000000000001100

4.11.1 [5] <4.4> Какими будут данные на выходах блока расширения знака и блока перехода «Сдвиг влево на два» (в верхнем левом углу рис. 4.21) для этого слова инструкции?

4.11.2 [10] <4.4> Какими будут значения на входах управления блоков АЛУ для этой инструкции?

4.11.3 [10] <4.4> Каким будет новый адрес в счетчике команд (PC) после выполнения этой инструкции? Выделите путь, через который нужно пройти, чтобы определить это значение.

Для остальных задач данного упражнения предполагается, что память данных заполнена одними нулями, а регистры процессора в начале цикла извлечения показанного выше слова инструкции имеют следующие значения:

	$0	$1	$2	$3	$4	$5	$6	$8	$12	$31
а	0	1	2	3	–4	5	6	8	1	–32
б	0	–16	–2	–3	4	–10	–6	–1	8	–4

4.11.4 [10] <4.4> Покажите для каждого мультиплексора Mux значения его выходных данных во время выполнения этой инструкции, а также значения этих регистров.

4.11.5 [10] <4.4> Какими будут входные значения для АЛУ и двух сумматоров?

4.11.6 [10] <4.4> Какими будут значения на всех входах для блока «Регистры»?

Упражнение 4.12

В данном упражнении будет изучено влияние конвейеризации на продолжительность тактового цикла процессора. Для задач данного упражнения предполагается, что отдельные стадии в операционном блоке имеют следующие показатели латентности:

	IF	ID	EX	MEM	WB
а	300 пс	400 пс	350 пс	500 пс	100 пс
б	200 пс	150 пс	120 пс	190 пс	140 пс

4.12.1 [5] <4.5> Какой будет продолжительность тактового цикла в конвейеризированном и неконвейеризированном процессоре?

4.12.2 [10] <4.5> Каким будет общий показатель латентности инструкции `lw` в конвейеризированном и неконвейеризированном процессоре?

4.12.3 [10] <4.5> Если мы можем разбить одну стадию конвейеризированного операционного блока на две новые, у каждой из которых была бы вдвое более низкая латентность, чем у исходной стадии, какую бы стадию вы разбили и какой была бы новая продолжительность тактового цикла процессора?

Для остальных задач данного упражнения предположим, что инструкции, выполняемые процессором, имеют следующее процентное распределение:

	АЛУ	beq	lw	sw
а	50%	25%	15%	10%
б	30%	25%	30%	15%

4.12.4 [10] <4.5> Каким будет объем использования памяти данных (в процентах от используемых циклов) при условии, что задержки или конфликты отсутствуют?

4.12.5 [10] <4.5> Каким будет объем использования порта записи в регистр блока «Регистры» при условии, что задержки или конфликты отсутствуют?

4.12.6 [30] <4.5> Вместо однотактной организации можно воспользоваться многотактной, где выполнение каждой инструкции занимает несколько циклов, но выполнение инструкции должно завершиться до того, как будет извлечена следующая инструкция. При такой организации инструкция проходит только через те стадии, которые ей действительно необходимы (например, инструкция `ST` проходит только через четыре стадии, потому что ей не нужна стадия WB). Сравните продолжительности тактовых циклов и показатели времени выполнения для однотактной, многотактной и конвейеризированной организации.

Упражнение 4.13

В данном упражнении будет изучено влияние зависимости от данных на работу базового пятистадийного конвейера, рассмотренного в разделе 4.5. Задачи данного упражнения ссылаются на следующую последовательность инструкций:

	Последовательность инструкций

а	lw $1,40($6)
	add $6,$2,$2
	sw $6,50($1)

б	lw $5,-16($5)
	sw $5,-16($5)
	add $5 $5,$5

4.13.1 [10] <4.5> Покажите зависимости и укажите их типы.

4.13.2 [10] <4.5> Предположим, что в данном конвейеризированном процессоре отсутствует препровождение данных. Покажите конфликты и добавьте пустые инструкции nop, позволяющие от них избавиться.

4.13.3 [10] <4.5> Предположим, что используется полное препровождение данных. Покажите конфликты и добавьте пустые инструкции nop, позволяющие от них избавиться.

Для остальных задач данного упражнения предполагаются следующие продолжительности тактовых циклов:

	Без препровождения данных	С полным препровождением данных	С препровождением только на уровне АЛУ-АЛУ
а	300 пс	400 пс	360 пс
б	200 пс	250 пс	220 пс

4.13.4 [10] <4.5> Каким будет общее время выполнения этой последовательности инструкций без препровождения данных и с полным препровождением? Какое ускорение будет достигнуто при добавлении полного препровождения к конвейеру, в котором его не было?

4.13.5 [10] <4.5> Добавьте nop-инструкции к этому коду для исключения конфликтов, при условии, что используется только препровождение данных на уровне АЛУ-АЛУ (препровождение из стадии MEM в стадию EX отсутствует)?

4.13.6 [10] <4.5> Каким будет общее время выполнения этой последовательности инструкций при препровождении данных только на уровне АЛУ-АЛУ? Каким будет ускорение по сравнению с конвейером, не использующим препровождение данных?

Упражнение 4.14

В данном упражнении будет изучено влияние конфликтов ресурсов, конфликтов управления и конструкции архитектуры набора инструкций на конвейеризированное выполнение. Задачи данного упражнения ссылаются на следующий фрагмент MIPS-кода:

	Последовательность инструкций

а	`lw $1,40($6)`
	`beq $2,$0,Label ; Предположим, что $2 = = $0`
	`sw $6,50($2)`
	`Label: add $2,$3,$4`
	`sw $3,50($4)`

б	`lw $5,-16($5)`
	`sw $4,-16($4)`
	`lw $3,-20($4)`
	`beq $2,$0,Label ; Предположим, что $2 != $0`
	`add $5,$1,$4`

4.14.1 [10] <4.5> Для данной задачи предположим, что все условные переходы удачно спрогнозированы (что исключает все конфликты управления) и не используются никакие слоты задержки. Если у нас используется только одна память (для инструкций и данных), то при необходимости каждого извлечения инструкции в том же цикле, в котором другая инструкция обращается к данным, у нас неизменно будет возникать структурный конфликт. Чтобы гарантировать дальнейший прогресс, этот конфликт должен быть разрешен в пользу инструкции, обращающейся к данным. Каким будет общее время выполнения этой последовательности инструкций в пятистадийном конвейере, имеющем только одну область памяти? Мы видели, что конфликты данных могут быть разрешены путем добавления к коду nop-инструкций. Можно ли сделать то же самое с этим структурным конфликтом? Почему?

4.14.2 [20] <4.5> Для данной задачи предположим, что все условные переходы удачно спрогнозированы (что исключает все конфликты управления) и не используются никакие слоты задержки. Если изменить инструкции загрузки-сохранения, чтобы они в качестве адреса использовали регистр (без смещения), этим инструкциям больше не понадобится использование АЛУ. В результате этого стадии MEM и EX могут быть совмещены и у конвейера останутся только четыре стадии. Измените данный код так, чтобы он соответствовал этой измененной архитектуре набора инструкций. Каким будет ускорение при выполнении данной последовательности инструкций, если предположить, что это изменение не повлияет на продолжительность тактового цикла?

4.14.3 [10] <4.5> Какое ускорение будет получено при выполнении этого кода, если исходы условных переходов будут определены на стадии ID, по сравнению с выполнением, при котором исходы будут определяться на стадии EX, если при этих условных переходах будут отсутствовать слоты задержки?

Для остальных задач данного упражнения предполагается, что отдельные стадии конвейера имеют следующие показатели латентности:

	IF	ID	EX	MEM	WB
а	100 пс	120 пс	90 пс	130 пс	60 пс
б	180 пс	100 пс	170 пс	220 пс	60 пс

4.14.4 [10] <4.5> Используя эти показатели латентности стадий конвейера, повторите вычисления ускорения из упражнения 4.14.2, но теперь примите в расчет

возможные изменения в продолжительности тактового цикла. Когда стадии EX и MEM совмещаются, многое из того, что на них делается, может быть проделано в параллельном режиме. В результате получившаяся стадия EX/MEM будет иметь показатель латентности больше, чем у исходных двух стадий, плюс 20 пс, необходимых для работы, которую нельзя выполнить в параллельном режиме.

4.14.5 [10] <4.5> Используя эти показатели латентности стадий конвейера, повторите вычисления ускорения из упражнения 4.14.3, но теперь примите в расчет возможные изменения в продолжительности тактового цикла. Представьте, что латентность стадии ID увеличилась на 50%, а латентность стадии EX уменьшилась на 10 пс, когда определение исхода условного перехода переместилось из стадии EX в стадию ID.

4.14.6 [10] <4.5> Каким будет новая продолжительность тактового цикла и время выполнения этой последовательности инструкций при условии отсутствия слотов задержки при условных переходах, если вычисление адреса для инструкции beq перемещено в стадию MEM? Каким будет ускорение, полученное благодаря такому изменению? Предположим, что при перемещении определения исхода условного перехода из стадии EX в стадию MEM латентность стадии EX уменьшилась на 20 пс, а латентность стадии MEM осталась без изменений.

Упражнение 4.15

В данном упражнении будет изучено, как архитектура набора инструкций влияет на конструкцию конвейера. Задачи данного упражнения ссылаются на следующую новую инструкцию:

а	bez1 (Rs),Label	если Память[Rs] = 0, тогда PC=PC+Смещение
б	swi Rd,Rs(Rt)	Память[Rs+Rt]=Rd

4.15.1 [20] <4.5> Что должно быть изменено в конвейеризированном операционном блоке для добавления этой инструкции к архитектуре набора инструкций MIPS?

4.15.2 [10] <4.5> Какие новые сигналы управления должны быть добавлены к вашему конвейеру из упражнения 4.15.1?

4.15.3 [20] <4.5, 4.13> Создает ли поддержка данной инструкции какие-нибудь новые конфликты? Стали ли задержки, связанные с существующими конфликтами, еще дольше?

4.15.4 [10] <4.5, 4.13> Приведите пример того, где эта инструкция может быть полезной, и последовательность существующих MIPS-инструкций, являющуюся заменой данной инструкции.

4.15.5 [10] <4.5, 4.11, 4.13> Если эта инструкция уже имеется в существующей архитектуре набора инструкций, объясните, как она может быть выполнена на современных процессорах, подобных AMD Barcelona.

Для последней задачи данного упражнения предполагается, что каждое использование новой инструкции заменяет заданное количество исходных инструкций,

что замена заданного количества исходных инструкций может быть произведена один раз и что при каждом выполнении новой инструкции ко времени выполнения программы добавляется заданное количество циклов задержки:

	Количество замен	По одной за каждые *n*-инструкций	Дополнительные циклы задержки
а	2	20	1
б	3	60	0

4.15.6 [10] <4.5> Каким будет увеличение скорости, достигаемое добавлением этой новой инструкции? В своих вычислениях нужно исходить из того, что показатель CPI исходной программы (без новой инструкции) равен 1.

Упражнение 4.16

Первые три задачи этого упражнения ссылаются на следующую MIPS-инструкцию:

	Инструкция
а	`lw $1,40($6)`
б	`add $5,$5,$5`

4.16.1 [5] <4.6> Что хранится в каждом регистре, расположенном между двумя стадиями конвейера в процессе выполнения данной инструкции?

4.16.2 [5] <4.6> Какие регистры должны быть прочитаны и какие регистры читаются на самом деле?

4.16.3 [5] <4.6> Что делает эта инструкция на стадиях EX и MEM?

Остальные три задачи данного упражнения ссылаются на следующий цикл. Предположим, что используется совершенное прогнозирование условных переходов (задержки из-за конфликтов управления отсутствуют), нет слотов задержки и что в конвейере реализована полная поддержка препровождения данных. Также предположим, что до выхода из цикла проводится множество итераций.

	Цикл

а	`Loop: lw $1,40($6)`
	`add $5,$5,$8`
	`add $6,$6,$8`
	`sw $1,20($5)`
	`beq $1 $0,Loop`

б	`Loop: add $1,$2,$3`
	`sw $0,0($1)`
	`sw $0,4($1)`
	`add $2,$2,$4`
	`beq $2,$0,Loop`

4.16.4 [10] <4.6> Покажите диаграмму конвейерного выполнения для третьей итерации данного цикла, от цикла, в котором извлекается первая инструкция этой

итерации, и (исключительно) до цикла, в котором извлекается первая инструкция следующей итерации. Покажите все инструкции, находящиеся в конвейере в течение этих циклов (а не только принадлежащие третьей итерации).

4.16.5 [10] <4.6> Как часто (в процентах ко всем циклам) будет появляться цикл, в котором все пять стадий конвейера будут проделывать полезную работу?

4.16.6 [10] <4.6> Что хранится в регистре IF/ID в начале цикла, в котором извлекается первая инструкция третьей итерации?

Упражнение 4.17

Для задач данного упражнения предполагается, что инструкции выполняются конвейеризированным процессором в следующих пропорциях:

	add	beq	lw	sw
а	50%	25%	15%	10%
б	30%	15%	35%	20%

4.17.1 [5] <4.6> В каком процентном отношении ко всем тактовым циклам сумматор условного перехода на стадии EX генерирует реально используемое значение при отсутствии задержек и при 60% показателе выполнения условных переходов?

4.17.2 [5] <4.6> Как часто (в процентном отношении ко всем циклам) действительно понадобится использование всех трех портов файла регистров (двух на чтение и одного на запись) в одном и том же цикле, если не будет задержек?

4.17.3 [5] <4.6> Как часто (в процентном отношении ко всем циклам) будет использоваться память данных, если не будет задержек?

У каждой стадии конвейера, показанного на рис. 4.29, имеется свой показатель латентности. Кроме этого, при конвейеризации используются регистры между стадиями (см. рис. 4.31), и каждый из них добавляет свою латентность. Для остальных задач данного упражнения предполагаются следующие показатели латентности для логики внутри каждой стадии конвейера и для каждого регистра между двумя стадиями:

	IF	ID	EX	MEM	WB	Регистр конвейера
а	100 пс	120 пс	90 пс	130 пс	60 пс	10 пс
б	180 пс	100 пс	170 пс	220 пс	60 пс	10 пс

4.17.4 [5] <4.6> Каким будет ускорение, достигнутое за счет конвейеризации однотактного операционного блока, при условии отсутствия задержек?

4.17.5 [10] <4.6> Все инструкции загрузки-сохранения можно превратить в инструкции, основанные на значении регистра (без смещения), и поместить обращение к памяти в параллель с работой АЛУ. Какой будет продолжительность тактового цикла, если это будет сделано за один цикл и в конвейеризированном операционном блоке? Предположим, что латентность новой стадии EX/MEM равна наибольшему из показателей их латентности.

4.17.6 [10] <4.6> Изменение, рассмотренное в упражнении 4.17.5, требует преобразования многих инструкций `lw`/`sw` в последовательность из двух инструкций. Если такое преобразование требуется для 50% таких инструкций, то каким будет общее ускорение, достигнутое благодаря замене пятистадийного конвейера на четырехстадийный, где стадии EX и MEM выполняются в параллельном режиме?

Упражнение 4.18

Первые три задачи данного упражнения ссылаются на выполнение следующей инструкции в конвейеризированном операционном блоке с рис. 4.45 и предполагают следующую продолжительность тактового цикла, латентность АЛУ и латентность мультиплексора Mux:

	Инструкция	Продолжительность тактового цикла	Латентность АЛУ	Латентность Mux
а	`add $1,$2,$3`	100 пс	80 пс	10 пс
б	`slt $2,$1,$3`	80 пс	50 пс	20 пс

4.18.1 [10] <4.6> Какими для каждой стадии конвейера будут значения сигналов управления, выставленных этой инструкцией на данной стадии?

4.18.2 [10] <4.6, 4.7> Сколько времени займет генерация сигнала управления ALUSrc? Сравните это показатель с показателем для однотактной организации.

4.18.3 Каково значение сигнала PCSrc для этой инструкции? Этот сигнал генерируется на раннем этапе стадии MEM (для этого используется только один логический элемент И). Какую можно привести причину в пользу того, чтобы сделать это на стадии EX? Какую можно привести причину против того, чтобы сделать это на стадии EX?

Остальные задачи данного упражнения ссылаются на следующие сигналы из табл. 4.6:

	Сигнал 1	Сигнал 2
а	RegDst	RegWrite
б	MemRead	RegWrite

4.18.4 [5] <4.6> Определите стадию конвейера, на которой генерируется и на которой используется каждый из этих сигналов.

4.18.5 [5] <4.6> Для какой MIPS-инструкции (или для каких инструкций) оба эти сигнала выставляются в единицу?

4.18.6 [10] <4.6> Один из этих сигналов возвращается назад по конвейеру. Какой это сигнал? Присутствует ли здесь парадокс путешествия во времени? Объясните.

Упражнение 4.19

Это упражнение предназначено для того, чтобы помочь разобраться в балансе «стоимость—сложность—производительность» перенаправления данных в конвейеризированном процессоре. Задачи в данном упражнении ссылаются на конвейеризированные операционные блоки, изображенные на рис. 4.41. В этих задачах предполагается, что для каждой инструкции, выполняемой в процессоре, следующая за ней инструкция имеет специфичный тип зависимости данных «чтение после записи». Тип зависит от стадии (EX или MEM) и следующей инструкции (первая инструкция, которая следует за той, которая произвела результат, вторая инструкция, которая следует за ней, или обе эти инструкции). Предполагается, что запись в регистр происходит в первой половине тактового цикла а чтение регистра происходит во второй половине цикла, поэтому зависимости «EX к третьей» и «MEM ко второй» в расчет не берутся, поскольку они не могут привести в конфликтам данных. Также предполагается, что в отсутствие конфликтов данных показатель CPI процессора равен единице.

	EX только к первой	EX к первой и второй	EX только ко второй	MEM к первой
а	10%	10%	5%	25%
б	15%	5%	10%	20%

4.19.1 [10] <4.7> Какая доля циклов придется на задержки, связанные с конфликтами данных, если не будет использовано препровождение данных?

4.19.2 [5] <4.7> Какая доля циклов придется на задержки, связанные с конфликтами данных, если будет использовано полное препровождение данных (препровождение всех результатов, которые могут ему подвергаться)?

4.19.3 [10] <4.7> Предположим, что мы не можем иметь мультиплексоры с тремя входами, которые необходимы для полного препровождения данных. Мы должны решить, откуда лучше осуществлять препровождение: только из регистра конвейера EX/MEM (препровождение в следующий цикл) или только из регистра конвейера MEM/WB (препровождение через два цикла). Какой из двух вариантов приводит к меньшему количеству циклов задержки данных?

Оставшиеся три задачи данного упражнения ссылаются на показатели латентности для отдельных стадий конвейера. Для стадии EX показатели латентности даются отдельно для процессора без препровождения и для процессора с различными разновидностями препровождения.

	IF	ID	EX (без препр.)	EX (с полным препр.)	EX (Препр. только из EX/MEM)	EX (Препр. только из MEM/WB)	MEM	WB
а	100 пс	50 пс	75 пс	110 пс	100 пс	100 пс	100 пс	60 пс
б	250 пс	300 пс	200 пс	350 пс	320 пс	310 пс	300 пс	200 пс

4.19.4 [10] <4.7> Каким будет достигнутое за счет полного препровождения данных ускорение при заданных конфликтных возможностях и показателях латентности стадий конвейера?

4.19.5 [10] <4.7> Каким будет дополнительное ускорение (относительно процессора с препровождением данных), если мы воспользовались путешествием во времени, которое устраняет все конфликты данных? Предположим, что только что путешествие во времени добавляет 100 пс к латентности полностью оборудованной препровождением стадии EX.

4.19.6 [20] <4.7> Еще раз выполните упражнение 4.19.3, но на этот раз определите, в каком из двух случаев время, затрачиваемое на инструкцию, будет меньше.

Упражнение 4.20

Задачи данного упражнения ссылаются на следующие последовательности инструкций:

	Последовательность инструкций

а	`lw $1,40($2)`
	`add $2,$3,$3`
	`add $1,$1,$2`
	`sw $1,20($2)`

б	`add $1,$2,$3`
	`sw $2,0($1)`
	`lw $1,4($2)`
	`add $2,$2,$1`

4.20.1 [5] <4.7> Найдите все зависимости от данных в этой последовательности инструкций.

4.20.2 [10] <4.7> Найдите для этой последовательности инструкций все конфликты, возникающие при ее выполнении на пятистадийном конвейере, использующем и не использующем препровождение данных.

4.20.3 [10] <4.7> Для сокращения продолжительности тактового цикла мы рассматриваем возможность разбиения стадии MEM на две стадии. Выполните упражнение 4.20.2 для такого шестистадийного конвейера.

Для остальных трех задач данного упражнения предполагается, что перед выполнением инструкций все значения в памяти данных равны нулю, а регистры с $0 по $3 имеют следующие исходные значения:

	$0	$1	$2	$3
а	0	1	31	1000
б	0	-2	63	2500

4.20.4 [5] <4.7> Какое значение будет препровождено первым и какое значение будет им переписано?

4.20.5 [10] <4.7> Если при разработке блока обнаружения конфликтов мы предположили, что будет реализовано препровождение данных, но потом забыли об этом, то какими в конечном итоге будут значения регистров после выполнения этой последовательности инструкций?

4.20.6 [10] <4.7>Добавьте для конструкции, описанной в упражнении 4.20.5, пустые инструкции (nop), чтобы обеспечить правильное выполнение, несмотря на отсутствие поддержки препровождения данных.

Упражнение 4.21

Это упражнение предназначено для того чтобы помочь вам разобраться во взаимоотношениях между препровождением, обнаружением конфликтов и конструкцией архитектуры набора инструкций. Задачи данного упражнения ссылаются на приведенные далее последовательности инструкций и предполагают, что они выполняются операционным блоком, имеющим конвейер из пяти стадий:

	Последовательность инструкций

а	lw $1,40($6)
	add $2,$3,$1
	add $1,$6,$4
	sw $2,20($4)
	and $1,$1,$4

б	add $1,$5,$3
	sw $1,0($2)
	lw $1,4($2)
	add $5,$5,$1
	sw $1,0($2)

4.21.1 [5] <4.7> Вставьте пустые инструкции (nop), гарантирующие правильное выполнение в случае отсутствия препровождения или обнаружения конфликтов.

4.21.2 [10] <4.7> Выполните упражнение 4.21.1 еще раз, но на этот раз используйте инструкции nop только в том случае, когда конфликт не может быть устранен путем изменения или перестановки этих инструкций. Можете предположить, что для хранения временных значений в вашем модифицированном коде допускается использование регистра R7.

4.21.3 [10] <4.7> Что произойдет при выполнении данного кода, если у процессора есть препровождение, но мы забыли реализовать блок обнаружения конфликтов?

4.21.4 [20] <4.7> Укажите, какие сигналы выставляются в каждом цикле блоками обнаружения конфликтов и препровождения, показанными на рис. 4.53, если в процессе выполнения данного кода для первых пяти циклов используется препровождение данных.

4.21.5 [10] <4.7> Какие новые входные и выходные сигналы нужны нам для блока обнаружения конфликтов, показанного на рис. 4.53, при отсутствии препровождения данных? Объясните, используя в качестве примера эту последовательность инструкций, зачем нужен каждый сигнал.

4.21.6 [20] <4.7> Определите какие выходные сигналы выставляются в каждом из пяти первых циклов при выполнении данного кода в новом блоке обнаружения конфликтов из упражнения 4.21.5.

Упражнение 4.22

Это упражнение предназначено для того, чтобы помочь вам разобраться во взаимоотношениях между слотами задержки, конфликтами управления и выполнением условных переходов в конвейеризированном процессоре. В данном упражнение предполагается, что следующий MIPS-код выполняется на процессоре, имеющем пятистадийный конвейер, полное препровождение и блок прогнозирования условных переходов:

а	`Label1: lw  $1,40($6)`
	`beq $2,$3,Label2 ; Состоялся`
	`add $1,$6,$4`
	`Label2: beq $1,$2,Label1 ; Не состоялся`
	`sw  $2,20($4)`
	`and $1,$1,$4`

б	`add $1,$5,$3`
	`Label1: sw  $1,0($2)`
	`add $2,$2,$3`
	`beq $2,$4,Label1 ; Не состоялся`
	`add $5,$5,$1`
	`sw  $1,0($2)`

4.22.1 [10] <4.8> Нарисуйте для этого кода конвейерную диаграмму выполнения, при условии, что слоты задержки отсутствуют, а условные переходы выполняются на стадии EX.

4.22.2 [10] <4.8> Выполните упражнение 4.22.1 еще раз, но теперь при условии использования слотов задержки. Теперь в этом коде инструкция, которая следует за инструкцией условного перехода, является инструкцией слота задержки для этого условного перехода.

4.22.3 [20] <4.8> Один из способов перемещения разрешения условного перехода на одну стадию раньше заключается в отказе от использования операции АЛУ в условных переходах. Инструкции условного перехода должны иметь вид «bez Rd,Label» и «bnez Rd,Label» и осуществлять условный переход, когда регистр соответственно имеет или не имеет нулевое значение. Внесите изменения в этот код для использования вместо инструкции `beq` этих инструкций условного перехода. Можно предположить, что для использования в качестве временного регистра доступен регистр `$8` и что может быть использована инструкция R-типа `seq` (set if equal — установка по равенству).

В разделе 4.8 описывается, как можно уменьшить трудности, возникающие из-за конфликтов управления, переместив выполнение условного перехода в стадию ID. Как показано на рис. 4.55, этот подход предполагает использование на стадии ID

специализированного блока сравнения (компаратора). Но такой подход потенциально увеличивает латентность стадии ID и требует дополнительной логики препровождения и обнаружения конфликтов.

4.22.4 [10] <4.8> Используя в качестве примера первую инструкцию условного перехода, имеющуюся в данном коде, опишите логику обнаружения конфликтов, необходимую для поддержки выполнения условного перехода на стадии ID, как на рис. 4.55. Конфликт какого типа предполагается обнаруживать с помощью этой новой логики?

4.22.5 [10] <4.8> Каким будет ускорение, достигнутое благодаря перемещению выполнения условного перехода на стадию ID, применительно к данному коду? Обоснуйте свой ответ. При вычислении ускорения следует предположить, что дополнительное сравнение на стадии ID не повлияет на продолжительность тактового цикла.

4.22.6 [10] <4.8> Используя в качестве примера первую инструкцию условного перехода, имеющуюся в данном коде, опишите поддержку препровождения, которая должна быть добавлена для обеспечения выполнения условного перехода на стадии ID. Сравните сложность этого нового блока препровождения со сложностью существующего блока препровождения, показанного на рис. 4.55.

Упражнение 4.23

Важность наличия хорошего блока прогнозирования условных переходов зависит от того, насколько часто выполняются условные переходы. Наряду с точностью блока прогнозирования определяется также и время, затрачиваемое на задержки, связанные с неверно спрогнозированными условными переходами. В данном упражнении следует представить, что динамические инструкции среди разных категорий инструкции имеют следующее распределение:

	R-типа	beq	jmp	lw	sw
а	50%	15%	10%	15%	10%
б	30%	10%	5%	35%	20%

Также нужно представить следующую точность прогнозирования условных переходов:

	Всегда состоится	Никогда не состоится	Двухразрядный предсказатель
а	40%	60%	80%
б	60%	40%	95%

4.23.1 [10] <4.8> Циклы задержек, возникающие из-за неверно спрогнозированных условных переходов, увеличивают показатель CPI. На сколько увеличится показатель CPI из-за неверно спрогнозированных условных переходов при использовании предсказателя, считающего, что переход всегда состоится? Предположим, что исход условного перехода определяется на стадии EX, конфликты данных отсутствуют и слоты задержек не используются.

4.23.2 [10] <4.8> Выполните упражнение 4.23.1 еще раз для предсказателя «никогда не состоится».

4.23.3 [10] <4.8> Выполните упражнение 4.23.1 еще раз для двухразрядного предсказателя.

4.23.4 [10] <4.8> Какое ускорение может быть достигнуто при использовании двухразрядного предсказателя, если мы сможем преобразоваться половину инструкций условного перехода путем замены инструкции условного перехода инструкцией АЛУ? Предположим, что у инструкций, действия которых будут предсказаны правильно и неправильно, имеются равные шансы на замену.

4.23.5 [10] <4.8> Какое ускорение может быть достигнуто при использовании двухразрядного предсказателя, если мы сможем преобразоваться половину инструкций условного перехода путем замены инструкции условного перехода двумя инструкциями АЛУ? Предположим, что у инструкций, действия которых будут предсказаны правильно и неправильно, имеются равные шансы на замену.

4.23.6 [10] <4.8> Действия одних инструкций условного перехода более предсказуемы, чем действия других. Если мы знаем, что 80% всех выполненных инструкций условного перехода являются легкими для предсказания условными переходами к началу цикла, то какой будет точность двухразрядного предсказателя для оставшихся 20% инструкций условного перехода?

Упражнение 4.24

В данном упражнении оценивается точность различных предсказателей условных переходов для следующих повторяющихся схем их исходов (например, в цикле):

	Исходы условных переходов
а	С, С, НС, С
б	С, С, С, НС, НС

4.24.1 [5] <4.8> Какой будет точность предсказателя «всегда состоится» и предсказателя «никогда не состоится» для этой последовательности исходов условных переходов?

4.24.2 [5] <4.8> Какой будет точность двухразрядного предсказателя для первых четырех условных переходов в этой схеме при условии, что предсказатель начинает свою работу из состояния, показанного в нижнем левом углу рис. 4.56 («Предсказание, что не состоится»).

4.24.3 [10] <4.8> Какой будет точность двухразрядного предсказателя, если эта схема постоянно повторяется?

4.24.4 [30] <4.8> Придумайте конструкцию предсказателя, достигающего совершенной точности, если эта схема постоянно повторяется. Ваш предсказатель должен быть последовательной схемой с одним выходом, выдающим предсказание (1 для предсказания, что переход состоится, и 0 для предсказания, что он не состоится), и со входами, предназначенными только лишь для тактовых импульсов

и сигнала управления, показывающего, что инструкция является условным переходом.

4.24.5 [10] <4.8> Какой будет точность вашего предсказателя из упражнения 4.24.4, если задана повторяющаяся схема, прямо противоположная данной схеме?

4.24.6 [20] <4.8> Выполните упражнение 4.24.4 еще раз, но теперь при условии, что ваш предсказатель должен уметь в конечном итоге (после периода выхода на стабильный режим работы, в котором он может выдавать неверные предсказания) выдавать точные предсказания для обеих представленных схем и их инверсий. У вашего предсказателя должен быть вход, который сообщает ему о реальном исходе условного перехода. *Подсказка*: этот вход позволяет вашему предсказателю определять, какая из двух повторяющихся схем задана.

Упражнение 4.25

В данном упражнении исследуется, как обработка исключений влияет на конструкцию конвейера. Первые три задачи данного упражнения ссылаются на следующие две инструкции:

	Инструкция 1	Инструкция 2
а	add $0,$1,$2	bne $1,$2,Label
б	lw $2,40($3)	nand $1,$2,$3

4.25.1 [5] <4.9> Какое исключение может каждая из этих инструкций инициировать? Укажите стадию конвейера, на которой обнаруживается каждое из этих исключений.

4.25.2 [10] <4.9> Покажите, как должна быть изменена организация конвейера, чтобы появилась возможность обработки этого исключения, если для каждого исключения имеется отдельный адрес обработчика. Можно предположить, что адреса этих обработчиков были известны еще при конструировании процессора.

4.25.3 [10] <4.9> Если вторая инструкция из этой таблицы извлекается сразу же после первой инструкции, опишите, что происходит в конвейере, когда первая инструкция становится причиной первого исключения из тех, что перечислены при выполнении упражнения 4.25.1.

Постройте диаграмму конвейерного выполнения от момента извлечения первой инструкции до момента завершения выполнения первой инструкции обработчика исключения.

Для остальных трех задач данного упражнения предполагается, что обработчики исключений находятся по следующим адресам:

	Обработчик переполнения	Обработчик недопустимого адреса данных	Обработчик неопределенной инструкции	Обработчик недопустимого адреса инструкции	Обработчик сбоя оборудования
а	0xFFFFF000	0xFFFFF100	0xFFFFF200	0xFFFFF300	0xFFFFF400
б	0x00000008	0x00000010	0x00000018	0x00000020	0x00000028

4.25.4 [5] <4.9> Каким будет адрес обработчика исключения из упражнения 4.25.3? Что произойдет, если по указанному адресу в памяти инструкций будет находиться недопустимая?

4.25.5 [20] <4.9> При векторной обработке исключений таблица с адресами обработчиков прерываний находится в памяти данных по известному (фиксированному) адресу. Внесите изменения в конвейер для реализации этого механизма обработки исключений. Выполните упражнение 4.25.3 еще раз, используя этот модифицированный конвейер и векторную обработку исключений.

4.25.6 [15] <4.9> Мы хотим эмулировать векторную обработку исключений (описанную в упражнении 4.25.5) на машине, имеющей только один фиксированный адрес обработчика. Напишите код, который должен находиться по этому фиксированному адресу. *Подсказка*: этот код должен идентифицировать исключение, получать правильный адрес из таблицы векторов исключений и переносить выполнение на этот обработчик.

Упражнение 4.26

В данном упражнении исследуется влияние обработки исключений на конструкцию блока управления и на продолжительность тактового цикла процессора. Первые три задачи данного упражнения ссылаются на следующие MIPS-инструкции, инициирующие исключение:

	Инструкция	Исключение
а	`add $0,$1,$2`	Арифметическое переполнение
б	`lw $2,40($3)`	Неправильный адрес в памяти данных

4.26.1 [10] <4.9> Определите для каждой стадии конвейера, показанного на рис. 4.59, значения сигналов управления, связанных с исключением, по мере прохождения этой инструкции через данную стадию конвейера.

4.26.2 [5] <4.9> Некоторые сигналы управления, генерируемые на стадии ID, сохраняются в регистре конвейера ID/EX, а некоторые следуют непосредственно в стадию EX. Объясните, почему так происходит, используя в качестве примера эту инструкцию.

4.26.3 [10] <4.9> Стадию EX можно ускорить, если мы ведем проверку исключений на стадии, которая следует за той стадией, на которой складываются условия для возникновения исключения. Используя эту инструкцию в качестве примера, опишите основной недостаток этого подхода.

Для остальных трех задач данного упражнения предполагается, что стадии конвейера обладают следующими показателями латентности:

	IF	ID	EX	MEM	WB
а	300 пс	320 пс	350 пс	350 пс	100 пс
б	200 пс	170 пс	210 пс	210 пс	150 пс

4.26.4 [10] <4.9> Каким будет общее ускорение при перемещении проверки на переполнение на стадию MEM, если исключение, связанное с переполнением, происходит один раз для каждых 100 000 выполняемых инструкций? Предполагается, что такое изменение уменьшает показатель латентности стадии EX на 30 пс и что показатель IPC, достигаемый конвейеризированным процессором при отсутствии исключений, равен 1.

4.26.5 [20] <4.9> Можем ли мы сгенерировать сигналы управления исключением не на стадии ID, а на стадии EX? Объясните, как это будет работать или почему это не будет работать, используя инструкцию «bne $4,$5,Label» и эти показатели латентности стадии конвейера в качестве примера.

4.26.6 [10] <4.9> Определите, сколько сигналов сброса должен сгенерировать блок управления при условии, что латентность каждого мультиплексора (Mux) равна 40 пс? Какой из сигналов является наиболее важным?

Упражнение 4.27

В данном упражнении исследуется взаимодействие обработки исключения с инструкциями условного перехода и инструкциями загрузки-сохранения. Задачи данного упражнения ссылаются на следующие инструкции условного перехода и соответствующие инструкции слота задержки:

	Условный переход и слот задержки
а	beq $1,$0,Label sw $6,50($1)
б	beq $5,$0,Label ror $5,$4,$3

4.27.1 [20] <4.9> Предположим, что данный условный переход правильно предсказан состоявшимся, но затем инструкция, находящаяся на метке «Label», оказалась неопределенной инструкцией. Опишите, что сделано на каждой стадии конвейера для каждого цикла, начиная с цикла, на котором условный переход декодирован, до цикла, на котором извлечена первая инструкция обработчика исключения.

4.27.2 [10] <4.9> Выполните упражнение 4.27.1 еще раз, но теперь при условии, что инструкция в слоте задержки также вызывает исключение, связанное с ошибкой оборудования, когда находится на стадии MEM.

4.27.3 [10] <4.9> Каким будет значение EPC, если условный переход состоялся, но слот задержки вызвал исключение? Что произойдет после того, как завершится работа обработчика исключения?

Остальные три задачи данного упражнения ссылаются на инструкцию сохранения:

	Инструкция сохранения
а	sw $6,50($1)
б	sw $5,60($3)

4.27.4 [10] <4.9> Что произойдет, если условный переход состоялся, инструкция, находящаяся на метке «Label», оказалась недопустимой, первой инструкцией обработчика исключения оказалась показанная выше инструкция sw и эта инструкция сохранения обращается по недопустимому адресу данных?

4.27.5 [10] <4.9> Если вычисление адреса загрузки-сохранения может привести к переполнению, можем ли мы отложить обнаружение исключения, связанного с переполнением до стадии MEM? Для объяснения происходящего используйте показанную инструкцию сохранения.

4.27.6 [10] <4.9> Для отладки полезно будет иметь возможность обнаружения момента, когда конкретное значение записывается по конкретному адресу памяти. Мы хотим добавить два новых регистра: WADDR и WVAL. Процессор должен инициировать исключение, когда значение, равное значению регистра WVAL, готово к записи по адресу, значение которого находится в регистре WADDR. Какие изменения вы внесете в конвейер для реализации этого замысла? Как эта инструкция sw будет обработана вашим модифицированным операционным блоком?

Упражнение 4.28

В данном упражнении сравнивается производительность процессоров с запуском одной инструкции с параллельным запуском двух инструкций, при котором учитывается преобразование программ, произведенное для оптимизации выполнения на процессоре с параллельным запуском двух инструкций. Задачи данного упражнения ссылаются на следующий цикл (написанный на языке C):

	Код на языке C
а	`for(i=0;i<j;i++)` `  b[i]=a[i];`
б	`for(i=0,a[i]!=a[i+1];i++)` `  a[i]=0;`

При написании MIPS-кода следует предположить, что переменные хранятся в регистрах в определенном ниже порядка и что все регистры, за исключением тех, что показаны свободными, используются для хранения различных переменных, поэтому они не могут использоваться для каких-нибудь других целей.

	i	j	a	b	c	Свободны
а	$1	$2	$3	$4	$5	$6,$7,$8
б	$4	$5	$6	$7	$8	$1,$2,$3

4.28.1 [10] <4.10> Транслируйте этот C-код в инструкции MIPS. Ваша трансляция должна быть непосредственной, без перестановки инструкций с целью получения более высокой производительности.

4.28.2 [10] <4.10> Если выход из цикла осуществляется после выполнения только двух итераций, нарисуйте конвейерную диаграмму для вашего MIPS-кода из упражнения 4.28.1, выполняемого на процессоре с параллельным запуском двух

инструкций, который показан на рис. 4.61. Предположим, что у процессора есть совершенный предсказатель условных переходов и он может извлекать в одном и том же цикле любые две инструкции (а не только инструкции, следующие друг за другом).

4.28.3 [10] <4.10> Переставьте инструкции своего кода из упражнения 4.28.1 для достижения более высокой производительности на процессоре с параллельным запуском двух инструкций со статической диспетчеризацией, изображенном на рис. 4.61.

4.28.4 [10] <4.10> Выполните упражнение 4.28.2 еще раз, но теперь используйте ваш код из упражнения 4.28.3.

4.28.5 [10] <4.10> Каким будет ускорение от перехода от процессора с запуском одной инструкции на процессор с параллельным запуском двух инструкций (см. рис. 4.61)? Для обоих вариантов с одиночным и двойным запуском инструкций используйте код из упражнения 4.28.1, полагая что выполняется 1 000 000 итераций цикла. Как и в упражнении 4.28.2, следует предположить, что у процессора есть совершенный предсказатель условных переходов и что процессор с параллельным запуском инструкций может извлекать любую из двух инструкций в одном и том же тактовом цикле.

4.28.6 [10] <4.10> Выполните упражнение 4.28.5 еще раз, но теперь предположите, что для процессора с параллельным запуском двух инструкций одна из инструкций, предназначенная для выполнения в цикле, может быть инструкцией любого типа, а другая должна быть инструкцией без обращения к памяти.

Упражнение 4.29

В данном упражнении рассматривается выполнение цикла на суперскалярном процессоре со статической диспетчеризацией. Чтобы упростить упражнение, предположим, что в одном и том же цикле разнотипные инструкции могут выполняться в любом сочетании, например на суперскалярном процессоре с параллельным запуском трех инструкций ими могут быть три АЛУ-операции, три инструкции условного перехода, три инструкции загрузки-сохранения или любая комбинация этих инструкций. Учтите, что это снимает только лишь ограничения, касающиеся ресурсов, но взаимозависимости, касающиеся данных и управления, должны по-прежнему обрабатываться соответствующим образом. Задачи в данном упражнении ссылаются на следующий цикл:

	Цикл

а	Loop: lw $1,40($6)
	add $5,$5,$1
	sw $1,20($5)
	addi $6,$6,4
	addi $5,$5,-4
	beq $5,$0,Loop

б	Loop: add $1,$2,$3
	sw $0,0($1)
	addi $2,$2,4
	beq $2,$0,Loop

4.29.1 [10] <4.10> Если выполняется большое количество (например, 1 000 000) итераций данного цикла, определите долю всех операций чтения регистров, которые будут полезны при использовании суперскалярного процессора со статическим параллельным запуском двух инструкций.

4.29.2 [10] <4.10> Если выполняется большое количество (например, 1 000 000) итераций данного цикла, определите долю всех операций чтения регистров, которые будут полезны при использовании суперскалярного процессора со статическим параллельным запуском трех инструкций. Сравните полученный результат с результатом для процессора с параллельным запуском двух инструкций из упражнения 4.29.1.

4.29.3 [10] <4.10> Если выполняется большое количество (например, 1 000 000) итераций данного цикла, определите долю всех циклов, в которых порты записи двух или трех регистров используются в суперскалярном процессоре со статическим параллельным запуском трех инструкций.

4.29.4 [20] <4.10> Проведите однократное развертывание этого цикла и спланируйте выполнение его инструкций для суперскалярного процессора со статическим параллельным запуском двух инструкций. Предположим, что в цикле всегда выполняется четное количество итераций. При внесении изменений в код для исключения зависимостей можно использовать регистры от $10 и до $20.

4.29.5 [20] <4.10> Каким будет ускорение от использования вашего кода из упражнения 4.29.4 вместо исходного кода при выполнении его на суперскалярном процессоре с параллельным запуском двух инструкций? Предположим, что в этом цикле выполняется большое количество итераций (например, 1 000 000).

4.29.6 [10] <4.10> Каким будет ускорение от использования вашего кода из упражнения 4.29.4 вместо исходного кода при выполнении его на конвейеризированном процессоре (с запуском только одной инструкции за один тактовый цикл)? Предположим, что в этом цикле выполняется большое количество итераций (например, 1 000 000).

Упражнение 4.30

В данном упражнении у нас будет ряд допущений. Во-первых, мы предположим, что суперскалярный процессор с параллельным запуском *N*-инструкций может выполнить любые *N* инструкций в одном и том же тактовом цикле, независимо от их типов. Во-вторых, мы предположим, что каждая инструкция выбирается независимо, безотносительно той инструкции которая ей предшествует или следует за ней. В-третьих, мы предположим, что зависимости, связанные с данными, не приводят к задержкам, что не используются слоты задержки и что условные переходы выполняются в конвейере на стадии EX. И наконец, мы предположим, что инструкции, выполняемые в программе, распределены следующим образом:

	АЛУ	Инструкция beq с правильно предсказанным исходом	Инструкция beq с неправильно предсказанным исходом	lw	sw
а	50%	18%	2%	20%	10%
б	40%	10%	5%	35%	10%

4.30.1 [5] <4.10> Каким будет показатель CPI, достигаемый при выполнении этой программы на статическом суперскалярном процессоре с параллельным запуском двух инструкций?

4.30.2 [10] <4.10> Каким будет ускорение, достигнутое за счет добавления возможности предсказания двух условных переходов за один цикл на статическом суперскалярном процессоре с параллельным запуском двух инструкций, чей предсказатель способен обслужить лишь один условный переход за один цикл? Предположим, что для условных переходов, которые предсказатель не может обработать, применяется политика задержки при условном переходе.

4.30.3 [10] <4.10> Какое ускорение достигается добавлением второго порта записи в файл регистров к статическому суперскалярному процессору, имеющему только один порт записи в файл регистров?

4.30.4 [5] <4.10> Какое ускорение достигается за счет доведения до совершенства работы предсказателя условных переходов на статическом суперскалярном процессоре с параллельным запуском двух инструкций?

4.30.5 [10] <4.10> Выполните упражнение 4.30.4 еще раз, но теперь для процессора с параллельным запуском четырех инструкций. Какое можно сделать заключение о важности удачного предсказания условных переходов, когда в процессоре увеличивается количество параллельно запускаемых инструкций?

4.30.6 <4.10> Выполните упражнение 4.30.5 еще раз, но теперь при условии, что у процессора с параллельным запуском имеется 50 стадий конвейера. Предположим, что каждая из исходных пяти стадий разбита на десять новых стадий и что условные переходы выполняются в первых из десяти новых стадий EX. Какое можно сделать заключение о важности удачного предсказания условных переходов, когда у процессора увеличивается глубина конвейера?

Упражнение 4.31

Задачи данного упражнения ссылаются на следующий цикл, представленный как в виде кода x86, так и в виде MIPS-трансляции этого кода. Можно предположить, что до выхода из этого цикла выполняется большое количество итераций. При определении производительности это означает, что вам нужно только определить, какая будет производительность в «устоявшемся состоянии», а не для нескольких первых и нескольких последних итераций цикла. Также можно предположить полную поддержку препровождения данных и совершенное предсказание условных переходов без слотов задержки, при которых вызывают беспокойство только конфликты ресурсов и конфликты данных. Обратите внимание на то, что у большинства инструкций x86, используемых в этой задаче, имеется по два операнда. Последний (как правило, второй) операнд инструкции показывает как значение данных первого источника, так и расположение итоговых данных. Если операции требуется значение данных второго источника, оно показывается другим операндом инструкции. Например, инструкция «sub (edx),eax» считывает данные из места памяти, указанного в регистре edx, вычитает это значение из значения регистра eax и помещает результат обратно в регистр eax.

	Инструкции x86	Трансляция в виде MIPS-кода

a	Label mov -4(esp), eax
	add (edx), eax
	
	mov eax, -4(esp)
	add 1, ecx
	add 4, edx
	cmp esi, ecx
	jl Label
		Label: lw $2,-4($sp)
		lw $3,0($4)
		add $2,$2,$3
		sw $2,-4($sp)
		addi $6,$6,1
		addi $4,$4,4
		slt $1,$6,$5
		bne $1,$0,Label

b	Label: add eax, (edx)
	
	mov eax, edx
	add 1, eax
	jl Label
		Label: lw $2,0($4)
		add $2,$2,$5
		sw $2,0($4)
		add $4 $5,$0
		addi $5,$5,1
		slt $1,$5,$0
		bne $1,$0,Label

4.31.1 [20] <4.11> Какой показатель CPI будет достигнут, если MIPS-версия этого цикла будет выполняться на процессоре со статической диспетчеризацией, который имеет конвейер из пяти стадий и запускает только одну инструкцию за один тактовый цикл?

4.31.2 [20] <4.11> Какой показатель CPI будет достигнут, если x86-версия этого цикла будет выполняться на процессоре со статической диспетчеризацией, который имеет конвейер из семи стадий и запускает только одну инструкцию за один тактовый цикл? Стадии конвейера называются IF, ID, ARD, MRD, EXE и WB. Стадии IF и ID аналогичны таким же стадиям в пятистадийном конвейере MIPS. На стадии ARD вычисляется адрес считываемого места в памяти, на стадии MRD выполняется чтение из памяти, на стадии EXE выполняется операция, а на стадии WB результат записывается в регистр или в память. Память данных имеет порт чтения (для инструкций на стадии MRD) и отдельный порт записи (для инструкций на стадии WB).

4.31.3 [20] <4.11> Какой показатель CPI будет достигнут, если x86-версия этого цикла будет выполняться на процессоре, который производит внутреннюю трансляцию этих инструкций в MIPS-подобные микрооперации, а затем выполняет эти микрооперации на пятистадийном конвейере со статической диспетчеризацией и запуском одной инструкции за один тактовый цикл. Учтите, что счетчик инструкций, используемый при вычислении CPI для этого процессора, является счетчиком инструкций x86.

4.31.4 [20] <4.11> Какой показатель CPI будет достигнут, если MIPS-версия этого цикла выполняется на процессоре с динамической диспетчеризацией и с запуском одной инструкции за один тактовый цикл? Предположим, что наш процессор не осуществляет переименование регистров, поэтому вы можете только изменить порядок следования инструкций, не имеющих зависимостей от данных.

4.31.5 [30] <4.10, 4.11> Предположим, что доступно большое количество свободных регистров. Выполните переименование регистров в MIPS-версии этого цикла для устранения как можно большего количества зависимостей от данных, существующих между инструкциями в одной и той же итерации цикла. А теперь

повторите упражнение 4.31.4, используя ваш новый код, подвергшийся переименованию регистров.

4.31.6 [20] <4.10, 4.11> Выполните упражнение 4.31.4 еще раз, но теперь при условии, что процессор присваивает новое имя результату каждой инструкции по мере декодирования этой инструкции, а затем переименовывает регистры, используемые следующими по порядку инструкциями, чтобы работать с правильными значениями регистров.

Упражнение 4.32

Для задач данного упражнения предполагается, что условные переходы составляют некоторую долю от всех выполняемых инструкций и имеют определенный показатель точности предсказания исхода условного перехода. Предположим, что процессор никогда не задерживает свою работу из-за зависимостей, связанных с данными и ресурсами, то есть процессор всегда извлекает и выполняет максимальное количество инструкций за цикл, если отсутствуют конфликты управления. Для зависимостей, связанных с управлением, процессор использует предсказание условных переходов и продолжает извлечение инструкций из предсказанного пути. Если условный переход был предсказан неправильно, когда разрешен исход условного перехода, инструкции, извлеченные после неверно предсказанного перехода, сбрасываются, и следующий цикл процессор начинает с извлечения инструкций из правильного пути.

	Условные переходы в процентном отношении ко всем выполняемым инструкциям	Точность предсказания условного перехода, %
а	20	90
б	20	99,5

4.32.1 [5] <4.11> Сколько инструкций предположительно будет выполнено между обнаружением одного неудачного предсказания условного перехода и следующим обнаружением такого же неудачного предсказания?

Для остальных задач данного упражнения предполагается, что при заданной глубине исход условного перехода обнаружился на заданной стадии конвейера (при счете, ведущемся от стадии 1):

	Глубина конвейера	Стадия, на которой стал известен исход условного перехода
а	12	10
б	25	18

4.32.2 [5] <4.11> Сколько инструкций условного перехода ожидаются в состоянии «прогресса» (уже извлечены, но еще не переданы) в каждый заданный момент времени в процессоре с параллельным запуском четырех инструкций, имеющем указанные параметры конвейера?

4.32.3 [5] <4.11> Сколько инструкций извлечено из неправильного пути для каждого неверно спрогнозированного условного перехода в процессоре с параллельным запуском четырех инструкций?

4.32.4 [10] <4.11> Каким будет ускорение, вызванное заменой процессора с расширением возможности по параллельному запуску с четырех до восьми инструкций? Предполагается, что процессор с восемью параллельно запускаемыми инструкциями отличается от процессора с четырьмя параллельно запускаемыми инструкциями только количеством инструкций, приходящихся на один цикл, а во всем остальном они идентичны (в глубине конвейера, стадии обнаружения исхода условного перехода и т. д.).

4.32.5 [10] <4.11> Каким будет ускорение работы, если условные переходы будут выполняться на одну стадию раньше в процессоре с параллельным запуском четырех инструкций?

4.32.6 [10] <4.11> Каким будет ускорение работы, если условные переходы будут выполняться на одну стадию раньше в процессоре с параллельным запуском восьми инструкций? Проанализируйте разницу между этим результатом и результатом, полученным при решении упражнения 4.32.5.

Упражнение 4.33

В данном упражнении исследуется влияние предсказания условного перехода на производительность глубоко конвейеризированного процессора с параллельным запуском инструкций. Задачи данного упражнения ссылаются на процессор со следующим количеством стадий конвейера и следующим количеством запускаемых в одном тактовом цикле инструкций:

	Глубина конвейера	Ширина параллельного запуска
а	10	4
б	25	2

4.33.1 [10] <4.11> Сколько портов чтения файла регистров должен иметь процессор, чтобы избежать любых конфликтов ресурсов при чтении регистров?

4.33.2 [10] <4.11> Каким ожидается улучшение производительности по сравнению с процессором с запуском одной инструкции за тактовый цикл, имеющим пятистадийный конвейер, при отсутствии неправильно предсказанных условных переходов и зависимостей от данных? Предположим, что продолжительность тактового цикла уменьшается пропорционально количеству стадий конвейера.

4.33.3 [10] <4.11> Выполните упражнение 4.33.2 еще раз, но теперь при условии, что у каждой выполняемой инструкции имеется зависимость данных по типу «чтение после записи» с инструкцией, выполняющейся сразу после нее. Можете предположить, что вам не нужны циклы задержки, то есть препровождение данных позволяет следующим друг за другом инструкциям выполняться в непрерывных циклах.

Для остальных трех задач данного упражнения, если задача не определена иначе, нужно принять в качестве условия некоторую статистику, касающуюся процентного отношения инструкций условного перехода, точности предсказания и потерь производительности из-за неточности предсказаний:

	Доля условных переходов среди всех выполняемых инструкций	Условные переходы выполняются на стадии	Точность предсказания, %	Снижение производительности, %
а	30	7	95	10
б	15	8	97	2

4.33.4 [10] <4.11> Какой процент от всех циклов полностью тратится на извлечение инструкций из неверных путей при заданной доле инструкций условных переходов и точности предсказания условных переходов? Показатель снижения производительности следует проигнорировать.

4.33.5 [20] <4.11> Какой должна быть точность предсказаний условных переходов, если мы хотим ограничить задержки, связанные с неправильно предсказанными условными переходами показателем, не превышающим данный процентный показатель, относящийся к идеальному (без задержек) времени выполнения? Заданный показатель точности предсказания следует проигнорировать.

4.33.6 [10] <4.11> Какой должна быть точность предсказаний условных переходов, если нам нужно получить ускорение 0,5 (половинное) относительно этого же процессора, но только с идеальным предсказателем условных переходов?

Упражнение 4.34

Данное упражнение разработано, чтобы помочь вам разобраться в смысле заблуждения «Конвейеризация — это просто» из раздела 4.12. Первые четыре задачи данного упражнения ссылаются на MIPS-инструкции:

	Инструкция	Интерпретация
а	add Rd,Rs,Rt	Рег[Rd]=Рег[Rs]+Рег[Rt]
б	lw Rt,Offs(Rs)	Рег[Rt]=Память[Рег[Rs]+Смещение]

4.34.1 [10] <4.13> Опишите конвейеризированный операционный блок, необходимый для поддержки только этой инструкции. Ваш операционный блок должен быть сконструирован исходя из предположения, что единственные выполняемые на нем инструкции будут экземплярами данной инструкции.

4.34.2 [10] <4.13> Опишите требования к блокам препровождения данных и обнаружения конфликтов для вашего операционного блока из упражнения 4.34.1.

4.34.3 [10] <4.13> Что должно быть сделано для поддержки исключений, связанных с неопределенной инструкцией в вашем операционном блоке из упражнения 4.34.1? Учтите, что исключение, связанное с неопределенной инструкцией,

...жно быть инициировано, как только процессор сталкивается с инструкцией ...ого другого типа.

Остальные две задачи данного упражнения также ссылаются на эту MIPS-...струкцию:

	Инструкция	Интерпретация
...	beq Rs, Rt, Label	Если Per[Rs] == Per[Rt] PC=PC+Смещение
...	and Rd, Rs, Rt	Per[Rd]= Per[Rs]& Per[Rt]

4.34.4 [10] <4.13> Опишите, как нужно расширить ваш операционный блок ...упражнения 4.34.1, чтобы он мог поддерживать также и эту инструкцию. Ваш ...сширенный операционный блок должен быть сконструирован для поддержки ...лько указанных двух инструкций.

4.34.5 [10] <4.13> Выполните упражнение 4.34.2 еще раз для вашего расширен...го операционного блока из упражнения 4.34.4.

4.34.6 [10] <4.13> Выполните упражнение 4.34.3 еще раз для вашего расширен...го операционного блока из упражнения 4.34.4.

Упражнение 4.35

Это упражнение должно помочь вам лучше разобраться со взаимоотношениями между конструкцией архитектуры набора инструкций и конвейеризацией. Для задач данного упражнения предполагается, что у нас имеется конвейеризированный процессор с параллельным запуском нескольких инструкций с определенными показателями: количество стадий конвейера, количество параллельных инструкций, запускаемых в одном тактовом цикле, стадия, в которой определяется исход условного перехода, и точность его предсказания:

	Глубина конвейера	Ширина запуска	Стадия выполнения условного перехода	Точность предсказания условного перехода	Процент отношения условных переходов к общему количеству инструкций
а	10	4	7	80%	20%
б	25	2	17	92%	25%

4.35.1 [5] <4.8, 4.13> Конфликты управления могут быть устранены путем добавления слотов задержки условных переходов. Сколько слотов задержки должно следовать за каждым условным переходом, если мы хотим исключить все конфликты управления в данном процессоре?

4.35.2 [10] <4.8, 4.13> Каким будет ускорение, достигаемое за счет использования четырех слотов замедления условных переходов для сокращения конфликтов управления в данном процессоре? Предположим, что зависимости данных между инструкциями отсутствуют и что все четыре слота задержки могут быть заполнены полезными инструкциями без увеличения количества выполняемых инструкций.

Чтобы облегчить ваши вычисления, можете также предположить, что неправильно предсказанные условные переходы приходятся всегда на последнюю инструкцию, которая должна быть извлечена в тактовом цикле, то есть в одной и той же стадии конвейера, на которой находится инструкция условного перехода, отсутствуют инструкции, извлеченные из неправильного пути.

4.35.3 [10] <4.8, 4.13> Выполните упражнение 4.35.2 еще раз, но теперь при условии, что 10% от выполняемых условных переходов имеют все четыре слота задержки заполненными полезными инструкциями, 20% имеют только три полезные инструкции в слотах задержки (четвертой является инструкция `nop`), 30% имеют только две полезные инструкции в слотах задержки и 40% не имеют полезных инструкций в своих слотах задержки.

Остальные три задачи в данном упражнении ссылаются на следующий цикл, написанный на языке C:

а	`for(i=0;i!=j;i++){`
	`  b[i]=a[i];`
	`}`

б	`for(i=0;a[i]!=a[i+1];i++){`
	`  c++;`
	`}`

4.35.4 [10] <4.8, 4.13> Транслируйте этот цикл на языке C в MIPS-инструкции, при условии, что наша архитектура набора инструкций требует одного слота задержки после каждой инструкции условного перехода. Попытайтесь по возможности заполнить слоты задержки инструкциями, не являющимися пустыми операциями (`nop`). Можете предположить, что переменные `a`, `b`, `c`, `i` и `j` хранятся в регистрах `$1`, `$2`, `$3`, `$4` и `$5`.

4.35.5 [10] <4.7, 4.13> выполните упражнение 4.35.4 еще раз для процессора, имеющего два слота задержки для каждого условного перехода.

4.35.6 [10] <4.10, 4.13> Сколько итераций для вашего цикла из упражнения 4.35.4 может быть «в полете» внутри конвейера вашего процессора? Итерация считается «в полете», когда хотя бы одна из ее инструкций была уже извлечена и еще не была передана.

Упражнение 4.36

Это упражнение предназначено для того что бы помочь вам лучше разобраться с последним заблуждением из раздела 4.12 «Недооценка особенностей набора инструкций может отрицательно сказаться на конвейеризации». Первые четыре задачи данного упражнения ссылаются на следующие новые MIPS-инструкции:

	Инструкция	Интерпретация
а	`lwinc Rt,Offset(Rs)`	`Per[Rt]= Память[Per[Rs]+Смещение]` `Per[Rs]= Per[Rs]+4`
б	`addr Rt,Offset(Rs)`	`Per[Rt]=Память[Per[Rs]+Смещение]+ Per[Rt]`

4.36.1 [10] <4.11, 4.13> Транслируйте эту инструкцию в микрооперации MIPS.

4.36.2 [10] <4.11, 4.13> Как бы вы изменили пятистадийный MIPS-конвейер путем добавления поддержки трансляции в микрооперации, необходимые для поддержки этой новой инструкции?

4.36.3 [20] <4.13> Если вы хотите добавить эту инструкцию к архитектуре набора инструкций MIPS, то расскажите, какие изменения конвейера (какие стадии, какие структуры на каких стадиях) необходимы для непосредственной (без микроопераций) поддержки этой инструкции.

4.36.4 [10] <4.13> Как часто по вашей оценке будет востребована эта инструкция? Считаете ли вы, что мы посчитаем оправданным добавление этой инструкции архитектуре набора команд MIPS?

Остальные две задачи данного упражнения касаются добавления к архитектуре набора команд новой инструкции `addm`. В этих задачах предполагается, что в процессоре, в который добавляется `addm`, имеется следующее распределение тактовых циклов в соответствии с тем, какая инструкция завершается в данном цикле (или какая задержка не дает инструкции завершиться):

	add	beq	lw	sw	addm	Задержки, связанные с управлением	Задержки, связанные с данными
a	35%	20%	20%	10%	5%	5%	5%
б	25%	10%	25%	10%	10%	10%	10%

4.36.5 [10] <4.13> С учетом данного распределения циклов выполнения в процессоре с непосредственной поддержкой инструкции `addm` определите ускорение, достигаемое заменой этой инструкции на последовательность из трех инструкций (`lw`, `add` и затем `sw`). Предположим, что инструкция `addm` каким-то (магическим) образом поддерживается в классическом пятистадийном конвейере без возникновения конфликтов управления.

4.36.6 [10] <4.13> Выполните упражнение 4.36.5 еще раз, но теперь при условии, что `addm` была поддержана с добавлением стадии конвейера. При трансляции `addm` эта дополнительная стадия может быть удалена, и в результате будет исключена половина существующих задержек данных. Учтите, что устранение задержек данных применяется только к тем задержкам, которые существовали до трансляции инструкции `addm`, а не к тем задержкам, которые добавились самой трансляцией `addm`.

Упражнение 4.37

В данном упражнении исследуются некоторые побочные эффекты, присущие конвейеризации, такие как продолжительность тактового цикла и использование ресурсов оборудования. Первые три задачи данного упражнения ссылаются на MIPS-код, написанный с учетом того, что процессор не использует слоты задержки:

а	lw $1,40($6)
	beq $1,$0,Label ; Предполагается, что $1 = = $0
	sw $6,50($1)
	Label: add $2,$3,$1
	sw $2,50($1)

б	lw $5,-16($5)
	sw $5,-16($5)
	lw $5,-20($5)
	beq $5,$0,Label ; Предполагается, что $5!=$0
	add $5,$5,$5

4.37.1 [5] <4.3, 4.14> Какие части основного однотактного процессора используются всеми этими инструкциями? Какие части используются меньше всего?

4.37.2 [10] <4.6, 4.14> Какова востребованность порта чтения и порта записи блока памяти данных?

4.37.3 [10] <4.6, 4.14> Предположим, что у нас уже имеется однотактная конструкция. Сколько разрядов нам понадобится для регистров конвейера для реализации конвейеризированной конструкции?

Остальные три задачи данного упражнения предполагают, что компоненты операционного блока имеют следующие показатели латентности:

	Память инструкций (I-Mem)	Сумматор (Add)	Мультиплексор (Mux)	АЛУ (ALU)	Файл регистров (Regs)	Память данных (D-Mem)	Расширитель знака (Sign-extend)	Блок сдвига влево на два (Shift-left-2)
а	400 пс	100 пс	30 пс	120 пс	200 пс	350 пс	20 пс	0 пс
б	500 пс	150 пс	100 пс	180 пс	220 пс	1000 пс	90 пс	20 пс

4.37.4 [10] <4.3, 4.5, 4.14> Сравните продолжительности тактовых циклов для однотактового и конвейеризированного пятистадийного операционного блока, используя эти показатели латентности отдельных элементов.

4.37.5 [10] <4.3, 4.5, 4.14> выполните упражнение 4.37.4 еще раз, но теперь при условии, что нам нужна лишь поддержка инструкций ADD.

4.37.6 [20] <4.3, 4.5, 4.14> Если сокращение латентности одного компонента операционного блока на 1 пс обходится в один доллар, то какой будет стоимость сокращения продолжительности тактового цикла на 20% в однотактной и в конвейеризированной конструкции?

Упражнение 4.38

В данном упражнении исследуются энергоэффективность и ее взаимоотношения с производительностью. Задачи данного упражнения предполагают использование некоторых показателей: расход энергии на работу памяти инструкций, файл регистров и память данных. Можете предположить, что все остальные компоненты операционного блока потребляют незначительное количество энергии.

	Память инструкций (I-Mem)	Чтение одного регистра	Запись регистра	Чтение памяти данных	Запись памяти данных
а	100 пДж	60 пДж	70 пДж	120 пДж	100 пДж
б	200 пДж	90 пДж	80 пДж	300 пДж	280 пДж

4.38.1 [10] <4.3, 4.6, 4.14> Сколько энергии тратится на выполнение инструкции add в одногактной конструкции и в конструкции с конвейером, состоящим из пяти стадий?

4.38.2 [10] <4.6, 4.14> Какая MIPS-инструкция имеет наихудший показатель энергопотребления и сколько энергии тратится на ее выполнение?

4.38.3 [10] <4.6, 4.14> Если главная цель состоит в снижении энергопотребления, то как вы бы изменили конструкцию конвейера? Насколько сократятся в процентах энергозатраты на выполнение инструкции lw после этих изменений?

Для остальных трех задач данного упражнения предполагается, что компоненты операционного блока имеют следующие показатели латентности. Можете предположить, что другие компоненты операционного блока имеют незначительные показатели латентности.

	Память инструкций (I-MEM)	Блок управления (Control)	Чтение или запись регистра	АЛУ	Чтение или запись памяти данных (D-Mem)
а	400 пс	300 пс	200 пс	120 пс	350 пс
б	500 пс	400 пс	220 пс	180 пс	1000 пс

4.38.4 [10] <4.6, 4.14> Каково будет влияние ваших изменений из упражнения 4.38.3 на производительность?

4.38.5 [10] <4.6, 4.14> Мы можем убрать сигнал MemRead и заставить память данных считываться при каждом тактовом цикле, например мы можем постоянно иметь MemRead=1. Объясните, почему процессор по-прежнему работает правильно после такого изменения. Каково влияние этого изменения на тактовую частоту и на энергопотребление?

4.38.6 [10] <4.6, 4.14> Если неактивный блок потребляет 10% от мощности, затрачиваемой в активном состоянии, то какую энергию потребляет память инструкций в каждом тактовом цикле? Какой процент от всей энергии, потребляемой памятью инструкций, приходится на эту энергию холостого режима?

Упражнение 4.39

В задачах данного упражнения предполагается, что в процессе выполнения программы циклы процессора «тратятся» следующим образом: цикл «тратится» на инструкцию, если процессор завершает выполнение этого типа инструкции в данном тактовом цикле; цикл «тратится» на задержку, если процессор не может завершить выполнение инструкции в данном цикле по причине задержки.

	add	beq	lw	sw	Задержки, связанные с управлением	Задержки, связанные с данными
а	35%	20%	20%	10%	10%	5%
б	25%	10%	25%	10%	20%	10%

Для задач данного упражнения также предполагается, что отдельные стадии конвейера имеют некоторые показатели латентности и энергопотребления. Стадия расходует эту энергию в целях выполнения своей работы в пределах заданной латентности. Учтите, что энергия не тратится на стадии MEM при выполнении цикла, в котором нет обращения к памяти. Точно так же энергия не тратится на стадии WB stage в том цикле, в котором не осуществляется запись в файл регистров. В некоторых задачах делается предположение о том, как энергопотребление изменяется, если стадия выполняет свою работу медленнее или быстрее, чем показано в данной таблице.

	IF	ID	EX	MEM	WB
а	300 пс / 120 пДж	400 пс / 60 пДж	350 пс / 75 пДж	500 пс / 130 пДж	100 пс / 20 пДж
б	200 пс / 150 пДж	150 пс / 60 пДж	120 пс / 50 пДж	190 пс / 150 пДж	140 пс / 20 пДж

4.39.1 [10] <4.14> Определите производительность (в инструкциях в секунду)?

4.39.2 [10] <4.14> Определите рассеиваемую мощность в ваттах (джоулях в секунду)?

4.39.3 [10] <4.6, 4.14> Работу каких стадий конвейера можно замедлить и на сколько, чтобы это не повлияло на продолжительность тактового цикла?

4.39.4 [20] <4.6, 4.14> Зачастую можно пожертвовать некоторой скоростью в работе схемы в целях сокращения ее энергопотребления. Предположим, что мы можем сократить энергопотребление в X раз (новое энергопотребление составляет $1/X$ от старого энергопотребления), если увеличим латентность в X раз (новая латентность в X раз больше старой). Как можно настроить показатели латентности стадий конвейера для минимизации энергопотребления, не жертвуя при этом производительностью? Выполните упражнение 4.39.2 еще раз для этого перенастроенного процессора.

4.39.5 [10] <4.6, 4.14> Выполните упражнение 4.39.4 еще раз, но теперь задайтесь целью минимизировать затраты энергии на каждую инструкцию при повышении продолжительности тактового цикла, но не более чем на 10%.

4.39.6 [10] <4.6, 4.14> Выполните упражнение 4.39.5, но теперь при условии, что энергопотребление сокращено в X^2 раза, при возросшей в X раз латентности. Какой будет экономия энергии по сравнению с той, что была вычислена при выполнении упражнения 4.39.2?

Ответы на вопросы самопроверок

Раздел 4.1: 3 из 5: Блок управления, операционный блок, память. Отсутствуют ввод и вывод.

Раздел 4.2: Это утверждение ложное. Элементы памяти, управляемые фронтом синхроимпульса, позволяют одновременно осуществлять чтение и запись без возникновения конфликтов.

Раздел 4.3: *I*. а. II: в.

Раздел 4.4: Да, сигналы Branch и ALUOp0 являются идентичными. Кроме этого, сигналы MemtoReg и RegDst являются инверсными по отношению к друг другу. Инвертор не нужен; просто нужно использовать другой сигнал и поменять местами входы мультиплексора!

Раздел 4.5: 1. Задержка на результате выполнения инструкции `lw`. 2. Препровождение записи результата первой инструкции `add`, записываемого в регистр `$t1`. 3. Никаких задержек или препровождений данных не требуется.

Раздел 4.6: Утверждения 2 и 4 являются правильными, а остальные утверждения не соответствуют истине.

Раздел 4.8: 1. Прогнозирование, что переход не состоится. 2. Прогнозирование, что переход состоится. 3. Динамическое прогнозирование.

Раздел 4.9: В первой инструкции, поскольку, по логике, она выполняется перед всеми остальными.

Раздел 4.10: 1. К обоим подходам. 2. К обоим подходам. 3. К программным подходам. 4. К аппаратным подходам. 5. К аппаратным подходам. 6. К аппаратным подходам. 7. К обоим подходам. 8. К аппаратным подходам. 9. К обоим подходам.

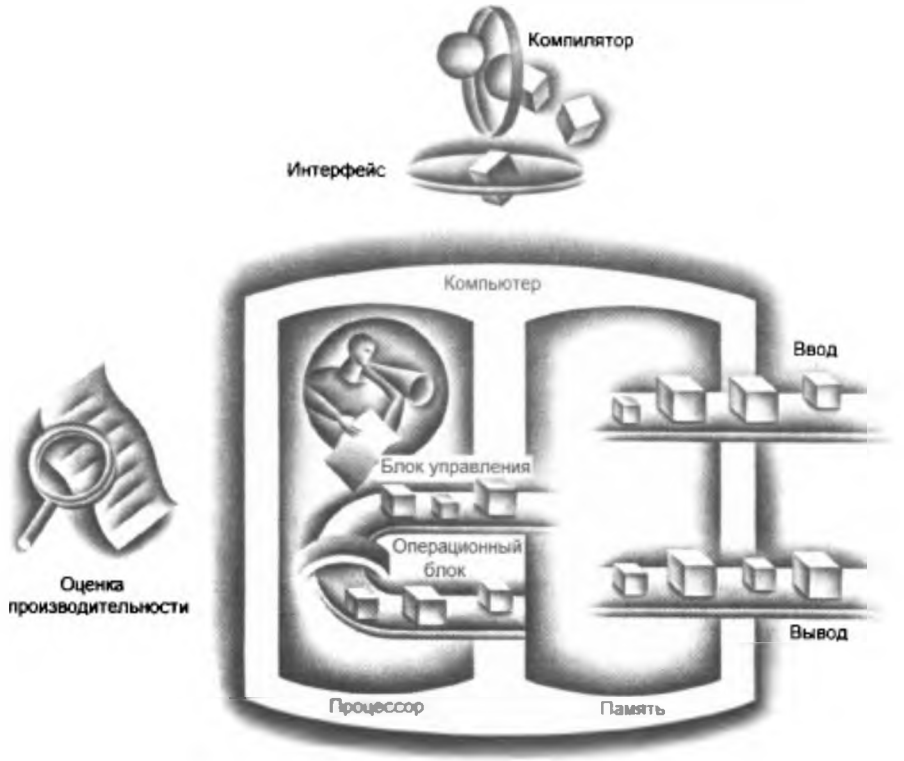
Раздел 4.11: Первые два утверждения являются ложными, а последние два — истинными.

Глава 5

Объемная и быстродействующая: анализ иерархии памяти

В идеале желательно получить бесконечно большой объем памяти, и чтобы любое конкретное ... слово было моментально доступно ... Мы ... вынуждены признать возможность построения иерархии устройств памяти, каждое из которых имеет больший объем, чем предыдущее, но при менее быстром доступе к нему.

А. В. Буркс, Х. Х. Гольдштейн и Дж. фон Нейман



5.1. Введение

С самых первых дней существования компьютерной техники программистам хотелось иметь неограниченные объемы быстродействующей памяти. Темы, рассматриваемые в данной главе, помогают программистам в воплощении этой иллюзии в жизнь. Перед тем как приступить к созданию этой иллюзии, давайте рассмотрим простую аналогию, которая послужит иллюстрацией используемых нами ключевых принципов и механизмов.

Предположим, что вам приходилось в студенческие времена писать курсовую работу о важных исторических событиях в мире компьютерного оборудования. Вы сидите за столом в библиотеке с подборкой книг, которые вы сняли с полок и изучаете. Оказалось, что часть важных компьютеров, о которых нужно было написать, описаны в имеющихся у вас книгах, но в них нет никаких сведений о компьютере EDSAC. Поэтому вы возвращаетесь к полкам и ищете еще одну книгу. Вы нашли книгу о первых английских компьютерах, в которой есть сведения о EDSAC. Как только у вас на столе окажется хорошая подборка литературы, возникнет высокая вероятность, что многое о том, что вам нужно, будет найдено в этих книгах, и основная часть времени будет затрачена на работу.

Такой же принцип позволяет нам создать иллюзию большого объема памяти, к которой можно обращаться так же быстро, как и к очень маленькому по объему устройству памяти. Точно так же, как у вас не возникает потребность просмотра сразу всех книг в библиотеке, программа не обращается одновременно с равной вероятностью ко всему своему коду или данным. В противном случае было бы невозможно обеспечить быстрый доступ к основному объему памяти и при этом оснащать компьютеры большими объемами памяти, точно так же, как вам было бы невозможно поместить все книги, имеющиеся в библиотеке, на вашем столе и сохранить возможность быстрого поиска нужного материала.

В обоих случаях, и при работе в библиотеке, и в работе программы, в основу кладется *принцип локальности*. Этот принцип гласит, что программы в любой момент времени обращаются к относительно небольшой части своего адресного пространства, точно так же, как вы обращаетесь к очень маленькой части библиотечной коллекции. Существуют две разновидности локальности:

- **Локальность, связанная со временем:** если было обращение к элементу, то есть вероятность, что к нему скоро последует новое обращение. Если вы недавно приносили книгу на свой стол для просмотра, то, возможно, совсем скоро она понадобится вам еще раз.
- **Локальность, связанная с пространством:** если было обращение к элементу, то, скорее всего, скоро будет обращение и к тем элементам, чьи адреса находятся поблизости.

Локальность, связанная со временем
Принцип, в котором утверждается, что если было обращение к элементу данных, то высока вероятность, что к нему скоро последует новое обращение.

Локальность, связанная с пространством
Принцип локальности, в котором утверждается, что если было обращение к элементу данных, то высока вероятность, что скоро последует обращение и к элементам данных по соседним адресам.

Например, когда вы нашли книгу по первым английским компьютерам для поиска материалов по EDSAC, вы при этом заметили, что рядом с ней на полке стояла еще одна книга по первым механическим компьютерам, и вы захватили с собой еще и эту книгу, и позже нашли в ней кое-что полезное для вас. В библиотеках книги по сходной тематике соседствуют друг с другом, повышая показатель локальности, связанной с пространством. Как локальность, связанная с пространством, используется в иерархии памяти, мы увидим в данной главе чуть позже.

Точно так же, как обращение к книгам, лежащим на столе, является естественным проявлением локальности, в программах локальность возникает благодаря простым и естественным программным структурам. Например, многие программы содержат циклы, поэтому инструкции и данные, к которым, скорее всего, последует повторное обращение, демонстрируют локальность, связанную со временем. Поскольку обычно используется последовательное обращение к инструкциям, программы также демонстрируют высокую локальность, связанную с пространством. Обращения к данным также являются естественным проявлением локальности, связанной с пространством. Например, у последовательного обращения к элементам массива или к записи вполне естественной будет высокая степень локальности, связанной с пространством.

Принцип локальности успешно используется путем реализации памяти компьютера в виде **иерархии памяти**. Эта иерархия состоит из нескольких уровней памяти с разными скоростями и объемами. Более быстродействующие устройства памяти имеют более высокий показатель стоимости бита, чем менее быстродействующие устройства памяти, поэтому они имеют меньший объем.

Сегодня для построения иерархии памяти используются три основные технологии. Оперативная память выполнена из динамических запоминающих устройств с произвольной выборкой — DRAM (dynamic random access memory), а уровни, находящиеся ближе к процессору (кэш-память), используют статические запоминающие устройства с произвольной выборкой — SRAM (static random access memory). У DRAM-памяти меньше стоимость бита, чем у SRAM, хотя она существенно медленнее. Разница в стоимости обусловлена тем, что в DRAM используется значительно меньшая площадь на один бит, и у DRAM-элементов более высокая емкость; разница в скорости обусловлена сразу несколькими факторами. Третьей технологией, используемой для реализации наиболее объемного и самого медленного уровня иерархии, обычно является магнитный диск. (Во многих встроенных устройствах вместо дисков используется флэш-память; см. раздел 6.4.) Время доступа к биту данных и его стоимость широко варьируются в зависимости от того, какая из этих технологий применяется. Эти показатели, применительно к значениям 2008 года, показаны в следующей таблице.

Иерархия памяти
Структура, использующая несколько уровней памяти; по мере удаления от процессора увеличивается объем памяти и время доступа к ней.

Технология памяти	Обычное время доступа, нс	Цена в долларах за 1 Гбайт в 2008 году
SRAM	0,5–2,5	2000–5000
DRAM	50–70	20–75
Магнитный диск	5 000 000–20 000 000	0,20–2

Из-за различий в стоимости и времени доступа предпочтительнее строить память в виде иерархии уровней. На рис. 5.1 показано, что самая быстродействующая память находится ближе к процессору, а медленная, менее дорогая память, находится ниже его. Цель состоит в том, чтобы снабдить пользователя как можно большим объемом по самой дешевой технологии, обеспечивая ему доступ на скорости, предлагаемой самой быстродействующей памятью.

Рис. 5.1. Основная структура иерархии памяти. За счет иерархической реализации системы памяти у пользователя создается иллюзия, что она по объему соответствует самому большому уровню иерархии, но может быть доступна, как будто изготовлена из самых быстродействующих элементов. Флэш-память заменила диски во многих встраиваемых устройствах и может привести к появлению нового уровня в иерархии запоминающих устройств для настольных компьютеров и серверов; см. раздел 6.4

Данные имеют аналогичную иерархию: тот уровень, который ближе к процессору, обычно является поднабором любого, более отдаленного уровня, а все данные хранятся на самом нижнем уровне. По аналогии, книги на вашем столе составляют поднабор библиотеки, в которой вы работаете, которая, в свою очередь, является поднабором всех библиотек на территории университета. Более того, по мере удаления от процессора доступ к уровням постепенно становятся более длительным, точно так же, как это может быть в иерархии университетских библиотек.

Иерархия памяти может состоять из нескольких уровней, но данные единовременно копируются только между двумя смежными уровнями, поэтому мы можем сконцентрировать свое внимание только на двух уровнях. Самый верхний уровень — тот, что ближе всех к процессору, — меньше и быстрее расположенного ниже уровня, поскольку на более высоком уровне используется более дорогостоящая технология. На рис. 5.2 показано, что минимальная порция информации, которая может либо присутствовать, либо не присутствовать в двухуровневой иерархии, называется **блоком** или **строкой**; в нашей библиотечной аналогии блок информации соответствует книге.

Если данные, запрашиваемые процессором, имеются в каком-нибудь блоке на верхнем уровне, то это называется *попаданием* (по аналогии с тем, как если бы вы нашли информацию в одной из книг на своем столе). Если данные не найдены на верхнем уровне, запрос называется *промахом* (неудачей). Затем следует обращение к более низкому уровню для извлечения блока, содержащего запрошенные данные. (Продолжая нашу аналогию, вы идете от стола к полкам, чтобы найти нужную книгу.) **Коэффициент попаданий**, или *результативность*, является долей обращений к памяти, удовлетворенных на верхнем уровне; он часто используется в качестве показателя производительности иерархии памяти. **Коэффициент промахов** (1 – коэффициент попаданий) является долей обращений к памяти, которые не были удовлетворены на верхнем уровне.

Поскольку иерархия памяти главным образом создавалась ради повышения производительности, важная роль уделяется времени обслуживания попаданий и промахов. **Время попадания** затрачивается на доступ к верхнему уровню иерархии памяти, в него включается время, необходимое для определения, будет ли обращение попаданием или промахом (то есть время, необходимое для просмотра всех книг на столе). Время промаха, или **издержка**, складывается из времени на замену блока в верхнем уровне соответствующим блоком из нижнего уровня, сюда же прибавляется и время на доставку этого блока процессору (время, затрачиваемое на то, чтобы взять с полки еще одну книгу и положить ее на стол). Поскольку верхний уровень меньше по объему и построен из элементов памяти с более высоким быстродействием, время попадания будет намного меньше, чем время доступа к следующему уровню иерархии, которое является основной составляющей издержки. (Время на изучение книг на столе намного меньше времени, необходимого на то, чтобы встать и принести с полки новую книгу.)

Как будет показано в данной главе, концепция, использованная для создания систем памя-

Блок (или строка)
Минимальная порция информации, которая может либо присутствовать, либо не присутствовать в кэш-памяти.

Коэффициент попаданий
Доля обращений к памяти, удовлетворенных на верхнем уровне ее иерархии.

Коэффициент промахов
Доля обращений к памяти, не удовлетворенных на верхнем уровне ее иерархии.

Время попадания
Время, необходимое для доступа к уровню иерархии памяти, включая и время, необходимое для определения попадания или промаха.

Издержка промаха
Время, необходимое для извлечения блока на уровень иерархии памяти из нижнего уровня, включая время доступа к блоку, передачи его с одного уровня на другой, вставки его в уровень, обращение к которому было неудачным, и затем передачи блока инициатору запроса.

...оказывает влияние на многие другие аспекты компьютера, включая и то, как операционная система управляет памятью и вводом-выводом, как компиляторы генерируют код и даже то, как приложения используют возможности компьютера. Разумеется, поскольку все программы тратят много времени на доступ к памяти, система памяти неизменно является главным фактором в определении производительности. Зависимость достижения высокой производительности от иерархии памяти означает, что программисты, привыкшие думать о памяти, как о неком одномерном запоминающем устройстве с произвольным доступом, теперь для получения высокой производительности должны понимать, что память представляет собой сложную иерархию. Вся важность такого представления о системе памяти будет показана на следующих примерах, в частности она будет продемонстрирована на рис. 5.15.

Рис. 5.2. Каждую пару уровней в иерархии памяти можно представлять в виде верхнего и нижнего уровней. В пределах каждого уровня порция присутствующей или отсутствующей в нем информации называется блоком или строкой. Обычно при копировании каких-либо данных между уровнями передается сразу весь блок

Поскольку системы памяти играют важную роль для достижения высокой производительности, разработчики компьютеров уделяют большое внимание этим системам и разрабатывают сложные механизмы для повышения производительности их работы. В данной главе будут рассмотрены основные концептуальные идеи, но при этом будет использоваться множество упрощений и абстракций, чтобы не усложнять материал.

Общее представление

Программы демонстрируют как локальность, связанную со временем, то есть тенденцию к повторному использованию недавно запрошенных элементов данных, так и локальность, связанную с пространством, то есть тенденцию обращения к элементам данных, находящимся по соседству с недавно запрошенными элементами данных. Иерархии памяти используют локальность, связанную со временем, располагая последние запрошенные данные ближе к процессору. А локальность,

связанная с пространством, используется ими путем перемещения блоков, состоящих из нескольких смежных слов в памяти, на более высокие уровни иерархии.

На рис. 5.3 показано, что ближе к процессору иерархии памяти используют менее объемные и более быстродействующие технологии хранения данных. Таким образом, обращение, удовлетворяемое на самом высоком уровне иерархии, может быть обработано быстрее. Неудовлетворенные обращения перенаправляются на нижние уровни иерархии, у которых больше объем, но ниже быстродействие. Если коэффициент попадания достаточно высок, иерархия памяти обладает эффективным временем доступа, приближающимся ко времени, характерному для наивысшего (и самого быстродействующего) уровня, и к объему, соизмеримому с тем, который характерен для самого низкого (и самого объемного) уровня.

Фактически большинство систем имеют память с истинно иерархической структурой, это означает, что данные не могут присутствовать на уровне i, пока они не будут также присутствовать и на уровне $i + 1$.

Рис. 5.3. На этой схеме показана структура иерархии памяти: по мере увеличения расстояния от процессора растет и объем. Эта структура с соответствующими рабочими механизмами позволяет процессору иметь время доступа, которое определяется главным образом уровнем *1* иерархии, и при этом обладать памятью с объемом, соответствующим уровню *n*. Воплощение в жизнь этой иллюзии и является темой данной главы. Хотя в нижней части иерархии обычно находится локальный диск, в некоторых системах в качестве следующих уровней иерархии используется магнитная лента или файловый сервер, доступный по локальной сети

Самопроверка

Какие из следующих утверждений в целом соответствуют действительности?

1. Кэш-память использует локальность, связанную со временем.
2. При чтении возвращаемое значение зависит от того, какой блок находится в кэш-памяти.
3. Основная стоимость иерархии памяти приходится на самый верхний уровень.
4. Основной объем иерархии памяти приходится на самый нижний уровень.

5.2. Основы кэш-памяти

Кэш: укромное место, позволяющее прятать или хранить вещи.

«Webster's New World Dictionary of the American Language», Third College Edition, 1988 г.

В нашем примере с библиотекой стол работает как кэш — укромное место для хранения вещей (книг), которые нужно изучить. Название *кэш* было выбрано для представления дополнительного уровня иерархии памяти между процессором и оперативной памятью в первом коммерческом компьютере. Кэш является простой заменой устройств памяти в операционном блоке, рассмотренном в главе 4. Но в наши дни употребление слова *кэш* в данном смысле стало доминирующим, этот термин используется также для ссылки на любые устройства хранения информации, предназначенные для использования принципа локальности доступа. Сначала кэш-память появилась на исследовательских компьютерах начала 1960-х годов, а чуть позже, в том же десятилетии, она появилась и на компьютерах, предназначенных для управления производством; сегодня кэш-память является атрибутом всех универсальных компьютеров, от серверов и до маломощных встроенных процессоров.

В данном разделе сначала будет рассмотрена очень простая кэш-память, в которой процессор делает пословные запросы, а блоки также состоят из одного слова. (Читатели, уже знакомые с основами кэш-памяти, могут раздел 5.2 пропустить.) На рис. 5.4 эта простая кэш-память показана до и после запроса элемента данных, который изначально в ней отсутствовал. Перед запросом в кэш-памяти находилась коллекция данных, на которые были последние ссылки $X_1, X_2, \ldots, X_{n-1}$, а процессор запросил слово X_n, которого в кэш-памяти нет. Этот запрос оказался неудачным, и слово X_n было перенесено из оперативной памяти в кэш.

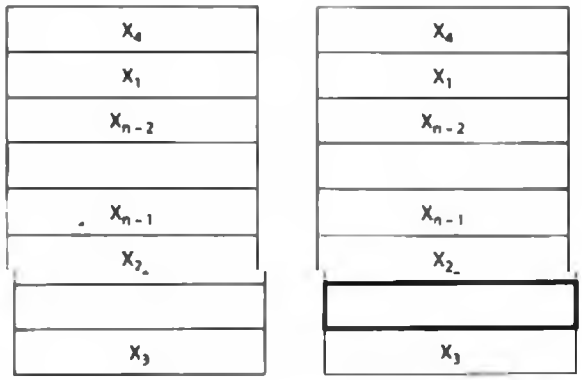
При рассмотрении сценария на рис. 5.4 возникают два вопроса. Как мы узнаем, что элемент данных находится в кэш-памяти? И потом, если он там, то как мы его найдем? Ответы на эти вопросы взаимосвязаны. Если каждое слово может попадать во вполне определенное одно и то же место в кэш-памяти, то найти слово, находящееся в кэш-памяти, будет нетрудно. Наиболее простым способом определения места в кэш-памяти для каждого слова является определение места в кэш-памяти на основе *адреса* слова в памяти. Такая структура кэша называется **непосредственно отображенной**, поскольку каждое место в памяти отображается непосредственно на одно место в кэш-памяти. Обычное отображение между адресами и местами в кэше для кэш-памяти с непосредственным отображением, как правило, устроено довольно просто. Например, почти во всех устройствах кэш-памяти с непосредственным отображением для поиска блока используется следующее проецирование:

(Адрес блока) modulo (Количество блоков в кэш-памяти)

Если количество записей в кэш-памяти является степенью числа 2, то модуль (modulo) может быть вычислен просто за счет использования младших $\log_2$ (размер кэш-памяти в бло-

Кэш-память с непосредственным отображением
Структура кэша, в которой каждое место в памяти проецируется на одно конкретное место в кэш-памяти.

ках) разрядов адреса. Таким образом, кэш-память из восьми блоков использует три самых младших разряда ($8 = 2^3$) адреса блока. Например, на рис. 5.5 показано, как память с адресами между 1_{10} (00001_2) и 29_{10} (11101_2) отображается на места 1_{10} (001_2) и 5_{10} (101_2) в кэш-памяти с непосредственным отображением, состоящей из восьми слов.



а) до ссылки на X_n б) после ссылки на X_n

Рис. 5.4. Кэш-память непосредственно перед и сразу после обращения к слову X_n, изначально отсутствующему в кэш-памяти. Это обращение становится причиной промаха, заставляющего кэш-память извлечь X_n из памяти и вставить в себя этот элемент

Поскольку в каждом месте кэш-памяти может находиться содержимое из разных мест памяти, то как мы узнаем, соответствуют данные запрошенному слову или нет? То есть как мы узнаем, имеется ли запрошенное слово в кэш-памяти или нет? Ответ на этот вопрос будет дан за счет добавления к кэш-памяти **тегов**. Тег содержит адресную информацию, необходимую для идентификации соответствия слова в кэш-памяти запрашиваемому слову. В теге нужно содержать лишь старшую часть адреса, соответствующую разрядам, не используемым в качестве индекса в кэш-памяти. Например, на рис. 5.5 нам нужны в теге только старшие два из пяти адресных разрядов, поскольку младшие три разряда составляют индексное поле адреса, выбирающее блок. Разработчики опускают разряды индекса, потому что они избыточны, поскольку по определению индексное поле любого адреса блока кэш-памяти должно быть номером этого блока.

Тег
Поле в таблице, используемой для иерархии памяти, которое содержит адресную информацию, необходимую для определения, соответствует ли связанный блок в иерархии запрошенному слову.

Бит достоверности
Поле в таблицах иерархии памяти, показывающее, что связанный с ним блок в иерархии содержит достоверные данные.

Нам также нужен способ определения того, что блок кэш-памяти не содержит достоверной информации. Например, когда процессор начинает работу, кэш-память не содержит достоверных данных, и поля тегов не будут иметь никакого смысла. Даже после выполнения множества инструкций некоторые из записей кэш-памяти могут все еще быть пустыми, как показано на рис. 5.4. Таким образом, нам нужно знать, что тег для таких записей должен быть

игнорирован. Наиболее распространенным является метод добавления **бита** верности, показывающего, содержит ли запись достоверный адрес. Если раз- не установлен, то для этого блока не может быть соответствия.

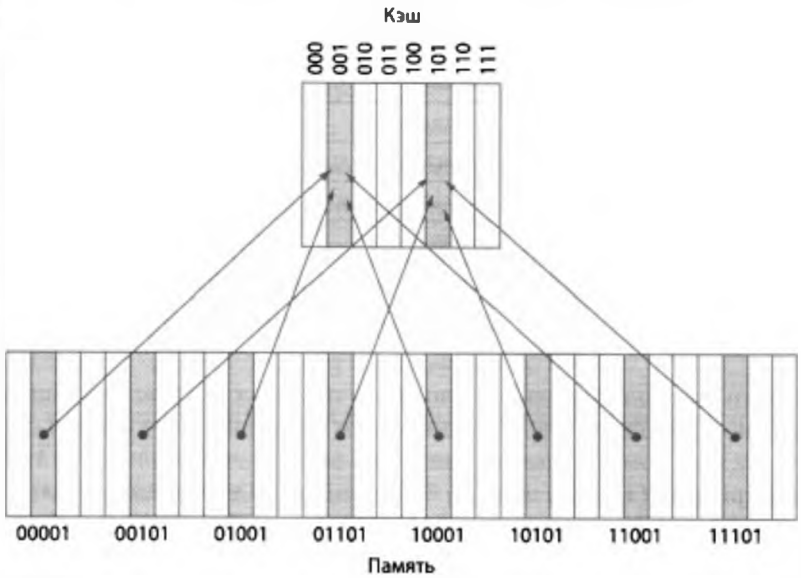


Рис. 5.5. Кэш-память с непосредственным отображением, имеющая восемь записей, с указанием адресов слов памяти в диапазоне от 0 до 31, отображаемых на одни и те же места в кэш-памяти. Поскольку в кэш-памяти восемь слов, адрес X проецируется на имеющееся в кэш-памяти с непосредственным отображением слово X по модулю 8. То есть младшие $\log_2(8) = 3$ разряда используются в качестве индекса кэш-памяти. Таким образом, адреса 00001_2, 01001_2, 10001_2 и 11001_2 проецируются на запись 001_2 кэш-памяти, а адреса 00101_2, 01101_2, 10101_2 и 11101_2 проецируются на запись 101_2 кэш-памяти

В остальной части данного раздела основное внимание будет уделено объяснению того, как кэш-память справляется с чтением. В целом, справиться с чтением немного легче, чем с записью, поскольку чтение не должно изменять содержимое кэш-памяти. После изучения основ работы чтения и обработки промахов при обращениях к кэш-памяти мы рассмотрим конструкции кэш-памяти для настоящих компьютеров и займемся вопросом о том, как эти конструкции справляются с записью.

Обращение к кэш-памяти

Ниже показана последовательность из девяти обращений к памяти, приходящихся на пустую кэш-память, состоящую из восьми блоков, включая действие, предпринимаемое при каждом обращении. На рис. 5.6 показано, как содержимое кэш-памяти изменяется после каждого промаха. Поскольку в кэш-памяти восемь блоков, самые младшие три разряда адреса дают номер блока:

Десятичный адрес обращения	Двоичный адрес обращения	Попадание или промах	Заданный блок кэш-памяти (где найдены или куда помещены данные)
22	10110_2	промах (5.6, *б*)	$(10\mathbf{110}_2 \bmod 8) = \mathbf{110}_2$
26	11010_2	промах (5.6, *в*)	$(11\mathbf{010}_2 \bmod 8) = \mathbf{010}_2$
22	10110_2	попадание	$(10\mathbf{110}_2 \bmod 8) = \mathbf{110}_2$
26	11010_2	попадание	$(11\mathbf{010}_2 \bmod 8) = \mathbf{010}_2$
16	10000_2	промах (5.6, *г*)	$(10\mathbf{000}_2 \bmod 8) = \mathbf{000}_2$
3	00011_2	промах (5.6, *д*)	$(10\mathbf{011}_2 \bmod 8) = \mathbf{011}_2$
16	10000_2	попадание	$(10\mathbf{000}_2 \bmod 8) = \mathbf{000}_2$
18	10010_2	промах (5.6, *е*)	$(10\mathbf{010}_2 \bmod 8) = \mathbf{010}_2$
16	10000_2	попадание	$(10\mathbf{000}_2 \bmod 8) = \mathbf{000}_2$

Из-за того, что кэш-память не заполнена, несколько первых обращений будут неудачными; подписи на рис. 5.6 описывают действия для каждого обращения к памяти. На восьмом обращении для блока возникает конфликтное требование. Слово по адресу 18 (10010_2) должно быть помещено в блок кэш-памяти 2 (010_2). Следовательно, оно должно заменить слово из адреса 26 (11010_2), которое уже находится в блоке кэш-памяти 2 (010_2). Такое поведение позволяет кэш-памяти воспользоваться локальностью, связанной со временем: слова, обращение к которым состоялось недавно, заменяют те слова, обращение к которым было чуть раньше.

Эта ситуация является прямой аналогией с необходимостью наличия книги с полки и необходимостью дополнительного места на столе — какая-то книга, которая уже на нем лежит, должна быть возвращена на полку. В кэш-памяти с непосредственным отображением есть только одно место, куда можно поместить только что запрошенный элемент, и, следовательно, только один вариант замены.

Нам известно, куда обращаться в кэш-памяти для каждого возможного адреса: самые младшие разряды адреса могут быть использованы для поиска уникальной записи в кэш-памяти, на которую мог быть спроецирован адрес. На рис. 5.7 показано, как адрес, к которому идет обращение, делится на:

- *поле тега*, которое используется для сравнения со значением поля тега кэш-памяти;
- *индекс кэш-памяти*, который используется для выбора блока.

Индекс блока кэш-памяти наряду с содержимым тега этого блока уникальным образом определяют адрес в памяти того слова, которое содержится в блоке кэш-памяти. Поскольку поле индекса используется в качестве адреса для обращения к кэш-памяти и поскольку поле, состоящее из n разрядов, имеет 2^n значений, общее количество записей в кэш-памяти с непосредственным отображением должно быть кратным степени числа 2. В MIPS-архитектуре, поскольку слова выровнены таким образом, чтобы быть кратными четырем байтам, самые младшие два разряда

каждого адреса определяют байт внутри слова. Следовательно, самые младшие два разряда при выборе слова в блоке игнорируются.

Общее количество разрядов, необходимых для кэш-памяти, определяется функцией от размера кэш-памяти и размера адресного пространства, поскольку кэш-память включает в себя как место для хранения данных, так и теги. Размером рассмотренного ранее блока было одно слово, но обычно блок состоит из нескольких слов. Для следующей ситуации:

- 32-разрядный байтовый адрес;
- кэш-память с непосредственным отображением;
- размер кэш-памяти в 2^n блоков, следовательно, для индекса используются n бит;
- размер блока 2^m слов (2^{m+2} байт), следовательно, для слова внутри блока используются m разрядов, и два разряда используются для байтовой части адреса

размер поля тега составляет

$$32 - (n + m + 2).$$

Общее количество разрядов в кэш-памяти с непосредственным отображением составляет

$$2^n \times (\text{размер блока} + \text{размер тега} + \text{размер поля достоверности}).$$

Поскольку размер блока равен 2^m слов (2^{m+5} бит) и нам нужен один бит для поля достоверности (V), количество бит в такой кэш-памяти будет равно

$$2^n \times (2^m \times 32 + (32 - n - m - 2) + 1) = 2^n \times (2^m \times 32 + 31 - n - m).$$

Хотя здесь представлен реальный размер в битах, соглашение о названиях требует исключить размер тега и поля достоверности и учитывать только размер данных. Поэтому кэш-память на рис. 5.7 называется кэш-памятью размером 4 Кбайт.

Упражнение

Количество бит в кэш-памяти

Сколько всего бит требуется для кэш-памяти с непосредственным отображением, содержащей 16 Кбайт данных и состоящей из блоков в четыре слова при условии, что используется 32-разрядная адресация?

Ответ

Мы знаем, что 16 Кбайт — это 4 К (2^{12}) слов. При размере блока, равном 4 слова (2^2), в кэш-памяти содержится 1024 (2^{10}) блока. У каждого блока имеется 4 × 32, или 128, битов данных плюс тег, в котором 32 – 10 – 2 – 2 бита, плюс бит достоверности. Таким образом, общий размер кэш-памяти равен

$$2^{10} \times (4 \times 32 + (32 - 10 - 2 - 2) + 1) = 2^{10} \times 147 = 147 \text{ Кбит},$$

или 18,4 Кбайт для 16 Кбайт кэш-памяти. Для этой кэш-памяти общее количество битов почти в 1,15 раз больше, чем необходимо для хранения данных.

Индекс	V	Тег	Данные
000	Нет		
001	Нет		
010	Нет		
011	Нет		
100	Нет		
101	Нет		
110	Нет		
111	Нет		

а) начальное состояние кэш-памяти после подачи питания

Индекс	V	Тег	Данные
000	Нет		
001	Нет		
010	Нет		
011	Нет		
100	Нет		
101	Нет		
110	Да	10_2	Память (10110_2)
111	Нет		

б) после обработки неудачи по адресу (10110_2)

Индекс	V	Тег	Данные
000	Нет		
001	Нет		
010	Да	11_2	Память (11010_2)
011	Нет		
100	Нет		
101	Нет		
110	Да	10_2	Память (10110_2)
111	Нет		

в) после обработки неудачи по адресу (11010_2)

Рис. 5.6. Содержимое кэш-памяти показано после каждого неудачного обращения, поля индекса и тега показаны в двоичном формате для последовательности адресов, показанной в предыдущей таблице. Изначально кэш-память не содержит никаких данных, и все разряды достоверности (V-записи в кэш-памяти) сброшены (имеют значение Нет). Процессор запрашивает данные по следующим адресам: 10110_2 (промах), 11010_2 (промах), 10110_2 (попадание), 11010_2 (попадание), 10000_2 (промах), 00011_2 (промах), 10000_2 (попадание), 10010_2 (промах) и 10000_2 (попадание). На рисунках показано содержимое кэш-памяти после обработки

Индекс	V	Тег	Данные
000	Нет	10_2	Память (10000_2)
001	Нет		
010	Да	11_2	Память (11010_2)
011	Нет		
100	Нет		
101	Нет		
110	Да	10_2	Память (10110_2)
111	Нет		

г) после обработки неудачи по адресу (10000_2)

Индекс	V	Тег	Данные
000	Нет	10_2	Память (10000_2)
001	Нет		
010	Да	11_2	Память (11010_2)
011	Нет	00_2	Память (00011_2)
100	Нет		
101	Нет		
110	Да	10_2	Память (10110_2)
111	Нет		

д) после обработки неудачи по адресу (00011_2)

Индекс	V	Тег	Данные
000	Нет	10_2	Память (10000_2)
001	Нет		
010	Да	10_2	Память (10010_2)
011	Нет	00_2	Память (00011_2)
100	Нет		
101	Нет		
110	Да	10_2	Память (10110_2)
111	Нет		

е) после обработки неудачи по адресу (10010_2)

каждого неудачного обращения в этой последовательности. Когда происходит обращение по адресу 10010_2 (18), запись для адреса 11010_2 (26) должна быть заменена, а обращение по адресу 11010_2 должно стать причиной следующей неудачи. Поле тега будет содержать только старшую часть адреса. Полный адрес слова, содержащегося в блоке кэш-памяти i с полем тега j для данной кэш-памяти будет равен $j \times 8 + i$, или эквивалентен конкатенации поля тега j и индекса i. Например, в показанном выше состоянии кэш-памяти *е* индекс 010_2 имеет тег 10_2 и соответствует адресу 10010_2

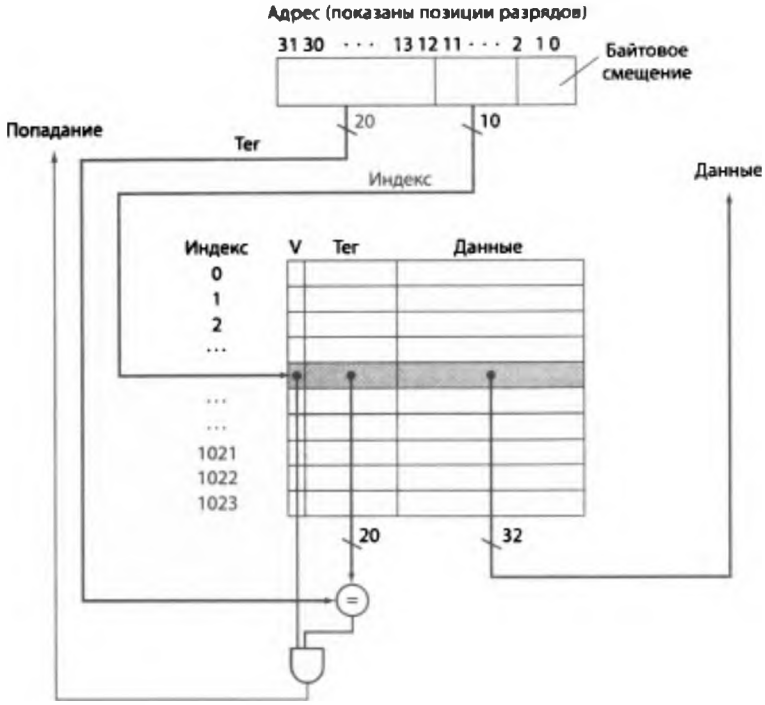


Рис. 5.7. Для данной кэш-памяти младшая часть адреса используется для выбора записи кэш-памяти, состоящей из слова данных и тега. Эта кэш-память сохраняет 1024 слов или 4 Кбайт. В данной главе мы предполагаем использование 32-разрядных адресов. Тег, имеющийся в кэш-памяти, сравнивается со старшей частью адреса для определения, соответствует ли запись в кэш-памяти запрошенному адресу. Поскольку кэш-память содержит 2^{10} (или 1024) слов и размер блока составляет одно слово, для индекса кэш-памяти используются 10 разрядов, оставляя 32 – 10 – 2 = 20 разрядов для сравнения с тегом. Если тег и старшие 20 разрядов адреса равны друг другу и бит достоверности установлен, значит, обращение к кэш-памяти было успешным, и слово предоставляется процессору. В противном случае обращение заканчивается неудачей

Упражнение

Отображение адреса на блок кэш-памяти, состоящий из нескольких слов

Рассмотрим кэш-память, имеющую 64 блока с размером блока, равным 16 байт. На какой номер блока проецируется байтовый адрес 1200?

Ответ

В начале раздела была дана формула для поиска блока:

(Адрес блока) modulo (Количество блоков в кэш-памяти)

где адрес блока это

$$\frac{\text{Адрес байта}}{\text{Байтов в блоке}}.$$

Следует учесть, что этот адрес блока относится к блоку, содержащему все адреса между

$$\left\lfloor \frac{\text{Адрес байта}}{\text{Байтов в блоке}} \right\rfloor \times \text{Байтов в блоке}$$

и

$$\left\lfloor \frac{\text{Адрес байта}}{\text{Байтов в блоке}} \right\rfloor \times \text{Байтов в блоке} + (\text{Байтов в блоке} - 1).$$

Таким образом, при 16 байтах на блок байтовый адрес 1200 является адресом блока

$$\left\lfloor \frac{1200}{16} \right\rfloor = 75,$$

который проецируется на блок кэш-памяти (75 по модулю 64) = 11. На самом деле этот блок проецируется на все адреса между 1200 и 1215.

Для уменьшения количества неудачных обращений более крупные блоки используют локальность, связанную с пространством. Как показано на рис. 5.8, увеличение размера блока обычно снижает коэффициент промахов. Со временем этот коэффициент может повышаться, когда размер блока станет составлять существенную долю размера кэш-памяти, потому что количество блоков, которое может содержаться в кэш-памяти, станет небольшим, и вокруг этих блоков возникнет множество соревновательных ситуаций. В результате содержимое блока будет удалено из кэш-памяти еще до того, как будет осуществлено обращение ко многим его словам. Иначе говоря, при использовании блоков очень большого размера пространственная локальность среди слов блока уменьшается; следовательно, уменьшаются и преимущества, связанные с уменьшением коэффициента промахов.

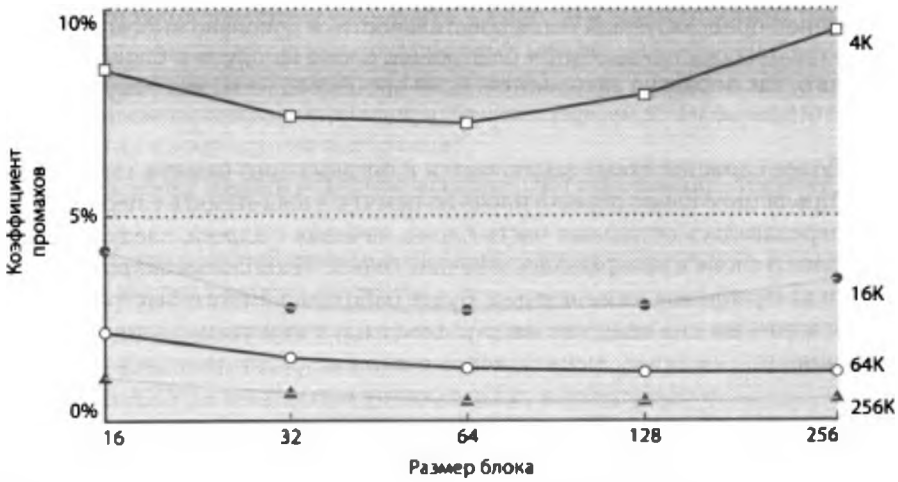


Рис. 5.8. Зависимость коэффициента промахов от размера блока. Заметьте, что коэффициент промахов повышается, если размер блока становится слишком большим относительно размера кэш-памяти. Каждая линия представляет собой кэш-память разного размера. (Этот рисунок не зависит от ассоциативности, которая вскоре будет рассмотрена.) К сожалению, прохождение тестов SPEC2000 заняло бы много времени, если бы был включен размер блока, поэтому эти данные основаны на тестах SPEC92

Более серьезной проблемой, связанной исключительно с увеличением размера блока, является увеличение издержек, связанных с промахами. Они определяются временем, которое требуется для извлечения блока из следующего нижнего уровня иерархии и загрузки его в кэш. Время извлечения блока складывается из двух частей: латентности обращения к первому слову и времени переноса всего остального блока. Понятно, что пока мы не изменим систему памяти, время переноса, а следовательно и издержки, будут, скорее всего, увеличиваться по мере увеличения размера блока. Более того, улучшение показателей, связанных с коэффициентом промахов, с ростом размера блоков начнет снижаться. Результат выразится в том, что увеличение издержек сведет на нет все преимущества от уменьшения коэффициента промахов для слишком больших блоков, и производительность кэш-памяти таким образом снизится. Конечно, если мы разработаем память для более эффективного переноса больших блоков, мы сможем увеличить размер блока и получить дальнейшее улучшение производительности кэш-памяти. Эта тема будет рассмотрена в следующем разделе.

Уточнение. Хотя сделать что-либо с таким компонентом издержек, как продолжительная латентность, для больших блоков довольно трудно, можно убрать часть времени переноса данных, существенно уменьшив таким образом издержки. Простейший метод для этого называется *ранним возобновлением* и заключается в простом возобновлении выполнения, как только будет возвращено запрошенное слово блока, не дожидаясь получения всего блока. На многих процессорах эта технология используется для доступа к инструкциям, где она работает лучше всего. Выборка инструкций чаще всего ведется последовательно, поэтому если система памяти может поставлять слово с каждым тактовым циклом, процессор может быть способен к возобновлению операции, как только будет возвращено запрошенное слово, а система памяти будет своевременно поставлять новые слова инструкций. Обычно такая технология менее эффективна для кэш-памяти данных, потому что чаще всего слова запрашиваются из блока в менее предсказуемой последовательности, и довольно высока вероятность того, что процессору понадобится следующее слово из другого блока кэш-памяти еще до того, как передача завершится. Если процессор не может получить доступ к кэш-памяти данных из-за незавершенной передачи, то должна произойти задержка.

Еще более сложная схема заключается в организации памяти таким образом, чтобы запрошенное слово переносилось из памяти в кэш-память в первую очередь, а затем передавалась остальная часть блока, начиная с адреса, следующего после запрошенного слова и возвращаясь к началу блока. Технология *запрошенное слово первым* или *требуемое слово первым*, будет работать немного быстрее, чем ранее возобновление, но она имеет те же ограничения, что и технология раннего возобновления.

Обработка промахов при обращении к кэш-памяти

Перед тем как приступить к рассмотрению кэш-памяти реальной системы, давайте посмотрим, как блок управления справляется с **промахами при обращениях к кэш-памяти**. (Подробное описание контроллера кэш-памяти будет дано в разделе 5.7).

управления должен обнаружить промах и обработать его, извлекая запрошенные данные из памяти (или, как мы увидим, из кэш-памяти более низкого уровня). Когда кэш-память оповещает о попадании, компьютер продолжает использование данных, как будто ничего не произошло.

Изменение системы управления процессором для обработки попаданий является достаточно простой задачей, а вот обработка промахов требует дополнительных усилий. Обработка промахов при обращениях к кэш-памяти проводится во взаимодействии с блоком управления процессора и с помощью отдельного контроллера, инициирующего обращение к памяти и пополнение кэш-памяти. Обработка промаха создает задержку конвейера (глава 4) в отличие от прерывания, для которого требуется сохранение состояния всех регистров. В случае промаха при обращении к кэш-памяти мы можем остановить весь процессор, по существу заморозив содержимое временных и видимых программисту регистров на время ожидания памяти. Более сложные процессоры с произвольным выполнением инструкций могут допустить выполнение инструкций на время ожидания обработки промаха, но в данном разделе мы будем рассматривать работу процессора с последовательным выполнением инструкций, ожидающего обработки промаха.

Давайте более подробно рассмотрим обработку промаха при обращении к кэш-памяти инструкций; аналогичный подход может быть легко распространен и на обработку промаха при обращении к кэш-памяти данных. Если в результате обращения за инструкцией произошел промах, значит, в регистре инструкции неверное содержимое. Для получения в кэш-память нужной инструкции нужно иметь возможность приказать нижнему уровню в иерархии памяти произвести чтение. Поскольку счетчик команд получает приращение при первом тактовом цикле выполнения, адрес инструкции, генерируемый при промахе обращения к кэш-памяти инструкций, равен значению счетчика команд минус 4. Имея адрес, нам нужно приказать основной памяти осуществить чтение. Мы ждем ответа от памяти (поскольку обращение займет несколько тактовых циклов), а затем записываем слово, содержащее нужную инструкцию, в кэш-память.

Теперь можно определить шаги, которые нужно предпринять в связи с промахом при обращении к кэш-памяти инструкций:

1. Отправить блоку памяти исходное значение счетчика команд (текущее значение PC – 4).
2. Приказать оперативной памяти выполнить чтение и подождать, пока память не завершит доступ к своему элементу.
3. Сделать запись в элемент кэш-памяти, поместив данные из памяти в ту часть элемента кэш-памяти, куда помещаются данные, записав старшие разряды адреса (из АЛУ) в поле тега и установив бит достоверности.
4. Перезапустить выполнение инструкции с первого шага, на котором она будет заново извлечена, на этот раз будучи найденной в кэш-памяти.

Управление кэш-памятью данных по сути такое же: при промахе процессор просто останавливается до тех пор, пока память не распорядится данными.

Промах при обращении к кэш-памяти Запрос на получение данных из кэш-памяти, который не может быть удовлетворен из-за отсутствия данных в кэш-памяти.

Обработка записи

Запись осуществляется несколько иначе. Применительно к инструкции сохранения данные записываются только в кэш-память данных (без внесения изменений в оперативную память); затем, после записи в кэш, в памяти будет храниться значение, отличное от хранящегося в кэш-памяти. В таком случае говорится о том, что кэш-память и оперативная память являются *несоответствующими* друг другу. Простейший способ обеспечения соответствия оперативной памяти и кэш-памяти заключается в неизменной записи данных и в оперативную память, и в кэш-память. Эта схема называется **сквозной записью**.

Другим ключевым аспектом записи является то, что происходит при промахе записи. Сначала из памяти извлекаются слова блока. После того как блок извлечен и помещен в кэш, мы можем переписать слово, вызвавшее промах, в блок кэш-памяти. Также это слово с использованием его полного адреса записывается в оперативную память.

Хотя эта конструкция довольно просто справляется с записью, она не даст высокой производительности. При использовании сквозной записи каждая запись становится причиной записи данных в оперативную память. Эта запись займет много времени, возможно, не менее 100 тактовых циклов процессора, и может существенно замедлить его работу. Например, представим, что 10% инструкций являются сохранениями. Если показатель CPI без учета промахов в кэш-памяти был равен 1,0, затрата 100 дополнительных циклов на каждую запись приведет к тому, что показатель CPI будет равен 1,0 + 100 × 10% = 11, снижая производительность более чем в 10 раз.

Одно из решений заключается в использовании **буфера записи**. Буфер записи хранит данные в ожидании их записи в память. После записи данных в кэш-память и в буфер записи процессор может продолжить выполнение программы. Когда запись в оперативную память завершается, место в буфере памяти освобождается. Когда буфер записи полон, а процессору подошло время вести запись, он должен остановить свою работу и дождаться освобождения места в буфере. Разумеется, если скорость, с которой память может завершить запись, меньше скорости, с которой процессор инициирует операции записи, не поможет никакой объем буфера, потому что операции записи будут инициироваться быстрее, чем система памяти сможет их воспринимать.

Скорость, с которой инициируются операции записи, может также быть ниже скорости, с которой память может их воспринимать, но задержки могут все же происходить. Это может происходить при внезапном росте количества инициируемых операций записи. Для сокращения случаев таких задержек процессоры обычно увеличивают глубину буфера записи, размещая в нем более одного элемента.

Сквозная запись
Схема, при которой в процессе записи всегда обновляется и кэш-память, и следующий, расположенный ниже уровень иерархии памяти, обеспечивая согласование между этими двумя уровнями.

Буфер записи
Очередь, содержащая данные, ожидающие записи в память.

Альтернативой схеме сквозной записи служит схема **отложенной записи**, или *отложенного копирования*. В схеме отложенной записи,

инициируется запись, новое значение записывается только в блок кэш-памяти. Измененный блок при своей замене записывается на уровень иерархии, расположенный ниже текущего. Схемы с отложенной записью могут повысить производительность, особенно когда процессоры могут инициировать запись так быстро или быстрее, чем операции записи могут обрабатываться оперативной памятью; но схема с отложенной записью более сложна в реализации, чем схема со сквозной записью.

Далее в главе будут даны описания кэш-памяти, принадлежащей реальным процессорам, и показано, как они справляются с чтением и записью. В разделе 5.5 будет дано более подробное описание обработки записи.

Замечание. Запись вносит в конструкцию кэш-памяти ряд дополнений, которых не было при обеспечении чтения. Здесь рассматриваются два из них: стратегия обработки промахов при записи и эффективная реализация операций записи в кэш-памяти с отложенной записью.

Рассмотрим промахи в кэш-памяти со сквозной записью. Наиболее распространенная стратегия заключается в выделении блока в кэш-памяти, которое называется *выделением для записи*. Блок извлекается из памяти, а затем соответствующая часть блока переписывается. Альтернативой ей служит стратегия обновления части блока в памяти без помещения его в кэш, которая называется *без выделения для записи*. Причиной такого решения служит то, что иногда программы записывают целые блоки данных, например, когда оперативная система обнуляет страницы памяти. В таких случаях извлечение, связанное с изначальным промахом записи, может не иметь смысла. На некоторых компьютерах разрешается изменение стратегии выделения для записи индивидуально для каждой страницы.

Обычно эффективная реализация инструкций сохранения в кэш-памяти, использующем отложенную запись, сложнее, чем в кэш-памяти со сквозной записью. Кэш-память со сквозной записью может записывать данные в кэш-память и читать тег; если тег не совпадает, происходит промах. Поскольку кэш-память осуществляет сквозную запись, перезаписывание блока в кэш-памяти не приводит к катастрофической ситуации, так как в памяти имеется правильное значение. В кэш-памяти с отложенной записью сначала нужно записать блок обратно в память, если данные в кэш-памяти были изменены и произошел промах при обращении к кэш-памяти. Если мы просто перепишем блок по инструкции сохранения еще до того, как узнаем, попало ли обращение этой инструкции в кэш-память или нет (как это можно сделать при кэш-памяти со сквозной записью), мы уничтожим содержимое блока, который не был скопирован на нижний уровень иерархии памяти.

В кэш-памяти с отложенной записью по причине того, что мы не можем переписать блок, операции сохранения либо требуют два цикла (цикл для проверки попадания, за которым следует цикл, действительно выполняющий запись), либо требуется буфер записи, содержащий эти данные, фактически позволяя инструкции сохранения путем ее конвейеризации занимать только один цикл. Когда используется буфер хранения, процессор

Отложенная запись
Схема, способная обрабатывать операции записи путем обновления только блока в кэш-памяти с последующей записью измененного блока на нижний уровень иерархии во время замены этого блока.

осуществляет поиск в кэш-памяти и помещает данные в буфер хранения в течение обычного цикла доступа к кэш-памяти. При условии попадания в кэш-память новые данные записываются из буфера хранения в кэш-память в течение следующего неиспользуемого цикла доступа к кэш-памяти.

Для сравнения: в кэш-памяти со сквозной записью операции записи всегда могут осуществляться за один цикл. Происходит считывание тега и запись порции данных в выбранный блок. Если тег соответствует адресу записываемого блока, процессор может продолжить нормальную работу, поскольку был обновлен правильный блок. Если тег не соответствует, процессор генерирует промах при записи для извлечения остатка блока, соответствующего этому адресу.

Многие устройства кэш-памяти с отложенной записью также включают буферы записи, используемые для сокращения издержек промахов, когда при обработке промаха заменяется измененный блок. В таком случае измененный блок перемещается в буфер отложенной записи, связанный с кэш-памятью, в то время как запрошенный блок считывается из памяти. Чуть позже содержимое буфера отложенной записи записывается в память. Если предположить, что следом не произойдет еще один промах, эта технология вдвое снижает издержки промаха, когда должен быть заменен измененный блок.

Пример кэш-памяти: процессор Intrinsity FastMATH

Intrinsity FastMATH является быстродействующим встроенным микропроцессором, использующим MIPS-архитектуру и простую реализацию кэш-памяти. Ближе к концу главы будет рассмотрена более сложная конструкция кэш-памяти процессора AMD Opteron X4 (Barcelona), но из педагогических соображений начнем мы с простого, но все же настоящего примера. На рис. 5.9 показана организация кэш-памяти данных процессора Intrinsity FastMATH.

Этот процессор имеет 12-стадийный конвейер, похожий на тот, который рассматривался в главе 4. При работе на пиковой скорости процессор может запросить в каждом цикле как слово инструкции, так и слово данных. Для удовлетворения потребностей конвейера без задержек используются отдельные устройства кэш-памяти инструкций и данных. Каждый кэш имеет объем 16 Кбайт, или 4 Кслов, и состоит из блоков размером в 16 слов.

Запросы на чтение кэш-памяти устроены просто. Поскольку используется отдельная кэш-память для данных и инструкций, нам нужны отдельные управляющие сигналы чтения и записи для каждой кэш-памяти. (Вспомним, что при промахе нам нужно обновить кэш-память инструкций.) Таким образом, для запроса чтения для обоих блоков кэш-памяти требуются следующие шаги:

1. Отправка адреса соответствующей кэш-памяти. Адрес поступает либо из счетчика команд (для инструкции) либо из АЛУ (для данных).
2. Если кэш-память посылает сигнал попадания, запрошенное слово становится доступным. Поскольку в нужном блоке содержится 16 слов, необходимо выбрать из них только одно. Поле индекса блока используется для управления

мультиплексором (показан в нижней части рисунка), который выбирает запрошенное слово из 16 слов в проиндексированном блоке.

Если кэш-память посылает сигнал промаха, адрес посылается оперативной памяти. Когда память возвращает данные, они записываются в кэш-память и затем считываются для удовлетворения запроса.

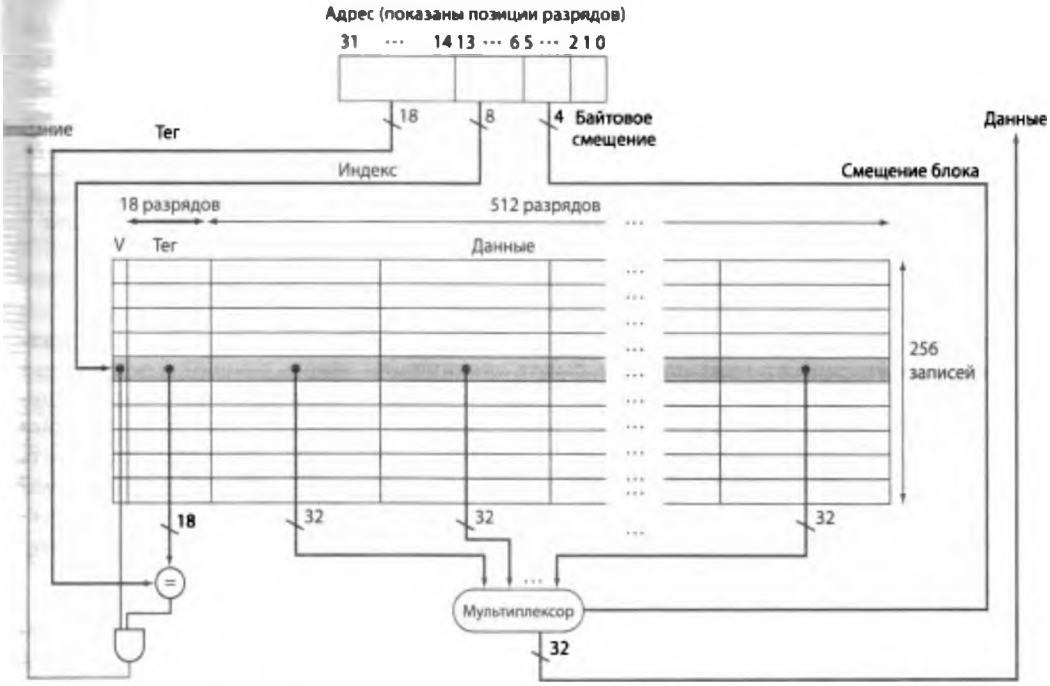


Рис. 5.9. Каждый кэш в Intrinsity FastMATH объемом 16 Кбайт содержит 256 блоков по 16 слов на блок. Поле тега содержит 18 разрядов, а индексное поле содержит 8 разрядов, 4-разрядное поле (разряды 5–2) используется для индексации блока, для выбора слова из блока используется мультиплексор 16-к-1. На практике, чтобы исключить мультиплексор, устройства кэш-памяти используют отдельную большую RAM-память для данных и меньшую по размеру RAM-память для тегов, а смещение блока предоставляет дополнительные разряды адреса для большой RAM-памяти данных. В таком случае большая RAM-память имеет ширину 32 разряда и должна иметь в шестнадцать раз больше слов, чем блоков в кэш-памяти

Для записи Intrinsity FastMATH предлагает как сквозную, так и отложенную стратегию, оставляя право операционной системе решать, какую из них применить для приложения. В этом процессоре имеется буфер на одну запись.

Какой коэффициент промахов достигается с помощью структуры кэша, подобной используемой в Intrinsity FastMATH? В табл. 5.1 показаны коэффициенты промахов для кэш-памяти инструкций и данных. Смешанный коэффициент промахов является эффективным коэффициентом промахов для каждой программы после вычисления различий в частотах обращений к инструкциям и данным.

Хотя коэффициент промахов является важной характеристикой конструкций кэш-памяти, конечным показателем будет влияние системы памяти на время

выполнения программы; скоро мы увидим, какая взаимосвязь имеется у коэффициента промахов и у времени выполнения.

Таблица 5.1. Примерный коэффициент промахов при обращении к кэш-памяти инструкций и кэш-памяти данных для процессора Intrinsity FastMATH при выполнении контрольных задач SPEC2000. Смешанный коэффициент промахов является эффективным коэффициентом, показанным для комбинации 16-Кбайтной кэш-памяти инструкций и 16-Кбайтной кэш-памяти данных. Он получен путем вычисления взвешенной величины индивидуальных коэффициентов промахов инструкций и данных по отношению к частоте обращений к инструкциям и данным

Коэффициент промахов при обращении к инструкциям	Коэффициент промахов при обращении к данным	Эффективный смешанный коэффициент промахов
0,4%	11,4%	3,2%

Уточнение. Объединенная кэш-память с общим размером, равным сумме двух **раздельных устройств кэш-памяти**, будет, как правило, иметь лучший коэффициент попаданий. Этот более высокий коэффициент получается благодаря тому, что объединенная кэш-память не проводит жесткое разделение по количеству записей, которые могут быть использованы инструкциями, и количеству записей, которые могут быть использованы данными. Тем не менее многие процессоры для повышения *пропускной способности* кэш-памяти используют раздельные устройства кэш-памяти инструкций и данных. (У них также может быть меньше промахов, связанных с конфликтами; см. раздел 5.5.)

Вот как выглядят коэффициенты промахов для кэш-памяти с размером таким же, как у процессора Intrinsity FastMATH, и для объединенной кэш-памяти, чей размер равен сумме размеров двух устройств кэш-памяти:

- общий размер кэш-памяти: 32 Кбайт;
- эффективный коэффициент промахов раздельных устройств кэш-памяти: 3,24%;
- коэффициент промахов объединенной кэш-памяти: 3,18%.

Коэффициент раздельной кэш-памяти лишь немногим хуже.

Преимущество удвоенной пропускной способности кэш-памяти, которое заключается в одновременной поддержке доступа к инструкциям и данным, вполне компенсирует недостаток небольшого увеличения коэффициента промахов. Раздел 5.3 показывает, что данное наблюдение предостерегает от использования коэффициента промахов в качестве единственного показателя производительности кэш-памяти.

Раздельная кэш-память
Схема, в которой уровень иерархии памяти составлен из двух независимых устройств кэш-памяти, работающих в параллельном режиме по отношению друг к другу, причем один из них работает с инструкциями, а другой с данными.

Конструирование системы памяти с поддержкой кэш-памяти

Промахи при обращениях к кэш-памяти удовлетворяются за счет обращения к оперативной памяти, которая построена на DRAM-элементах. В разделе 5.1 мы видели, что главными характеристиками DRAM-элементов являются стоимость и плотность. Хотя сократить латентность извлечения первого слова из памяти довольно трудно, издержки промахов можно сократить за счет повышения пропускной способности от оперативной памяти к кэш-памяти. Такое сокращение даст возможность использовать более крупные блоки при сохранении низких издержек на промахи, характерных для блоков меньшего размера.

Процессор традиционно соединяется с памятью по шине. (Как будет показано в главе 6, эта традиция изменяется, но более современная технология соединений в данной главе никакой роли не играет, поэтому мы будет пользоваться термином «шина».) Тактовая частота у шины обычно намного ниже, чем у процессора. Скорость работы этой шины влияет на издержки промахов.

Для оценки влияния различных способов организации памяти давайте определим набор гипотетических составляющих времени обращения к памяти. Предположим, что он состоит из:

- 1 тактового цикла шины, необходимого для отправки адреса.
- 15 тактовых циклов шины для инициализации обращения к каждому DRAM-элементу.
- 1 тактового цикла шины для отправки слова данных.

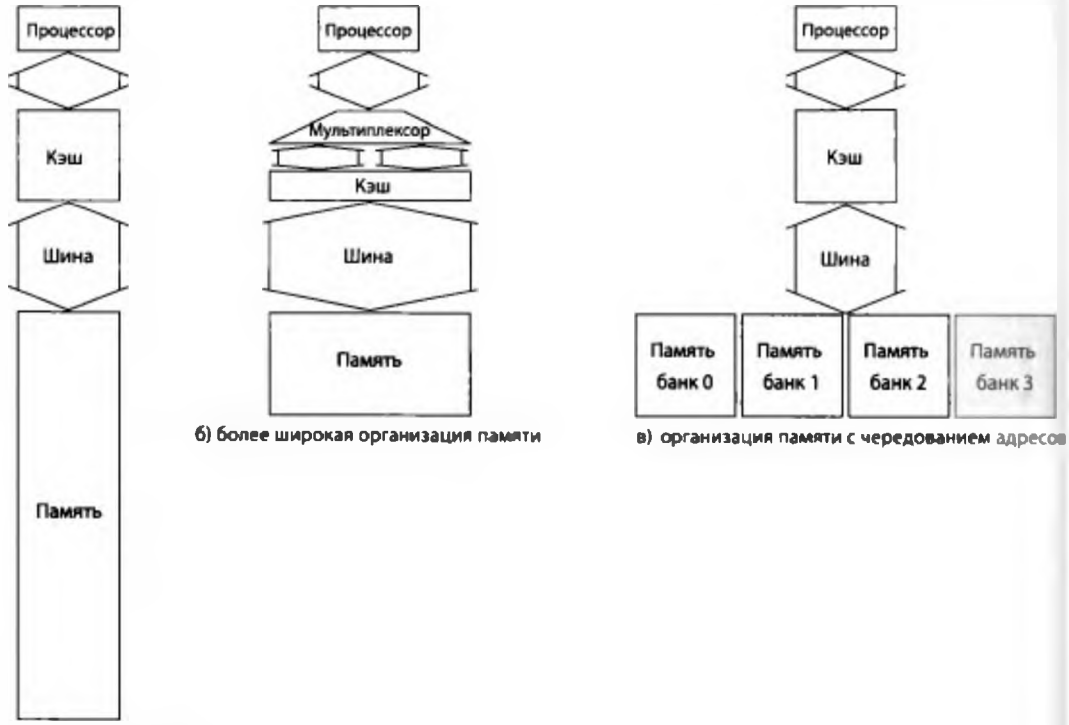
Если блок нашей кэш-памяти состоит из четырех слов и мы располагаем банком DRAM-элементов шириной в одно слово, издержки промаха составят $1 + 4 \times 15 + 4 \times 1 = 65$ тактовых циклов шины памяти. Таким образом, количество байтов, переносимых за один тактовый цикл шины при одном промахе, будет равно

$$\frac{4 \times 4}{65} = 0{,}25.$$

На рис. 5.10 показаны три варианта конструкции системы памяти. Первый вариант соответствует нашим предположениям: память имеет ширину в одно слово, и все обращения осуществляются последовательно. Во втором варианте полоса пропускания к памяти увеличена за счет расширения памяти и шины между памятью и процессором, что допускает параллельное обращение сразу к нескольким словам блока. В третьем варианте полоса пропускания увеличена за счет расширения памяти, но не соединительной шины. Таким образом, мы по-прежнему платим ту же цену за передачу каждого слова, но можем избежать более одной оплаты латентности доступа. Давайте посмотрим, насколько другие два варианта улучшают показатель 65-цикловой издержки промаха, характерный для первого варианта, показанного на рис. 5.10, *а*.

Расширение памяти и шины приводит к пропорциональному увеличению пропускной способности памяти, уменьшая при этом такие составляющие издержки

промаха, как время доступа и время переноса данных. При оперативной памяти шириной в два слова издержки промаха падают с 65 тактовых циклов шины памяти до 1 + (2 × 15) + 2 × 1 = 33 тактовых циклов этой шины. Пропускная способность для одного промаха составляет в таком случае 0,48 (почти вдвое выше) байт на тактовый цикл шины для памяти шириной в два слова. Основной ценой такого улучшения становится более широкая шина и потенциальное увеличение времени доступа к кэш-памяти из-за мультиплексора и логики управления между процессором и кэш-памятью.



а) организация памяти шириной в одно слово

б) более широкая организация памяти

в) организация памяти с чередованием адресов

Рис. 5.10. Главным методом достижения более высокой пропускной способности памяти является увеличение физической или логической ширины системы памяти. На данном рисунке пропускная способность памяти увеличивается двумя способами. В самой простой конструкции, *а*, используется память, в которой все компоненты имеют ширину в одно слово; в конструкции *б* показаны расширенные память, шина и кэш; а в конструкции *в* показана узкая шина и кэш при памяти с чередованием адресов. В конструкции *б* логика между кэш-памятью и процессором состоит из мультиплексора, используемого при чтении, и логики управления для обновления соответствующих слов кэш-памяти при записи

Вместо расширения всего пути между оперативной и кэш-памятью микросхемы памяти могут быть организованы в банки для чтения и записи сразу нескольких слов за одно время доступа, вместо того чтобы читать или записывать за это время всего одно слово. Каждый банк может быть шириной в одно слово, чтобы не

лось изменять ширину шины и кэш-памяти, но отправка адреса нескольким ам разрешает всем им проводить чтение одновременно. Эта схема, которая ывается *чередованием*, сохраняет преимущество однократных затрат на полную ентность памяти. Например, при наличии четырех банков время получения ока из четырех слов состоит из одного цикла для передачи адреса и чтения запро- в банкам, 15 циклов для всех четырех банков для доступа к памяти, и 4 циклов для отправки четырех слов обратно кэш-памяти. В результате издержки промаха ставляют 1 + (1 × 15) + 4 × 1 = 20 тактовых циклов шины памяти. Эффективная пускная способность для одного промаха при этом становится равной 0,80 бай- на тактовый цикл, или почти в три раза выше пропускной способности для мяти и шины шириной в одно слово.

Банки также выгодны при записи. Каждый банк может записываться неза- симо от других, увеличивая пропускную способность записи в четыре раза и снижая уровень задержек в кэш-памяти со сквозной записью. Мы еще увидим, что альтернативная стратегия записи еще больше повышает привлекательность чередования. Из-за широкого распространения кэш-памяти и желания иметь более крупные размеры блоков, производители DRAM-памяти предусмотрели пакетный доступ к данным из серии последовательных мест в DRAM. Самая по- следняя разработка называется *Double Data Rate* (DDR) — DRAM-память, память с удвоенной скоростью передачи данных. Ее название означает перенос данных как на возрастающем, так и на падающем фронте тактового импульса, благодаря чему достигается удвоение пропускной способности против ожидаемой при имеющихся тактовой частоте и ширине данных. Для достижения такой высокой пропускной способности внутренние DRAM-элементы организованы в банки памяти с чере- дованием адресов.

Преимущество такой оптимизации состоит в том, что здесь используются элек- трические схемы, расположенные в основном на DRAM-элементах, что незначи- тельно повышает стоимость системы при достижении существенного увеличения пропускной способности.

Уточнение. Микросхемы памяти выдают сразу несколько битов, обычно от 4 до 32, в 2008 году наиболее популярным количеством было число 16. Мы даем описание ор- ганизации RAM в виде $d \times w$, где d — это количество адресуемых мест (глубина), а w — это выход (или ширина каждого места). DRAM-элементы логически организованы в виде прямоугольных таблиц, и время доступа делится на доступ к строкам и доступ к столбцам. В DRAM-элементах буферизации подвергается строка. Пакетная передача данных позволяет осуществлять повторные обращения к буферу без затрат времени на доступ к строке. Буферы работают как SRAM; путем изменения адреса столбца до тех пор, пока не будет обращения к следующей строке, в буфере может быть получен про- извольный доступ к отдельным битам. Эта возможность существенно изменяет время доступа, поскольку время доступа к битам в строке намного меньше. На рис. 5.12 по- казано, как с годами изменялись плотность DRAM-элементов, их стоимость и время доступа. Для улучшения интерфейса с процессорами в DRAM-элементы был добавлен таймер, и они стали называться синхронными DRAM-элементами, или SDRAM.

Преимущество SDRAM-памяти состоит в том, что использование таймера устраняет необходимость траты времени на синхронизацию памяти и процессора.

Уточнение. Один из способов оценки производительности систем памяти состоит в использовании контрольных задач Stream benchmark (McCalpin, 1995). С их помощью измеряется производительность длинных векторных операций. У них нет локализации, связанной со временем, и они обращаются к таблицам, которые больше, чем кэш-память тестируемого компьютера.

Уточнение. Пакетный режим для DDR-памяти также применяется на шинах памяти, например на такой, как Intel Duo Core Front Side Bus.

Core Duo.

Таблица 5.2. Размер DRAM-элементов до 1996 года каждые три года увеличивался в четыре раза, а затем рост существенно замедлился. Сокращение времени доступа происходило медленнее, а стоимость падала практически со скоростью роста плотности, хотя стоимость часто зависела и от других факторов, например от баланса спроса и предложения. Инфляция не повлияла на стоимость хранения гигабайта данных

Год выпуска	Размер микросхемы	Долл. за Гбайт	Общее время доступа к новой строке/столбцу	Время доступа к столбцу имеющейся строки
1980	64 Кбит	1 500 000	250 нс	150 нс
1983	256 Кбит	500 000	185 нс	100 нс
1985	1 Мбит	200 000	135 нс	40 нс
1989	4 Мбит	50 000	110 нс	40 нс
1992	16 Мбит	15 000	90 нс	30 нс
1996	64 Мбит	10 000	60 нс	12 нс
1998	128 Мбит	4 000	60 нс	10 нс
2000	256 Мбит	1 000	55 нс	7 нс
2004	512 Мбит	250	50 нс	5 нс
2007	1 Гбит	50	40 нс	1,25 нс

Краткие выводы

Данный раздел начинался с изучения простейшей кэш-памяти: с непосредственным отображением и блоком, состоящим из одного слова. В кэш-памяти такой конструкции контроль промахов довольно прост, поскольку слово может помещаться только в конкретное место и для каждого слова есть отдельный тег. Для поддержания соответствия содержимого оперативной и кэш-памяти может быть использована схема сквозной записи, при которой каждая запись в кэш-память вызывает также обновление оперативной памяти. Альтернативой сквозной записи служит схема отложенной записи, при которой блок копируется обратно в оперативную память при его замене. Эта схема еще будет рассматриваться в следующих разделах.

Для того чтобы воспользоваться преимуществами локальности в пространстве, у кэш-памяти должен быть размер блока больше одного слова. Использование более крупных блоков уменьшает коэффициент промахов и повышает эффектив-

кость кэш-памяти за счет сокращения в кэш-памяти количества элементов памяти, занятых тегами, относительно количества элементов памяти, занятых данными. Несмотря на то что более крупные блоки уменьшают коэффициент промахов, они также могут увеличить издержки промахов. Если издержки промахов возрастают линейно по отношению к размеру блока, более крупные блоки могут вызвать снижение производительности.

Чтобы избежать потерь производительности, для более эффективной передачи блоков повышается пропускная способность оперативной памяти. Общепринятые методы повышения внешней пропускной способности DRAM-элементов заключаются в расширении разрядности памяти и использовании в ней чередования адресов. Разработчики DRAM-памяти стремятся улучшить интерфейс между процессором и памятью для повышения пропускной способности пакетного режима передачи с целью снижения негативных последствий от увеличения размеров блока кэш-памяти.

Самопроверка

Скорость работы системы памяти влияет на решения конструкторов о размере блока кэш-памяти. Какие из следующих конструкторских установок в целом верны?

1. Чем меньше латентность памяти, тем меньше размер блока кэш-памяти.
2. Чем меньше латентность памяти, тем больше размер блока кэш-памяти.
3. Чем выше пропускная способность памяти, тем меньше размер блока кэш-памяти.
4. Чем выше пропускная способность памяти, тем больше размер блока кэш-памяти.

5.3. Измерение и повышение производительности кэш-памяти

В данном разделе мы начнем с изучения способов измерения и анализа производительности кэш-памяти. Затем мы исследуем две разные технологии повышения производительности кэш-памяти. В одной из них основное внимание уделяется уменьшению коэффициента промахов путем уменьшения вероятности того, что два разных блока памяти будут соперничать за одно и то же место в кэш-памяти. Во второй технологии сокращаются издержки промахов путем добавления к иерархии еще одного уровня. *Многоуровневое кэширование* впервые появилось в дорогостоящих компьютерах, продававшихся в 1990 году более чем за 100 000 долларов, со временем оно стало вполне обычным для настольных компьютеров, которые стоят менее 500 долларов!

Время центрального процессора (ЦПУ) может быть поделено на тактовые циклы, которые он тратит на выполнение программы, и на тактовые циклы, в течение которых он ожидает завершения работы системы памяти. Обычно предполагается,

что стоимость успешных обращений к кэш-памяти является частью обычных циклов центрального процессора, затрачиваемых на выполнение программы. Таким образом,

$$\text{Время ЦПУ} = (\text{Тактовые циклы ЦПУ, потраченные на выполнение} + \text{Тактовые циклы, потраченные на ожидание памяти}) \times \text{Время тактового цикла}$$

Здесь предполагается, что тактовые циклы, затрачиваемые на ожидание памяти, берутся непосредственно из промахов при обращении к кэш-памяти. Мы также ограничиваем рассмотрение путем использования упрощенной модели системы памяти. В настоящих процессорах учесть задержки, генерируемые при чтении и записи, может быть очень сложно, и точное предсказание производительности обычно требует слишком подробного моделирования процессора и системы памяти.

Тактовые циклы, потраченные на ожидание памяти, могут быть определены в виде суммы циклов задержки, вызванных чтением, плюс сумма циклов задержки, вызванных записью:

$$\text{Тактовые циклы, потраченные на ожидание памяти} = \text{Циклы ожидания чтения} + \text{Циклы ожидания записи}$$

Количество циклов ожидания при чтении может быть определено в терминах количества обращений по чтению за программу, издержек промахов при чтении в тактовых циклах и коэффициента промахов при чтении:

$$\text{Количество циклов ожидания при чтении} = \frac{\text{Количество чтений}}{\text{На программу}} \times \text{Количество промахов при чтении} \times \text{Издержки промахов при чтении}$$

С записью дело обстоит сложнее. Для схемы сквозной записи у нас есть два источника задержек: промахи при записи, которые обычно требуют извлечения блока перед тем, как продолжить запись (подробности работы с записями даны в подразделе «Обработка записи»), и задержки буфера записи, которые происходят, когда буфер переполняется. Таким образом, количество циклов ожидания при записи равно сумме следующих двух составляющих:

$$\text{Количество циклов ожидания при чтении} = \left(\frac{\text{Количество чтений}}{\text{На программу}} \times \text{Количество промахов при чтении} \times \text{Издержки промахов при чтении}\right) + \text{Ожидания буфера записи}$$

Поскольку задержки буфера записи зависят от соседства записей, а не только от их частоты, дать простое уравнение для вычисления подобных задержек невозможно. К счастью, в системах с рациональной глубиной буфера (например, четыре слова и более) и памятью, способной воспринимать операции записи на скорости, которая значительно превышает среднюю частоту операций записи в программах (например, в два раза), задержки буфера записи будут незначительными, и мы их

можем спокойно проигнорировать. Если система не отвечает этим критериям, значит, она не была хорошо спроектирована; разработчик должен был воспользоваться либо более глубоким буфером записи, либо структурой с отложенной записью.

Схемы с отложенной записью также имеют потенциальные дополнительные задержки, возникающие из-за необходимости записывать блок кэш-памяти при его замене обратно в память. Более подробно эта схема будет рассмотрена в разделе 5.5.

В большинстве способов организации кэш-памяти со сквозной записью издержки промахов при чтении и записи имеют одинаковые значения (это время на извлечение блока из памяти). Если предположить, что задержки буфера записи пренебрежительно малы, мы можем объединить чтение и запись, используя один коэффициент промахов и одно значение издержек промахов:

$$\text{Количество циклов ожидания при чтении} = \frac{\text{Количество обращений к памяти}}{\text{На программу}} \times$$

$$\times \text{ Коэффициент промахов} \times \text{Издержки промахов}$$

Мы можем также разложить это выражение на множители:

$$\text{Количество циклов ожидания памяти} = \frac{\text{Количество инструкций}}{\text{На программу}} \times$$

$$\times \frac{\text{Коэффициент промахов}}{\text{На инструкцию}} \times \text{Издержки промахов}$$

Давайте рассмотрим простой пример, помогающий понять влияние производительности кэш-памяти на производительность процессора.

Упражнение

Вычисление производительности кэш-памяти

При условии, что коэффициент промахов при обращениях к кэш-памяти инструкций равен 2%, а коэффициент промахов при обращениях к кэш-памяти данных равен 4%. Если процессор без задержек памяти имеет показатель CPI, равный 2, и издержки промахов составляют для всех промахов 100 циклов, нужно определить, насколько быстрее процессор будет работать с совершенной кэш-памятью, при обращении к которой никогда не бывает промахов. Предположим, что частота всех загрузок и сохранений составляет 36%.

Ответ

Количество циклов ожидания памяти при промахах, связанных с обращениями к кэш-памяти инструкций, в расчете на количество инструкций (I) равно

$$\text{Циклов при промахах в получении инструкций} = I \times 2\% \times 100 = 2{,}00 \times I$$

При частоте всех загрузок и сохранений, равной 36%, мы можем определить количество циклов ожидания памяти при промахах, связанных с обращениями к кэш-памяти данных:

$$\text{Циклов при промахах в получении данных} = I \times 36\% \times 4\% \times 100 = 1{,}44 \times I$$

Общее количество циклов ожидания памяти будет равно 2,00 I + 1,44 I = 3,44 I. Это более чем три цикла ожидания памяти на одну инструкцию. Соответственно, общий показатель CPI,

включая циклы ожидания памяти, равен 2 + 3,44 = 5,44. Поскольку количество инструкций или тактовая частота не изменялись, соотношение процессорных времен равно:

$$\frac{\text{Время ЦПУ с задержками}}{\text{Время ЦПУ при идеальной кэш-памяти}} = \frac{I \times CPI_{\text{задержка}} \times \text{Продолжительность цикла}}{I \times CPI_{\text{идеал}} \times \text{Продолжительность цикла}} =$$

$$= \frac{I \times CPI_{\text{задержка}}}{I \times CPI_{\text{идеал}}} = \frac{5,44}{2}$$

Производительность при идеальной кэш-памяти лучше

$$\text{в } \frac{5,44}{2} = 2,72 \text{ раза}$$

Что произойдет, если процессор быстрее системы памяти? Время, затрачиваемое на ожидания памяти, будет занимать все большую долю времени выполнения задачи; закон Амдала, рассмотренный в главе 1, напоминает нам об этом факте. Несколько простых примеров показывают, насколько серьезной может быть эта проблема. Предположим, что мы ускорили работу компьютера в предыдущем упражнении за счет сокращения его CPI с 2 до 1, не меняя при этом тактовой частоты, что может быть осуществлено с помощью усовершенствованного конвейера. Тогда система с промахами обращений к кэш-памяти будет иметь показатель CPI равный 1 + 3,44 = 4,44, а система с идеальной кэш-памятью будет в

$$\frac{4,44}{1} = 4,44 \text{ раза быстрее}$$

Доля времени выполнения задачи, затрачиваемого на ожидания памяти, увеличится с

$$\frac{3,44}{5,44} = 63\%$$

до

$$\frac{3,44}{4,44} = 77\%$$

Аналогично этому, повышение тактовой частоты без внесения изменений в систему памяти также повысит потери производительности, связанные с промахами обращений к кэш-памяти.

Предыдущие примеры и уравнения предполагают, что время попадания не является фактором в определении производительности кэш-памяти. Понятно, что при увеличении времени попадания увеличится и общее время доступа к слову из системы памяти, что может вызвать увеличение продолжительности тактового цикла процессора. Скоро мы рассмотрим дополнительные примеры того, как может увеличиться время попадания, одним из таких примеров является увеличение размера кэш-памяти. Вполне очевидно, что у большей по объему кэш-памяти будет большее время доступа, точно так же, как если бы ваш стол в библиотеке был очень

большим (скажем, три квадратных метра), то нужную книгу на этом столе вы искали бы дольше. Увеличение времени попадания, скорее всего, заставит добавить к конвейеру еще одну стадию, поскольку попадание в кэш-память может занять несколько циклов. Хотя влияние более глубокого конвейера на производительность рассчитать труднее. Рано или поздно увеличение времени попадания для более объемной кэш-памяти может полностью свести на нет эффект от улучшения коэффициента попадания, приводя к снижению производительности процессора.

Чтобы зафиксировать факт влияния на производительность времени доступа к данным, как при попаданиях, так и при промахах, разработчики иногда используют *среднее время доступа к данным* (average memory access time — AMAT) в качестве способа изучения альтернативных конструкций кэш-памяти. Среднее время доступа к данным учитывает как попадания, так и промахи и частоту различных обращений; для него можно составить следующее уравнение:

$$\text{AMAT} = \text{Время попадания} + \text{Коэффициент промахов} \times \text{Издержка промаха}$$

Упражнение

Вычисление среднего времени доступа к памяти

Определите AMAT для процессора с продолжительностью тактового цикла в 1 нс, издержкой промаха в 20 тактовых циклов, коэффициентом промахов, равным 0,05 промахов на инструкцию, и временем доступа к кэш-памяти (включая определение факта попадания), равном одному тактовому циклу. Предположим, что издержки промахов при чтении и при записи имеют одинаковые значения, а все остальные паузы при записи игнорируются.

Ответ

Среднее время доступа к памяти на одну инструкцию равно

```
AMAT = Время попадания + Коэффициент промахов × Издержка промаха
     = 1 + 0.05 × 20
     = 2 тактовых цикла
```

или 2 нс.

В следующем подразделе будет рассмотрена альтернативная организация кэш-памяти, которая уменьшает коэффициент промахов, но может временами увеличивать время попадания. Дополнительные упражнения будут даны в разделе 5.10 «Заблуждения и недоразумения».

Уменьшение коэффициента промахов в кэш-памяти за счет более гибкого размещения блоков

До сих пор, помещая блок в кэш-память, мы использовали простую схему размещения: блок может попадать исключительно в одно и то же место в кэш-памяти. Как ранее упоминалось, это называется непосредственным отображением любого адреса блока в памяти на вполне определенное место на верхнем уровне иерархии. Но на самом деле существует целый ряд схем для размещения блоков. Непосредственное отображение, при котором блок может быть помещен только во вполне определенное место, является одной из граней всего спектра возможностей.

Другая схема предусматривает помещение блока в любое место кэш-памяти. Такая схема называется **полностью ассоциативной кэш-памятью**, потому что блок в памяти может быть связан с любой записью в кэш-памяти. Для поиска нужного блока в полностью ассоциативной кэш-памяти должны быть просмотрены все имеющиеся в ней записи, потому что блок может быть помещен в любую из них. Для удобства поиск ведется параллельно с использованием компаратора, связанного с каждой записью кэш-памяти. Компараторы существенно повышают стоимость компьютерного оборудования, из-за чего полностью ассоциативное размещение остается практичным только для кэш-памяти с небольшим количеством блоков.

Между конструкциями с непосредственным отображением и полной ассоциативностью существует промежуточный вариант, называемый **множественно-ассоциативным**. В кэш-памяти с множественной ассоциативностью существует вполне определенное число мест, в которые может быть помещен каждый блок. Кэш-память с n местами для каждого блока называется n-канальной ассоциативной кэш-памятью. Эта кэш-память состоит из группы наборов, каждый из которых состоит из n блоков. Каждый блок в памяти отображается на уникальный набор в кэш-памяти, задаваемый полем индекса, и блок может быть помещен в любой элемент этого набора. Таким образом, в множественно-ассоциативном размещении комбинируется размещение с непосредственным отображением и полностью ассоциативное размещение: блок непосредственно отображается на конкретный набор, а затем все блоки в наборе подвергаются поиску на соответствие. Например, на рис. 5.11 показано, куда может быть помещен блок 12 в кэш-памяти, состоящей всего из восьми блоков, согласно трем правилам размещения блоков.

Следует напомнить, что в кэш-памяти с непосредственным отображением позиция блока памяти задается следующим образом:

(Номер блока) mod (Количество блоков в кэш-памяти)

В кэш-памяти с множественной ассоциативностью набор, содержащий блок памяти, задается следующим образом:

(Номер блока) mod (Количество наборов в кэш-памяти)

Поскольку блок может быть помещен в любой элемент набора, должны быть просмотрены *все теги всех элементов набора*. В полностью ассоциативной кэш-памяти блок может быть помещен в любое место, и должны быть просмотрены *все теги всех блоков кэш-памяти*.

Все стратегии размещения всех блоков могут быть также представлены как вариации множественной ассоциативности. На рис. 5.12 показаны возможные ассоциативные структуры для кэш-памяти, состоящей из восьми блоков. Кэш-память с непосредственным отображением является просто одноканальной множественно-ассоциативной кэш-памятью: каждый ее элемент содержит один блок, а каждый набор имеет один элемент. Полностью ассоциативная кэш-память с m элементами — это просто m-канальная ассоциативная память; в ней имеется один набор из m блоков, и элемент может быть размещен в любом блоке в пределах набора.

Полностью ассоциативная кэш-память
Структура кэша, в которой блок может быть помещен в любое место кэш-памяти.

Множественно-ассоциативная кэш-память
Кэш-память, у которой имеется фиксированное количество мест (как минимум два), куда может быть помещен каждый блок.

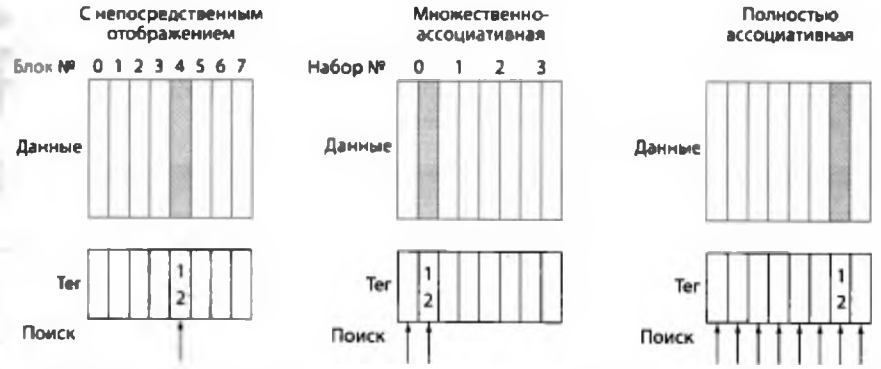


Рис. 5.11. Размещение блока памяти с адресом 12 в кэш-памяти из восьми блоков различается для кэш-памяти с непосредственным отображением, множественной ассоциативностью и полной ассоциативностью. В размещении с непосредственным отображением, существует только один блок кэш-памяти, где может быть найден блок памяти 12, и этот блок определяется как (12 по модулю 8) = 4. В двухканальной множественно-ассоциативной кэш-памяти будет четыре набора, и блок памяти 12 должен быть в наборе (12 по модулю 4) = 0; блок памяти может быть в любом элементе набора. В размещении с полной ассоциативностью блок памяти с адресом 12 может появиться в любом из восьми блоков кэш-памяти

Рис. 5.12. Кэш-память из восьми блоков, сконфигурированная с непосредственным отображением, с двухканальной ассоциативностью, с четырехканальной ассоциативностью и в полностью ассоциативном варианте. Общий размер кэш-памяти в блоках равен количеству наборов, помноженному на количество каналов ассоциативности. Таким образом, для фиксированного размера кэш-памяти увеличение количества каналов ассоциативности уменьшает количество наборов при увеличении количества элементов в наборе. При наличии восьми блоков восьмиканальная ассоциативная кэш-память становится аналогом полностью ассоциативной кэш-памяти

Преимущество от повышения количества каналов ассоциативности состоит в том, что оно, как показано в следующем упражнении, обычно снижает коэффициент промахов. Основной недостаток, который вскоре будет подробно рассмотрен, состоит в потенциальном увеличении времени попадания.

Упражнение

Промахи и количество каналов ассоциативности в кэш-памяти

Предположим, что у нас имеется три небольшие по объему кэш-памяти, каждая из которых состоит из четырех однословных блоков. Одна кэш-память является полностью ассоциативной, вторая имеет двухканальную ассоциативность, а третья является кэш-памятью с непосредственным отображением. Определите количество промахов для каждой организации кэш-памяти для следующей последовательности адресов блоков: 0, 8, 0, 6 и 8.

Ответ

Проще всего выполнить упражнение для кэш-памяти с непосредственным отображением. Сначала давайте определим, на какой блок кэш-памяти проецируется каждый адрес блока:

Адрес блока	Блок кэш-памяти
0	(0 по модулю 4) = 0
6	(6 по модулю 4) = 2
8	(8 по модулю 4) = 0

Теперь можно заполнить кэш-память после каждого обращения, используя пустые записи, чтобы обозначить недостоверность блока, жирный шрифт для того, чтобы показать добавленную в кэш-память новую запись для соответствующего обращения, а обычный шрифт для обозначения старой записи в кэш-памяти:

Адрес блока памяти, к которому произошло обращение	Попадание или промах	Содержимое блоков кэш-памяти после обращения			
		0	1	2	3
0	промах	**Память[0]**			
8	промах	**Память[8]**			
0	промах	**Память[0]**			
6	промах	Память[0]		**Память[6]**	
8	промах	**Память[8]**		Память[6]	

В кэш-памяти с непосредственным отображением произошло пять промахов при пяти обращениях.

У кэш-памяти с множественной ассоциативностью имеется два набора (с индексами 0 и 1) с двумя элементами в каждом наборе. Давайте сначала определим, на какой набор проецируется каждый адрес блока:

Адрес блока	Набор кэш-памяти
0	(0 по модулю 2) = 0
6	(6 по модулю 2) = 0
8	(8 по модулю 2) = 0

Поскольку у нас есть выбор, какую запись в наборе заменить в случае промаха, нам требуется правило замены. В кэш-памяти с множественной ассоциативностью обычно в наборе заменяется блок с наиболее длительным отсутствием обращений; то есть заменяется блок, которым воспользовались в самом отдаленном прошлом. (Другие правила замены будут далее рассмотрены более подробно.)

При использовании данного правила замены кэш-память с множественной ассоциативностью имеет после каждого обращения следующее содержимое:

Адрес блока памяти, к которому происходит обращение	Попадание или промах	Содержимое блоков кэш-памяти после обращения			
		Набор 0	Набор 1	Набор 2	Набор 3
0	промах	**Память[0]**			
8	промах	Память[0]	**Память[8]**		
0	попадание	Память[0]	Память[8]		
6	промах	Память[0]	**Память[6]**		
8	промах	**Память[8]**	Память[6]		

Заметьте, что при обращении к блоку 6 он заменяется блоком 8, поскольку блок имел более длительное отсутствие обращений, чем блок 0. У кэш-памяти с двухканальной ассоциативностью происходит четыре промаха, на один меньше, чем у кэш-памяти с непосредственным отображением.

У полностью ассоциативной кэш-памяти имеется четыре блока (в одном наборе); каждый блок памяти может быть сохранен в любом блоке кэш-памяти. У полностью ассоциативной кэш-памяти наблюдается наивысшая производительность со всего лишь тремя промахами:

Адрес блока памяти, к которому происходит обращение	Попадание или промах	Содержимое блоков кэш-памяти после обращения			
		Блок 0	Блок 1	Блок 2	Блок 3
0	промах	**Память[0]**			
8	промах	Память[0]	**Память[8]**		
0	попадание	Память[0]	Память[8]		
6	промах	Память[0]	Память[8]	**Память[6]**	
8	попадание	Память[0]	Память[8]	Память[6]	

Для данной серии обращений три промаха — наилучший возможный результат, поскольку обращения осуществляются к трем уникальным адресам блоков. Учтите, что при использовании в кэш-памяти восьми блоков в двухканальной ассоциативной кэш-памяти вообще не было бы замен (проверьте это самостоятельно), и у нее было бы такое же количество промахов, как и у кэш-памяти с полной ассоциативностью. Аналогично этому, если бы использовались 16 блоков, то все три кэш-памяти имели бы одинаковое количество промахов. Даже этот простейший пример показывает, что размер кэш-памяти и ассоциативность весьма зависимы при определении производительности кэш-памяти.

Насколько сократился коэффициент промахов за счет ассоциативности? В табл. 5.3 показано улучшение для кэш-памяти данных объемом 64 Кбайт, име-

ющей блок из 16 слов, и ассоциативность в диапазоне от непосредственного отображения и до восьми каналов. При переходе от одноканальной к двухканальной ассоциативности коэффициент промахов снижается примерно на 15%, но результаты его снижения при переходе к большему количеству каналов ассоциативности не столь впечатляющие.

Таблица 5.3. Коэффициент промахов в кэш-памяти данных для организации, подобной той, что используется в процессоре Intrinsity FastMATH при выполнении контрольных задач SPEC2000 с ассоциативностью, варьирующейся от одно- до восьмиканальной. Эти результаты для 10 программ SPEC2000 взяты из (Hennessy and Patterson, 2003)

Ассоциативность	Коэффициент промахов при обращении к данным
1	10,3%
2	8,6%
4	8,3%
8	8,1%

Расположение блока в кэш-памяти

Теперь давайте рассмотрим задачу поиска блока в кэш-памяти с множественной ассоциативностью. Как и в кэш-памяти с непосредственным отображением, каждый блок в кэш-памяти с множественной ассоциативностью включает в себя адресный тег, который дает адрес блока. Тег каждого блока кэш-памяти внутри соответствующего набора проверяется на предмет соответствия адресу блока из процессора. На рис. 5.13 показаны составные части адреса. Значение индекса используется для выбора набора, содержащего интересующий адрес, а теги всех блоков в наборе должны быть просмотрены. Поскольку здесь важна скорость, все теги в выбранном наборе просматриваются в параллельном режиме. Как и в полностью ассоциативной кэш-памяти, при последовательном поиске время попадания будет для кэш-памяти с множественной ассоциативностью слишком большим.

Рис. 5.13. Три составляющие адреса в кэш-памяти с множественной ассоциативностью или с непосредственным отображением. Индекс используется для выбора набора, затем для выбора блока путем сравнения с блоком в выбранном наборе используется тег. Смещение блока является адресом нужных данных внутри блока

Если общий размер кэш-памяти будет таким же, повышение количества каналов ассоциативности увеличит количество блоков в наборе, что будет соответствовать количеству одновременных сравнений, необходимых для осуществления параллельного поиска: каждое удвоение каналов ассоциативности удваивает количество блоков в наборе и уменьшает наполовину количество наборов. В соответствии с этим каждое удвоение каналов ассоциативности уменьшает размер индекса на

один разряд и увеличивает на один разряд размер тега. В полностью ассоциативной кэш-памяти имеется только один набор, и все блоки должны проверяться в параллельном режиме. Поэтому в ней отсутствуют индексы, и с тегом каждого блока сравнивается весь адрес, исключая смещение блока. Иными словами, поиск ведется во всей кэш-памяти без какой-либо индексации.

В кэш-памяти с непосредственным отображением нужен всего один компаратор, поскольку запись может быть только в одном блоке, и обращение к кэш-памяти происходит просто за счет индексации. На рис. 5.14 показано, что для четырехканальной ассоциативной кэш-памяти требуются четыре компаратора, а также мультиплексор 4-к-1, чтобы сделать выбор из четырех потенциальных представителей найденного набора. Доступ к кэш-памяти состоит из индексации соответствующего набора с последующим поиском тегов набора. В дополнительную стоимость ассоциативной кэш-памяти входят компараторы и любые задержки, налагаемые необходимостью сравнений и выборов среди элементов набора.

Выбор в любой иерархии памяти между непосредственным отображением, а также множественной или полной ассоциативностью будет зависеть от стоимости промаха по сравнению со стоимостью реализации ассоциативности, как во времени, так и в необходимости использования дополнительного оборудования.

Уточнение. Память, адресуемая содержимым, или ассоциативная память (Content Addressable Memory — CAM), является электрической схемой, сочетающей в одном устройстве сравнение и хранение данных. Вместо предоставления адреса и чтения слова как в RAM, вы предоставляете данные, а CAM-память производит поиск на наличие копии и возвращает индекс соответствующей строки. Ассоциативная память позволяет разработчикам кэш-памяти реализовать набор ассоциативности более высокого уровня, чем тот, для которого необходимо собирать оборудование из SRAM-элементов и компараторов. В 2008 году более крупный размер и потребляемая мощность CAM-памяти в целом привели к тому, что двух- и четырехканальная ассоциативная память создавалась из стандартных SRAM-элементов, а восьмиканальная и выше создавалась с использованием CAM-элементов.

Выбор блока для замены

Когда промах происходит при обращении к кэш-памяти с непосредственным отображением, запрошенный блок должен помещаться только в одно вполне определенное место, заменяя собой тот блок, который занимает это место. В ассоциативной кэш-памяти у нас есть выбор, куда поместить запрошенный блок, а следовательно, и выбор, какой блок заменить. В полностью ассоциативной кэш-памяти кандидатами на замену являются все блоки. В кэш-памяти с множественной ассоциативностью нам нужно сделать выбор из блоков намеченного набора.

Наиболее часто применяемой является **схема с наиболее длительным отсутствием обращений** (least recently used — LRU), которая использовалась в предыдущем упражнении.

Схема с наиболее длительным отсутствием обращений (LRU) Схема замены, в которой заменяемым является блок, не востребованный в течение наиболее длительного периода времени.

В LRU-схеме заменяется тот блок, который не был востребован наиболее длительный период времени. В примере кэш-памяти с множественной ассоциативностью из предыдущего упражнения использовалась LRU-схема, именно поэтому блок Память(0) был заменен блоком Память(6).

LRU-замена реализуется через отслеживание момента использования каждого элемента в наборе по отношению к другим элементам набора. Для двухканальной ассоциативной кэш-памяти отслеживание момента использования может быть реализовано за счет содержания одного разряда в каждом наборе и установки его значения для индикации элемента при обращении к нему. При увеличении количества каналов ассоциативности реализация LRU дается труднее; в разделе 5.5 будет показана альтернативная схема замены.

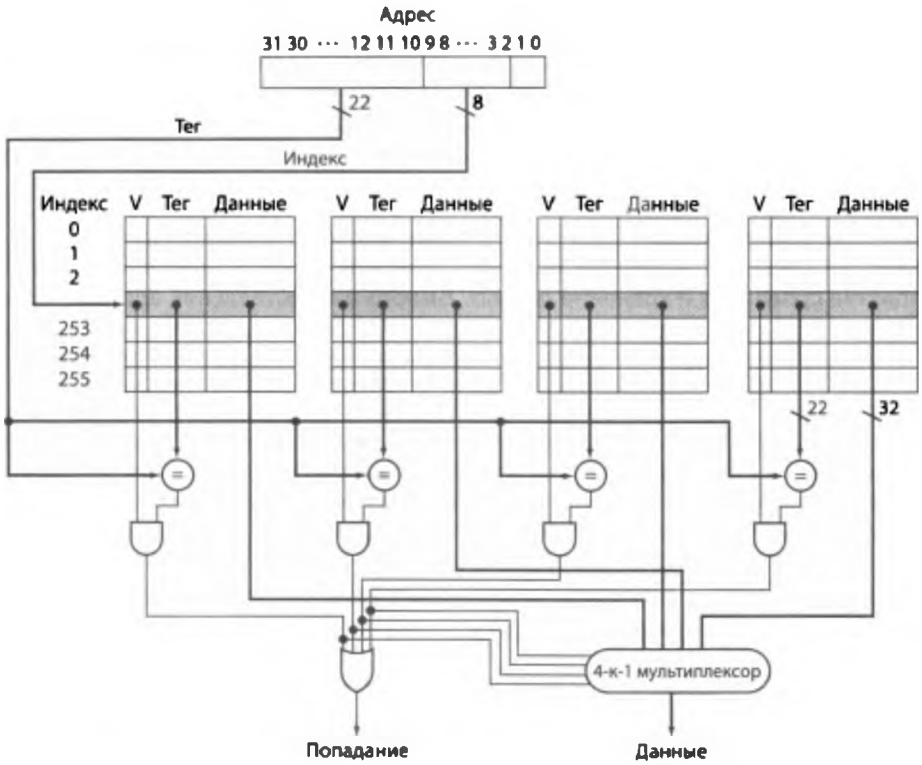


Рис. 5.14. Реализация четырехканальной ассоциативной кэш-памяти требует четырех компараторов и мультиплексора 4-к-1. Компараторы определяют, какой элемент избранного набора (если таковой имеется) соответствует тегу. Выход компаратора используется для выбора данных из одного из четырех блоков проиндексированного набора, с использованием мультиплексора с декодированным сигналом выбора. В некоторых реализациях сигналы разрешения вывода составляющей данных, имеющиеся в RAM-элементах кэш-памяти, могут быть использованы для выбора записи в наборе, который управляет выводом. Сигнал разрешения вывода поступает из компараторов, заставляя соответствующий элемент управлять выводами данных. Такая организация исключает необходимость использования мультиплексора

Упражнение

Зависимость размера тегов от количества каналов ассоциативности

Увеличение каналов ассоциативности требует использования большего количества компараторов и большей разрядности тегов на блок кэш-памяти. При условии, что кэш-память состоит из 4 К блоков, каждый блок состоит из четырех слов и используется 32-разрядная адресация, определите общее количество наборов и общее количество битов тегов для кэш-памяти с непосредственным отображением, а также для кэш-памяти с двухканальной, четырехканальной и полной ассоциативностью.

Ответ

Поскольку мы имеем 16 (= 2^4) байт на блок, 32-разрядная адресация дает 32 – 4 = 28 разрядов для использования в качестве индекса и тега. В кэш-памяти с непосредственным отображением имеется такое же количество наборов, как и количество блоков, и, следовательно, 12 разрядов на индекс, поскольку $\log_2(4\ К) = 12$; следовательно, общее количество составляет (28 – 12) × 4 К = 16 × 4 К = 64 Кбит тегов.

Каждая степень ассоциативности уменьшает количество наборов вдвое, уменьшая таким образом количество бит, используемых для индексации кэш-памяти на единицу, и увеличивая количество бит на тег также на единицу. Таким образом, для двухканальной ассоциативной кэш-памяти получится 2 К наборов, а общее количество битов на теги будет равно (28 – 11) × × 2 × 2 К = 34 × 2 К = 68 Кбит. Для четырехканальной ассоциативной кэш-памяти общее количество наборов будет равно 1 К, а общее количество битов тегов будет равно (28 – 10) × × 4 × 1 К = 72 × 1 К = 72 Кбит.

В полностью ассоциативной кэш-памяти будет только один набор из 4 К блоков, и тег из 28 бит, что составит 28 × 4 К × 1 = 112 Кбит тегов.

Уменьшение издержек промахов за счет использования многоуровневой кэш-памяти

Кэш-память используется на всех современных компьютерах. Для ликвидации растущего разрыва между высокими тактовыми частотами современных процессоров и все более длительным временем, требующимся для доступа к DRAM-элементам, многие микропроцессоры поддерживают дополнительный уровень кэширования. Кэш-память второго уровня обычно находится на том же самом кристалле, и обращение к ней происходит в случае промаха при обращении к первичной кэш-памяти. Если кэш-память второго уровня содержит нужные данные, издержки промаха при обращении к кэш-памяти первого уровня будут представлять собой время доступа к кэш-памяти второго уровня, которое будет намного меньше времени доступа к оперативной памяти. Если данных не будет в кэш-памяти первого и второго уровня, потребуется обращение к оперативной памяти, что приведет к более существенным издержкам промаха.

Насколько существенным будет улучшение производительности благодаря использованию вторичной кэш-памяти? Ответ на этот вопрос даст следующее упражнение.

Упражнение

ПРоизводительность многоуровневой кэш-памяти

Предположим, что у нас есть процессор с базовым показателем CPI, равным единице при условии успешности всех обращений в первичной кэш-памяти, и тактовой частотой 4 ГГц. Предположим, что время доступа к оперативной памяти составляет 100 нс, включая все обработки промахов. Допустим, что коэффициент промахов при обращении к инструкции в первичной кэш-памяти равен 2%. Насколько быстрее будет работать процессор, если будет добавлена вторичная кэш-память со временем доступа 5 нс как для попадания, так и для промаха, имеющая достаточный размер для сокращения коэффициента промаха при обращении к оперативной памяти до 0,5%?

Ответ

Издержки промаха при обращении к оперативной памяти составляют

$$\frac{100 \text{ нс}}{0{,}25 \frac{\text{нс}}{\text{тактовых циклов}}} = 400 \text{ тактовых циклов.}$$

Эффективный показатель CPI при одном уровне кэширования равен

Совокупный CPI = Основной CPI + Циклов задержки памяти на инструкцию

Для процессора с одним уровнем кэширования

Совокупный CPI = 1,0 + Циклов задержки памяти на инструкцию = 1,0 + 2% × 400 = 9

При двух уровнях кэширования промахи в первичном (или на первом уровне) кэше могут быть удовлетворены либо вторичной кэш-памятью, либо оперативной памятью. Издержки промаха при обращении к кэш-памяти второго уровня равны

$$\frac{5 \text{ нс}}{0{,}25 \frac{\text{нс}}{\text{тактовых циклов}}} = 20 \text{ тактовых циклов.}$$

Если промах остановился на вторичной кэш-памяти, то это и есть полные издержки промаха. Если промах вызвал необходимость обращения к оперативной памяти, то совокупные издержки промаха равны сумме времени доступа к вторичной кэш-памяти и времени доступа к оперативной памяти.

Таким образом, для двухуровневой кэш-памяти совокупный показатель CPI равен сумме циклов задержки из обоих уровней кэш-памяти и основного показателя CPI:

Совокупный CPI = 1 + Первичные задержки на инструкцию + Вторичные задержки на инструкцию = 1 + 2% × 20 + 0,5% × 400 = 1 + 0,4 + 2,0 = 3,4

Таким образом, процессор, имеющий вторичную кэш-память, быстрее

$$\text{в } \frac{9{,}0}{3{,}4} = 2{,}6 \text{ раза.}$$

Вместо этого мы могли бы вычислять циклы задержки путем суммирования циклов задержки тех обращений, которые завершились попаданием во вторичную кэш-память ((2% − 0,5%) × × 20 = 0,3). Те обращения, которые были направлены оперативной памяти и которые должны

включать стоимость обращения к вторичной кэш-памяти, а также время доступа к оперативной памяти, равны 0,5% × (20 + 400) = 2,1. Сумма, 1,0 + 0,3 + 2,1, опять составляет 3,4.

Конструкторский подход к первичной и вторичной кэш-памяти существенно различается, поскольку присутствие другой кэш-памяти изменяет приоритеты по сравнению с одноуровневой кэш-памятью. В частности, структура двухуровневой кэш-памяти дает возможность первичной кэш-памяти концентрироваться на минимизации времени попадания, чтобы получился более короткий тактовый цикл или было меньше стадий конвейера, позволяя при этом вторичной кэш-памяти быть сфокусированной на коэффициенте промахов с целью сокращения издержек продолжительного времени обращения к оперативной памяти.

Эффект от этих изменений, вносимых в две кэш-памяти, может быть заметен при сравнении каждой кэш-памяти с оптимальной конструкцией одноуровневой кэш-памяти. В сравнении с одноуровневой кэш-памятью первичная кэш-память системы **многоуровневой кэш-памяти** обычно меньше по размеру. Затем, в первичной кэш-памяти могут использоваться меньшие по размеру блоки, чтобы подходить меньшей по размеру кэш-памяти и сократить издержки промаха. В сравнении с первичной кэш-памятью вторичная кэш-память будет намного больше по размеру, поскольку время доступа к вторичной кэш-памяти менее критично. При значительно большем общем размере во вторичной кэш-памяти могут использоваться более крупные размеры блока, которые соответствуют размерам блоков одноуровневой кэш-памяти. В ней часто используется более высокая ассоциативность, чем в первичной кэш-памяти, поскольку она сконцентрирована на снижении коэффициента промахов.

Представление о производительности программ

Сортировка была основательно проанализирована с целью поиска наилучших алгоритмов: пузырьковая, быстрая, поразрядная сортировки — Bubble Sort, Quicksort, Radix Sort — и т. д. На рис. 5.15, *а* показано число инструкций на один элемент массива, выполняемых при поиске для поразрядной сортировки — Radix Sort в сравнении с поиском для быстрой сортировки — Quicksort. Как ожидалось, для больших массивов у Radix Sort имеются алгоритмические преимущества над Quicksort в понятиях числа операций. На рис. 5.15, *б* вместо выполняемых инструкций показано время, затрачиваемое на каждый ключ. Мы видим, что линии сначала идут по одинаковым траекториям, как на рис. 5.15, *а*, а затем линия, соответствующая Radix Sort, отклоняется по мере увеличения объема сортируемых данных. Что происходит? На рис. 5.15, *в* анализируется функция промахов обращений к кэш-памяти на один сортируемый элемент: Quicksort раз за разом имеет намного меньше промахов на сортируемый элемент.

Увы, стандартные приемы анализа зачастую игнорируют влияние иерархии памяти. Рост тактовой частоты и закон Мура позволяют конструкторам вытеснить все, что касается производительности, за пределы потока инструкций,

Многоуровневая кэш-память
Иерархия памяти с несколькими уровнями кэширования, в отличие от простой кэш-памяти и оперативной памяти.

используя важность иерархии памяти для достижения высокой производительности. Как было сказано во введении, понимание поведения иерархии памяти играет важную роль в понимании производительности программ в современных компьютерах.

Уточнение. Многоуровневая кэш-память создает ряд сложностей. Во-первых, теперь мы имеем дело с различными типами промахов и соответствующими им коэффициентами. В упражнении «Промахи и количество каналов ассоциативности в кэш-памяти» мы видели коэффициент промахов первичной кэш-памяти и **глобальный коэффициент промахов** — долю обращений, вызвавшую промахи на всех уровнях кэш-памяти. Есть также коэффициент промахов для вторичной кэш-памяти, являющийся частным от деления всех промахов во вторичной кэш-памяти на количестве обращений к ней. Этот коэффициент промахов называется **локальным коэффициентом промахов** вторичной кэш-памяти. Поскольку первичная кэш-память фильтрует обращения, особенно те, что хорошо согласуются с локальностями по времени и по месту, локальный коэффициент промахов вторичной кэш-памяти намного выше глобального коэффициента промахов. Например, для недавно упомянутого упражнения можно вычислить локальный коэффициент промахов: 0,5/2 = 25%! Хорошо, что глобальный коэффициент промахов определяет, насколько часто мы должны обращаться к оперативной памяти.

Уточнение. В процессорах, не придерживающихся порядка при выполнении инструкций (см. главу 4), определение производительности усложняется, поскольку они выполняют инструкции во время издержек промахов. Вместо коэффициента промахов при обращении к инструкциям и коэффициента промахов при обращении к данным используется количество промахов на инструкцию, и следующая формула:

$$\frac{\text{Циклы ожидания памяти}}{\text{На инструкцию}} = \frac{\text{Промахи}}{\text{На инструкцию}} \times$$

$$\times (\text{Общая латентность промаха} - \text{Перекрытая латентность промаха})$$

Общих способов расчета перекрытой латентности промаха не существует, поэтому оценки иерархий памяти для процессоров, не придерживающихся порядка выполнения инструкций, неминуемо требуют имитации процессора и иерархии памяти. Только путем наблюдения за выполнением инструкций процессором во время каждого промаха можно увидеть, будет ли процессор простаивать в ожидании данных или просто найдет для себя другое занятие. Ориентиром служит то, что процессор зачастую скрывает издержки промахов для кэш-памяти L1, которые являются попаданиями в кэш-памяти L2, но он редко скрывает промахи кэш-памяти L2.

Уточнение. Проблема производительности для алгоритмов заключается в том, что иерархия памяти в разных реализациях одной и той же архитектуры варьируется по размерам кэш-памяти, ассоциативности, размерам блока и количеству уровней кэш-памяти. Чтобы справиться с таким разнообразием, некоторые последние библиотеки для работы с числами задают параметры в своих алгоритмах, а затем в процессе выполнения осуществляют поиск пространства параметров, чтобы найти наилучшую комбинацию для конкретного компьютера. Такой подход называется *автонастройкой*.

Глобальный коэффициент промахов
Доля обращений, заканчивающихся промахами на всех уровнях многоуровневой кэш-памяти.

Локальный коэффициент промахов
Доля обращений к одному уровню кэш-памяти, заканчивающихся промахами; используется в многоуровневых иерархиях.

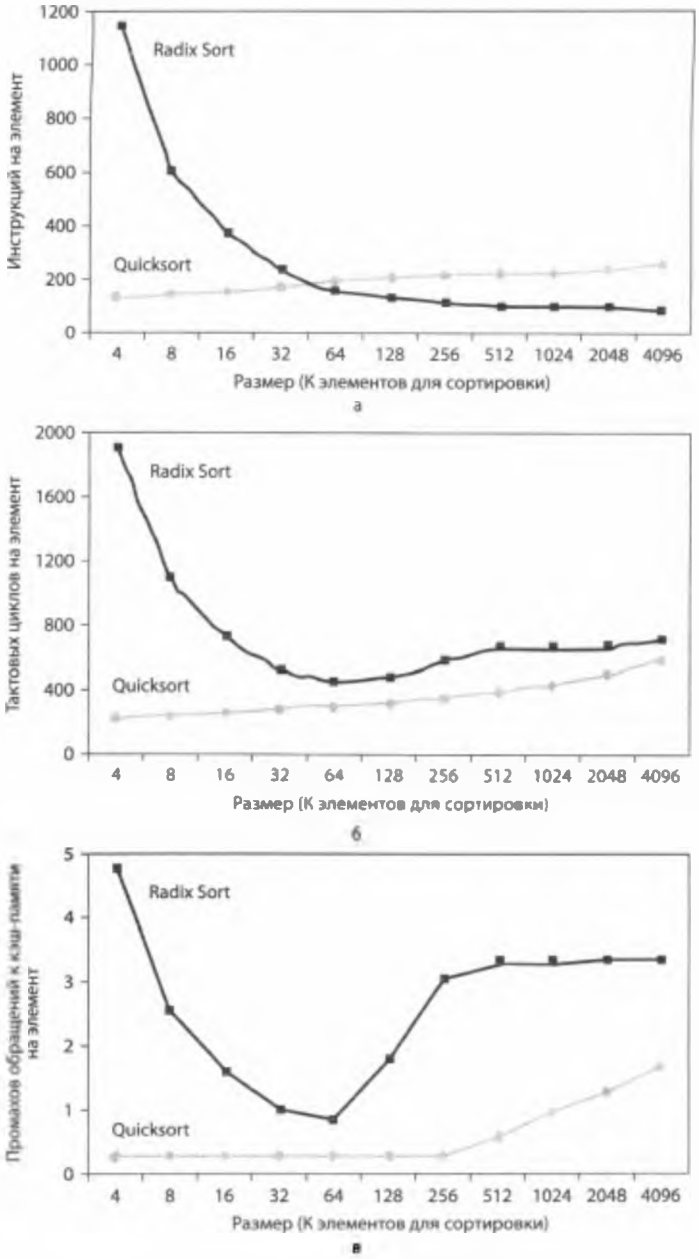


Рис. 5.15. Сравнение Quicksort и Radix Sort по: *а)* инструкциям, выполненным на сортируемый элемент, *б)* времени на сортируемый элемент и *в)* промахам обращений к кэш-памяти на сортируемый элемент. Эти данные взяты из статьи Ла Марка и Ладнера (LaMarc and Ladner, 1996). Хотя количественные характеристики для более новых компьютеров будут изменяться, общие тенденции сохранятся. Судя по результатам поиска, были изобретены новые версии Radix Sort, в которых учитывается иерархия памяти с целью восстановления алгоритмических преимуществ данного способа сортировки (см. раздел 5.10). Основной замысел оптимизации под кэш-память заключается в повторном использовании всех данных в блоке до того, как он будет заменен в случае промаха

Самопроверка

Что из нижеперечисленного справедливо для устройств, имеющих многоуровневую кэш-память?

1. Для кэш-памяти первого уровня больше внимания уделяется времени попадания, а для кэш-памяти второго уровня более важным фактором является коэффициент промахов.
2. Для кэш-памяти первого уровня больше внимания уделяется коэффициенту промахов, а для кэш-памяти второго уровня более важным фактором является время попадания.

Краткие выводы

В данном разделе основное внимание уделялось трем темам: производительности кэш-памяти, использованию ассоциативности для уменьшения коэффициента промахов и использованию иерархий с многоуровневой кэш-памятью для сокращения издержек промахов.

Система памяти оказывает существенное влияние на время выполнения программы. Количество циклов ожидания памяти зависит как от коэффициента промахов, так и от издержек промахов. Задача, как будет показано в разделе 5.5, заключается в уменьшении одного из этих факторов без существенного влияния на другие критические факторы в иерархии памяти.

Мы изучили использование схем ассоциативного размещения, позволяющих уменьшить коэффициент промахов. Эти схемы могут уменьшить коэффициент промахов кэш-памяти, допуская в ней более гибкое размещение блоков. Полностью ассоциативные схемы позволяют помещать блоки в любое место, но при этом требуют для удовлетворения запроса вести поиск в каждом блоке. Более высокая стоимость делает большую, полностью ассоциативную кэш-память непрактичной. Кэш-память с множественной ассоциативностью является более практичной альтернативой, поскольку поиск требуется вести только среди элементов уникального набора, выбираемого с помощью индексации. Такая кэш-память имеет более высокий коэффициент промахов, но более быстрый доступ. Степень ассоциативности, приводящая к наилучшей производительности, зависит как от применяемой технологии, так и от особенностей реализации.

И наконец, мы рассмотрели многоуровневую кэш-память в качестве технологии, сокращающей издержки промахов за счет того, что более крупная по размеру вторичная кэш-память может обрабатывать промахи, возникшие при обращении к первичной кэш-памяти. Кэш-память второго уровня стала популярной, как только конструкторы убедились в том, что ограниченные пространства кремниевого кристалла и стремление повысить тактовую частоту не позволяют увеличить размер первичной кэш-памяти. Вторичная кэш-память, которая зачастую в десять и более раз больше первичной кэш-памяти, обрабатывает многие обращения, завершившиеся промахами в первичной кэш-памяти. В таких случаях издержки промаха сводятся к времени доступа к вторичной кэш-памяти (обычно это менее

десяти тактовых циклов процессора) по сравнению со временем доступа к оперативной памяти (обычно это более ста тактовых циклов процессора). Как и в случае с ассоциативностью, конструкторский компромисс между размером вторичной кэш-памяти и временем доступа к ней зависит от ряда аспектов реализации.

5.4. Виртуальная память

> ...Система была разработана с расчетом на то, чтобы основная комбинация барабанов предстала перед программистом в виде одноуровневого запоминающего устройства с автоматическим выполнением необходимых передач.
>
> *Килбурн (Kilburn) и др., «One-level storage system», 1962 г.*

В предыдущем разделе мы увидели, как кэш-память предоставляет быстрый доступ к недавно использованному фрагменту программного кода и данных. Точно так же и оперативная память может работать как «кэш» для вторичного запоминающего устройства, которое обычно реализовано с помощью магнитных дисков. Такая технология называется **виртуальной памятью**. Исторически для появления виртуальной памяти были две основные мотивации: дать возможность эффективного и безопасного использования памяти сразу несколькими программами и устранить программные трудности, связанные с небольшим, ограниченным количеством оперативной памяти. Именно эти прежние причины доминируют и сегодня, спустя четыре десятилетия после изобретения этой технологии.

Рассмотрим совокупность программ, запущенных одновременно на одном и том же компьютере. Разумеется, чтобы позволить нескольким программам использовать одну и ту же память, нам нужна возможность защитить программы друг от друга, обеспечивая проведение чтения и записи программой с использованием той части оперативной памяти, которая была ей выделена. В оперативной памяти нужно содержать только активную часть многих программ, точно так же, как кэш-память содержит только активную часть одной из программ. Таким образом, принципы локальности распространяются не только на кэш-память, но и на виртуальную память, которая позволяет нам организовать эффективное совместное использование процессора, а также оперативной памяти.

При компиляции мы не можем знать, какие программы будут совместно использовать память с другими программами. Фактически, совместно использующие память программы изменяются динамически, в процессе своего выполнения. Поскольку речь идет о динамическом взаимодействии, мы предпочтем скомпилировать каждую программу в ее собственное *адресное пространство* — отдельный диапазон адресов памяти, доступный только этой программе. Виртуальная память реализует трансляцию адресного пространства программы

Виртуальная память
Технология, использующая оперативную память в качестве «кэш-памяти» для вторичного запоминающего устройства.

в **физические адреса**. Этот процесс трансляции обеспечивает **защиту** адресного пространства программы от других программ.

Вторая причина создания виртуальной памяти заключается в разрешении отдельной пользовательской программе выйти за пределы размеров первичной памяти. В прежние времена, если программа становилась слишком большой для памяти, программист должен был сделать так, чтобы она в ней поместилась. Программисты делили программы на части, а затем определяли те из них, которые были взаимоисключающими друг для друга. Эти накладываемые друг на друга части, или *оверлеи*, загружались или выгружались под управлением программы пользователя во время ее выполнения, а программист гарантировал, что эта программа никогда не будет пытаться обратиться к незагруженному оверлею и что загружаемые оверлеи никогда не выйдут за границу общего размера памяти. Оверлеи традиционно оформлялись в виде модулей, каждый из которых содержал как код, так и данные. Вызовы между процедурами в разных модулях приводили к перезаписи одного модуля поверх другого.

Как легко можно себе представить, эта ответственность была тяжелым грузом для программистов. Виртуальная память, которая была изобретена для освобождения программистов от этой обузы, автоматически управляет двумя уровнями иерархии памяти, представленными оперативной памятью (иногда называемой *физической памятью*, чтобы ее можно было отличить от виртуальной памяти) и вторичным запоминающим устройством.

Физический адрес
Адрес в оперативной памяти.

Защита
Набор механизмов для обеспечения совместного использования процессора, памяти или устройств ввода-вывода несколькими процессами без взаимных намеренно или ненамеренно создаваемых помех при чтении или записи данных друг друга. Эти механизмы также изолируют операционную систему от пользовательского процесса.

Отсутствие страницы
Событие, происходящее при отсутствии выбранной страницы в оперативной памяти.

Виртуальный адрес
Адрес, который соответствует месту в виртуальном пространстве и который при обращении к памяти транслируется (преобразуется) путем адресного отображения в физический адрес.

Преобразование адреса
Также называется отображением адреса. Процесс отображения виртуального адреса на адрес, используемый для доступа к памяти.

Хотя концепции работы виртуальной и кэш-памяти одинаковы, их разные исторические корни привели к использованию разной терминологии. Блок виртуальной памяти называется страницей, а промах при обращении к памяти называется **отсутствием страницы**. При использовании виртуальной памяти процессор генерирует **виртуальный адрес**, который транслируется комбинацией аппаратного и программного обеспечения в *физический адрес*, который в свою очередь может быть использован для доступа к оперативной памяти. На рис. 5.16 показана память с виртуальной адресацией со страницами, отображаемыми на оперативную память. Этот процесс называется *отображением адресов* или **преобразованием адресов**. Сегодня двумя уровнями иерархии памяти, контролируемыми виртуальной памятью, обычно являются DRAM-модули и магнитные диски (см. главу 1, раздел «Надежное место для хранения данных»). Если вернуться к нашей аналогии с библиотекой, виртуальный адрес можно представить названием книги, а физи-

ский адрес — тем местом, где она находится в библиотеке, которое может быть получено по шифру библиотеки Конгресса.

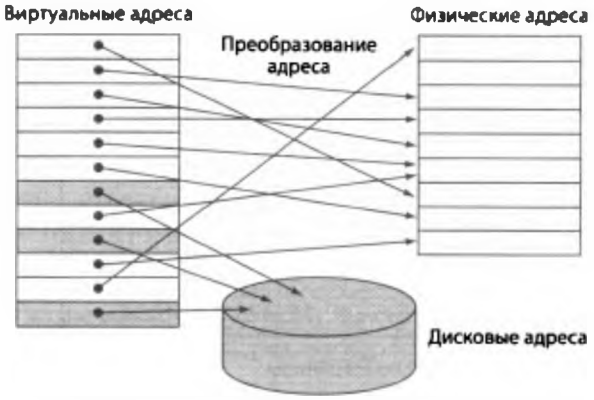


Рис. 5.16. В виртуальной памяти блоки (называемые страницами) отображаются из одного набора адресов (называемого виртуальными адресами) на другой набор (называемый физическими адресами). Процессор генерирует виртуальные адреса, а доступ к памяти осуществляется с использованием физических адресов. Как виртуальная, так и физическая память разбиты на страницы, таким образом виртуальная страница отображена на физическую страницу. Разумеется, возможно также, что виртуальная страница будет отсутствовать в оперативной памяти и не будет спроецирована на физический адрес; в таком случае страница находится на диске. Физические страницы могут совместно использоваться за счет существования двух виртуальных адресов, указывающих на один и тот же физический адрес. Эта возможность используется для того, чтобы позволить двум разным программам совместно использовать данные или код

Виртуальная память также упрощает загрузку программ для исполнения, предоставляя *перемещение*. Перемещение отображает виртуальные адреса, используемые программой, на разные физические адреса перед тем, как адреса используются для обращения к памяти. Это перемещение позволяет нам загружать программу в любое место оперативной памяти. Более того, все системы виртуальной памяти, используемые сегодня, перемещают программу в виде набора блоков фиксированной длины (страниц), исключая тем самым необходимость поиска смежного блока памяти для его распределения программе; вместо этого операционной системе нужно лишь найти достаточное количество страниц в оперативной памяти.

В виртуальной памяти адреса разбиты на *номер виртуальной страницы* и на *смещение страницы*. На рис. 5.17 показано преобразование номера виртуальной страницы в *номер физической страницы*. Номер физической страницы составляет старшую часть физического адреса, а смещение страницы, которое не изменяется, составляет его младшую часть. Количество разрядов в поле смещения страницы определяет размер страницы. Количество страниц, адресуемых виртуально, не обязательно должно соответствовать количеству страниц, адресуемых физически. Наличие большего количества виртуальных страниц, чем физических, является основой иллюзии безграничного объема виртуальной памяти.

Рис. 5.17. Отображение виртуальных адресов на физические. Размер страницы равен 2^{12} = 4 Кбайт. Количество физических страниц, допустимое в памяти, равно 2^{18}, поскольку номер физической страницы содержит 18 разрядов. Таким образом, оперативная память может иметь не более 1 Гбайт, а виртуальное адресное пространство составляет 4 Гбайт

Во многих случаях выбранные конструкторами решения в системах виртуальной памяти мотивируются высокой стоимостью промаха, который в виртуальной памяти традиционно называется отсутствием страницы. Обработка такого отсутствия займет миллионы тактовых циклов. (В таблице в самом начале главы показано, что латентность у оперативной памяти примерно в 100 000 раз ниже, чем у диска.) Такие огромные издержки промаха, основную часть которых составляет время получения первого слова для обычных размеров страницы, привели к ряду ключевых решений в конструировании систем виртуальной памяти:

- Страницы должны быть достаточно большими, чтобы попытаться амортизировать большое время доступа. Сегодня обычными являются размеры от 4 Кбайт до 16 Кбайт. Новые настольные и серверные системы были разработаны для поддержки страниц размером в 32 Кбайт и 64 Кбайт, но новые встроенные системы следуют в другом направлении, к страницам размером в 1 Кбайт.
- Привлекают способы организации, уменьшающие коэффициент отсутствия страниц. Здесь главной применяемой технологией является допуск к полностью ассоциативному размещению страниц в памяти.
- Отсутствия страниц могут быть обработаны программным обеспечением, поскольку издержки по сравнению со временем доступа к диску будут незначительны. Кроме этого, программное обеспечение может дать возможность использования хорошо продуманных алгоритмов для выбора способа разме-

шения страниц, поскольку даже небольшое снижение коэффициента промахов оправдает стоимость таких алгоритмов.

- Сквозная запись для виртуальной памяти работать не будет, поскольку запись занимает слишком много времени. Вместо этого в системах виртуальной памяти используется отложенная запись.

К этим факторам конструкции виртуальной памяти обращаются несколько следующих подразделов.

Уточнение. Хотя обычно считается, что виртуальных адресов намного больше, чем физических, может быть и наоборот, когда размер адреса процессора меньше по отношению к состоянию технологии памяти. Преимущества от отсутствия свопинга в памяти или от запуска на параллельных процессорах может получить не отдельно взятая программа, а совокупность программ, запущенных одновременно. Для серверов и настольных компьютеров 32-разрядные адреса проблематичны.

Уточнение. Рассмотрение виртуальной памяти в данной книге фокусируется на страничной организации памяти, использующей блоки фиксированного размера. Существует также и схема с изменяемым размером блока, которая называется **сегментацией**. При сегментации адрес состоит из двух частей: номера сегмента и смещения сегмента. Сегментный регистр отображается на физический адрес, а смещение добавляется, чтобы определить действительный физический адрес. Поскольку сегмент может быть разного размера, необходима также проверка границ, чтобы гарантировать нахождение смещения внутри сегмента. В основном сегментация используется для поддержки более мощных методов защиты и совместного использования адресного пространства. Многие руководства по операционным системам содержат развернутые обсуждения вопроса сегментации по сравнению со страничной организацией и использования сегментации для логического разделения адресного пространства. Основной недостаток сегментации в том, что она разбивает адресное пространство на логически отдельные части, работать с которыми нужно через адрес, состоящий из двух частей: номер сегмента и смещение. В отличие от этого, страничная организация памяти выстраивает границу между номером страницы и смещением, невидимую для программистов и компиляторов.

Сегменты также использовались как способ расширения адресного пространства без изменения размера слова компьютера. Подобные попытки не привели к успеху из-за громоздкости и издержек производительности, унаследованных от адреса, состоящего из двух частей, о которых должны знать программисты и компиляторы.

Во многих архитектурах адресное пространство делилось на большие блоки фиксированной длины, упрощавшие защиту между операционной системой и пользовательскими программами и повышавшие эффективность реализации страничной организации памяти. Хотя такие деления часто называли «сегментами», этот механизм намного проще, чем сегментация с переменным размером блоков, и невидим для пользовательских программ; далее он будет рассмотрен более подробно.

Сегментация
Схема отображения адресов переменного размера, в которой адрес состоит из двух частей: номера сегмента, проецируемого на физический адрес, и сегментного смещения.

Размещение страницы и ее последующий поиск

Поскольку отсутствие страницы приводит к очень большим издержкам, конструкторы сократили частоту таких отсутствий, оптимизировав размещение страниц. Если мы допускаем отображение виртуальной страницы на любую физическую страницу, то операционная система в случае отсутствия страницы может заменить ею любую страницу по своему усмотрению. Например, операционная система может использовать довольно сложный алгоритм и непростую структуру данных для отслеживания востребованности страницы, чтобы попытаться выбрать страницу, которая не понадобится еще долгое время. Возможность использования хорошо продуманной и гибкой схемы замены уменьшает коэффициент отсутствия страниц и упрощает использование полностью ассоциативного размещения страниц.

Как уже упоминалось в разделе 5.3, трудности использования полностью ассоциативного размещения заключаются в определении местоположения нужной записи, поскольку она может быть в любом месте верхнего уровня иерархии. Всеобъемлющий поиск здесь непрактичен. В системах виртуальной памяти местоположение страниц определяется с помощью таблицы, индексирующей память; такая структура называется **таблицей страниц** и размещается в памяти. Таблица страниц для нахождения соответствующего номера физической страницы индексируется по номеру страницы из виртуального адреса. У каждой программы имеется своя собственная таблица страниц, отображающая виртуальное адресное пространство этой программы на оперативную память. В нашей аналогии с библиотекой таблица страниц соответствует связи между названиями книг и их местами на полках в библиотеке. Точно так же, как карточка каталога может содержать записи для книг в другой библиотеке студенческого городка, а не только в библиотеке местного отделения, мы увидим, что таблица страниц может содержать записи для страниц, не присутствующих в памяти. Для обозначения местонахождения таблицы страниц в памяти, оборудование включает в себя регистр, указывающий на начало таблицы страниц; мы можем назвать его *регистром таблицы страниц*. Предположим пока, что таблица страниц находится во вполне определенной и неразрывной области памяти.

На рис. 5.18 используются регистр таблицы страниц, виртуальный адрес и обозначенная таблица страниц для того, чтобы показать, как оборудование может сформировать физический адрес. Бит достоверности используется в каждой записи таблицы страниц, точно так же, как он использовался в кэш-памяти. Если этот бит сброшен, страница отсутствует в оперативной памяти, и возникает ошибка отсутствия страницы. Если бит установлен, значит, страница находится в памяти и запись содержит физический номер страницы.

Таблица страниц
Таблица в системе виртуальной памяти, содержащая перевод виртуального адреса в физический. Эта таблица, которая хранится в памяти, обычно индексируется по номеру виртуальной страницы; каждая запись таблицы содержит номер физической страницы для данной виртуальной страницы, если страница на данный момент находится в памяти.

Поскольку в таблице страниц содержится отображение для каждой возможной виртуальной страницы, теги не нужны. Используя терминологию кэш-памяти, индекс, используемый для доступа к таблице страниц, состоит из полного адреса блока, который в данном случае является виртуальным номером страницы.

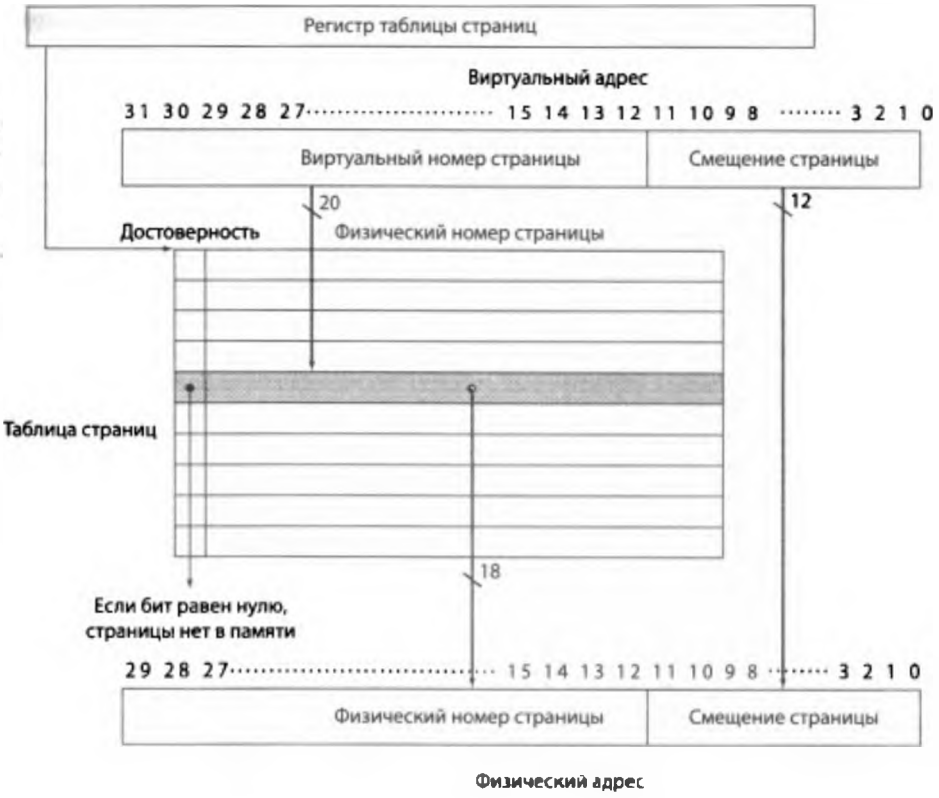


Рис. 5.18. Таблица страниц проиндексирована по виртуальным номерам страниц для получения соответствующей части физического адреса. Предполагается использование 32-разрядного адреса. Начальный адрес таблицы страниц задается указателем таблицы страниц. На этом рисунке размер страницы равен 2^{12} байт, или 4 Кбайт. Виртуальное адресное пространство равно 2^{32} байт, или 4 Гбайт, а физическое адресное пространство равно 2^{30} байт, что позволяет иметь оперативную память объемом до 1 Гбайт. Количество записей в таблице страниц равно 2^{20}, или 1 млн записей. Бит достоверности для каждой записи показывает, является ли отображение истинным. Если он сброшен, значит, страница отсутствует в памяти. Хотя показанная здесь запись таблицы страниц может иметь только лишь 19 разрядов в ширину, эта ширина, как правило, для простоты индексирования должна быть округлена до 32 разрядов. Лишние разряды могут использоваться для сохранения дополнительной информации, которую необходимо хранить на постраничной основе, например сведения о защите

Интерфейс аппаратного и программного обеспечения

Таблица страниц вместе со счетчиком команд и регистрами определяет состояние программы. Если мы хотим разрешить другой программе использовать процессор, этот состояние нужно сохранить. Позже, после восстановления этого состояния, выполнение программы может продолжиться. Часто это состояние называют процессом. Процесс считается активным, когда им владеет процессор; в противном случае он считается неактивным. Операционная система может активизировать процесс, загрузив его состояние, включая значение счетчика, что инициирует

выполнение с того места, которое было сохранено в качестве значения счетчика команд.

Адресное пространство процесса, а следовательно, все данные, к которым у него имеется доступ в памяти, определяются его таблицей страниц, которая находится в памяти. Вместо сохранения всей таблицы страниц операционная система просто загружает в регистр таблицы страниц значение, указывающее на таблицу страниц того процесса, который она хочет сделать активным. У каждого процесса есть своя собственная таблица страниц, поскольку разные процессы используют одни и те же виртуальные адреса. Операционная система отвечает за распределение физической памяти и обновление таблицы страниц, поэтому виртуальные адресные пространства разных процессов не конфликтуют друг с другом. Далее будет показано, что использование отдельных страничных таблиц также обеспечивает защиту одного процесса от другого.

Отсутствие страницы

Если бит достоверности для виртуальной страницы сброшен, возникает ошибка отсутствия страницы. Справиться с ней должна операционная система. Передача управления происходит с помощью механизма исключений, который будет рассмотрен в данном разделе чуть позже. После того как операционная система получила управление, она должна найти страницу на следующем уровне иерархии (обычно на магнитном диске) и решить, в какое место оперативной памяти нужно записать запрошенную страницу.

Сам по себе виртуальный адрес не может тут же сообщить нам, где именно на диске находится страница. Возвращаясь к нашей аналогии с библиотекой, мы не можем найти местонахождение библиотечной книги на полках, зная лишь ее название. Для этого нужно подойти к каталогу и найти книгу, получив ее местоположение на полке. Так же и в системе виртуальной памяти, мы должны отслеживать местоположение на диске каждой страницы в виртуальном адресном пространстве.

Поскольку нам не известно заранее, когда находящаяся в памяти страница будет заменена, операционная система при создании процесса обычно оставляет место на диске для всех его страниц, которое называется **пространством свопинга**. В это же время она также создает структуру данных, чтобы записать, в каком месте диска хранится каждая виртуальная страница. Эта структура данных может быть частью таблицы страниц или может быть дополнительной структурой данных, проиндексированной точно так же, как и таблица страниц. На рис. 5.19 показана организация, где в одной и той же таблице хранится либо физический номер страницы, либо адрес на диске.

Пространство свопинга
Пространство на диске, зарезервированное для всего пространства виртуальной памяти процесса.

Операционная система также создает структуру данных, в которой отслеживается, какие

процессы и какие виртуальные адреса используют каждую физическую страницу. Когда возникает ошибка отсутствия страницы, если все страницы в оперативной памяти используются, операционная система должна выбрать страницу, которая будет заменена. Поскольку нужно минимизировать количество ошибок отсутствия страницы, большинство операционных систем пытается выбрать страницу, которая гипотетически в ближайшее время не понадобится. Используя прошлое для предсказания будущего, операционная система следует схеме замены с наиболее длительным отсутствием обращений (LRU), которая рассматривалась в разделе 5.3. Операционная система ищет наименее востребованную страницу, предполагая, что именно эта страница, которая не была использована длительное время, скорее всего не понадобится, в отличие от тех страниц, обращение к которым было сравнительно недавно. Заменяемые страницы записываются в пространство свопинга на диске. Но операционная система — это ведь точно такой же процесс, и таблицы, управляющие памятью, находятся в той же памяти; подробнее это кажущееся противоречие будет рассмотрено чуть позже.

Рис. 5.19. Таблица страниц содержит отображение каждой страницы в виртуальной памяти либо на страницу в оперативной памяти, либо на страницу, хранящуюся на диске, который представляет следующий уровень иерархии. Для индексации таблицы страниц используется виртуальный номер страницы. Если бит достоверности установлен, таблица страниц предоставляет физический номер страницы (то есть начальный адрес страницы в памяти), соответствующей виртуальной странице. Если бит достоверности сброшен, значит, на данный момент страница находится только на диске по указанному дисковому адресу. Во многих системах таблица физических адресов страниц и таблица дисковых адресов страниц, будучи логически одной и той же таблицей, хранятся в двух отдельных структурах данных. Использование двух таблиц отчасти оправданно, поскольку нам нужно хранить дисковые адреса всех страниц, даже если они на данный момент находятся в памяти. Следует помнить, что страницы в оперативной памяти и страницы на диске имеют одинаковые размеры

Интерфейс аппаратного и программного обеспечения

Реализация абсолютно точной LRU-схемы обходится слишком дорого, поскольку это требует обновления структуры данных при каждом обращении к памяти. Вместо этого многие операционные системы реализуют приблизительную LRU-схему, отслеживая, какая из страниц была, а какая не была востребована в последнее время. Чтобы помочь операционной системе определить LRU-страницы, некоторые компьютеры предоставляют **бит обращения**, или **бит использования**, который устанавливается при обращении к странице. Операционная система периодически сбрасывает биты обращения, а затем их устанавливает, благодаря чему она может определить, какие страницы были востребованы в течение определенного периода времени. Используя эту информацию, операционная система может выбрать страницу среди наименее востребованных страниц (которая определяется по сброшенному биту обращения). Если этот бит оборудованием не предоставляется, операционная система должна найти другой способ определения, к какой странице было обращение.

Уточнение. При 32-разрядной виртуальной адресации, страницах размером 4 Кбайт, и четырех байтах на каждую запись таблицы страниц мы можем вычислить общий размер таблицы страниц:

$$\text{Количество записей таблицы страниц} = 2^{32} / 2^{12} = 2^{20}$$

$$\text{Размер таблицы страниц} = 2^{20} \text{ записей таблицы страниц} \times 2^{2} \text{ (байт на запись таблицы страниц)} = 4 \text{ Мбайт}$$

То есть нам нужно задействовать 4 Мбайт памяти для каждой программы, выполняемой в одно и то же время. Этот объем вполне подходит для отдельной программы. А что если запущены сотни программ и у каждой есть своя собственная таблица страниц? И как справиться с 64-разрядными адресами, для которых согласно расчетам потребуется 2^{52} слов?

Чтобы сократить объем, необходимый для хранения таблиц страниц, используется ряд технологий. Пять нижеперечисленных технологий нацелены на сокращение востребованного общего максимального объема запоминающего устройства, а также на минимизацию оперативной памяти, выделенной под таблицы страниц:

1. Самая простая технология заключается в использовании лимитирующего регистра, ограничивающего размер таблицы страниц для заданного процесса. Если виртуальный номер страницы становится больше, чем содержимое лимитирующего регистра, записи должны быть добавлены к таблице страниц. Эта технология позволяет таблице страниц расти по мере того, как процесс поглощает больше пространства. Таким образом, таблица страниц будет большой только в том случае, если процесс использует

Бит обращения
Также называется битом использования. Поле, которое устанавливается при каждом обращении к странице, а затем используется для реализации LRU или других схем замены

много страниц виртуального адресного пространства. Эта технология требует, чтобы адресное пространство расширялось только в одном направлении.

Предоставление возможности роста только в одном направлении не является полноценным решением, поскольку большинству языков требуются две области с расширяющимся пространством: в одной из них хранится стек, а в другой «куча», которая является областью памяти, выделяемой программе для динамически размещаемых структур данных. Из-за этой двойственности удобнее будет разделить таблицу страниц и позволить ей расти с самых верхних адресов вниз, а также с самых нижних адресов вверх. Это означает наличие двух отдельных таблиц страниц и двух отдельных лимитирующих регистров. При использовании двух таблиц страниц адресное пространство разбивается на два сегмента. Какой сегмент и, следовательно, какую таблицу страниц применять для конкретного адреса, обычно определяет его старший разряд. Поскольку сегмент определяется старшим разрядом адреса, каждый сегмент может быть таким же большим, как половина адресного пространства. Лимитирующий регистр для каждого сегмента определяет текущий размер сегмента, который растет постранично. Этот тип сегментации используется многими архитектурами, включая MIPS. В отличие от типа сегментации, рассмотренного во втором уточнении данного раздела, эта форма сегментации невидима для прикладных программ, но не для операционной системы. Основной недостаток этой схемы заключается в том, что она недостаточно хорошо работает, когда адресное пространство используется вразброс, а не в виде последовательного набора виртуальных адресов.

Еще одним подходом для сокращения таблицы страниц является обработка виртуального адреса хэш-функцией, чтобы размер таблицы страниц соответствовал только лишь количеству *физических* страниц в оперативной памяти. Такая структура называется *инвертированной таблицей страниц*. Разумеется, при этом немного усложняется процесс поиска, поскольку теперь недостаточно просто проиндексировать таблицу страниц.

Для сохранения общего объема запоминающего устройства, необходимого для хранения таблиц страниц, могут быть также применены многоуровневые таблицы страниц. На первом уровне отображаются крупные блоки фиксированной длины виртуального адресного пространства, общим размером от 64 до 256 страниц. Эти крупные блоки иногда называют сегментами, а таблицу отображения первого уровня иногда называют таблицей сегментов, тем не менее эти сегменты и в данном случае не видны пользователю. Каждая запись в таблице сегментов показывает, присутствует или нет та или иная страницы в сегменте, и если да, то указывает на таблицу страниц для этого сегмента. Преобразование адреса осуществляется сначала путем просмотра таблицы сегментов, используя старшие разряды адреса. Если адрес сегмента достоверен, следующий набор старших разрядов используется для индексации таблицы страниц, указанной в записи таблицы сегментов. Эта схема позволяет адресному пространству использоваться вразброс (могут активно использоваться несмежные сегменты), не вынуждая распределять всю таблицу страниц. В частности, такие схемы полезны при очень большом адресном пространстве и в программных системах, требующих

разбросанного распределения. Основной недостаток этого двухуровневого отображения заключается в более сложном процессе преобразования адреса.

5. Для сокращения задействованной для таблиц страниц оперативной памяти большинство современных систем также допускают страничную организацию этих таблиц. Хотя это звучит несколько необычно, работает такая организация с использованием тех же основных идей виртуальной памяти и просто позволяет таблицам страниц находиться в виртуальном адресном пространстве. Кроме этого имеется ряд небольших, но важных проблем, например бесконечная серия ошибок отсутствия страницы, которой нужно избегать. То, как решаются эти проблемы, в большей степени зависит от тонкостей специфики того или иного процессора. Вкратце, эти проблемы устраняются за счет размещения всех таблиц страниц в адресном пространстве операционной системы и помещения по крайней мере некоторых из таблиц страниц операционной системы в части оперативной памяти, имеющей физическую адресацию, благодаря чему они всегда присутствуют в ней и никогда не попадают на диск.

А что насчет записи?

Разница между временем доступа к кэш-памяти и к оперативной памяти составляет от десятков до сотен циклов, и могут быть использованы схемы сквозной записи, хотя, чтобы скрыть латентность записи из процессора, нам понадобится буфер записи. В системе виртуальной памяти запись в следующий уровень иерархии (диск) занимает миллионы тактовых циклов процессора; поэтому создание буфера записи, чтобы система могла вести сквозную запись на диск, будет совершенно непрактичным решением. Вместо этого системы виртуальной памяти должны использовать отложенную запись, выполняя отдельные операции записи на страницу в памяти и копируя страницу обратно на диск, когда она заменяется в памяти.

Интерфейс аппаратного и программного обеспечения

Схема отложенной записи имеет еще одно важное преимущество в системе виртуальной памяти. Поскольку по сравнению со временем доступа к диску перенос информации на диск осуществляется быстрее, обратное копирование целиком всей страницы гораздо эффективнее записи обратно на диск отдельных слов. Операция отложенной записи, несмотря на то что она эффективнее переноса отдельных слов, имеет все же слишком затратный характер. Поэтому нам нужно знать, *нуждается ли* страница в обратном копировании, когда она выбирается для замены. Чтобы узнать, подвергалась ли страница записи после того, как она была считана в память, к таблице страниц добавляется *бит изменения* (dirty bit). Этот бит устанавливается, когда в страницу было записано какое-либо слово. Если операционная система выбрала страницу для замены, бит изменения показывает, нуждается ли эта страница в обратной записи перед тем, как ее место в памяти может быть отдано другой странице. В соответствии с этим модифицированная страница часто называется измененной (dirty page).

Ускорение преобразования адреса: TLB

Поскольку таблицы страниц хранятся в оперативной памяти, каждый доступ к памяти со стороны программы может занять, по крайней мере, вдвое больше времени: одно обращение к памяти для получения физического адреса, а второе — для получения данных. Ключом для повышения производительности доступа является расчет на локальность обращения к таблице страниц. Когда используется преобразование для виртуального номера страницы, есть вероятность того, что в ближайшем будущем оно понадобится снова, поскольку ссылки на слова на этой странице обладают локальностью, связанной как со временем, так и с пространством.

В соответствии с этим современные процессоры включают в себя специальную кэш-память, в которой отслеживаются самые последние преобразования. Эта специальная кэш-память преобразования адресов традиционно называется **буфером быстрого преобразования адреса** (translation-lookaside buffer — TLB), хотя было бы логичнее назвать ее кэш-памятью преобразований. TLB похож на небольшую записку, которая обычно используется для записи местоположения подборки тех книг, которые мы ищем в библиотечной картотеке; вместо повторного поиска во всем каталоге мы записываем местоположение нескольких книг и используем эту записку в качестве кэш-памяти шифров библиотеки Конгресса.

На рис. 5.20 показано, что каждая запись тега в TLB содержит часть виртуального номера страницы, а каждая запись данных в TLB содержит физический номер страницы. Поскольку при каждой ссылке мы обращаемся к TLB вместо таблицы страниц, в TLB нужно включить другие биты статуса, такие как бит изменения и бит обращения.

При каждом обращении ведется поиск виртуального номера страницы в TLB. Если будет попадание, для формирования адреса используется физический номер страницы и устанавливается в единицу соответствующий бит обращения. Если процессор осуществляет запись, также в единицу устанавливается бит изменения. Если происходит промах при обращении к TLB, нужно определить, что это — ошибка отсутствия страницы или просто промах при обращении к TLB. Если страница присутствует в памяти, тогда промах при обращении к TLB свидетельствует только лишь о том, что произошел промах при получении результата преобразования адреса. В таких случаях процессор может справиться с промахом при обращении к TLB путем загрузки результата преобразования из таблицы страниц в TLB с последующей повторной попыткой обращения. Если страницы нет в памяти, тогда промах при обращении к TLB свидетельствует о возникновении ошибки отсутствия страницы. В таком случае процессор обращается к операционной системе, используя исключение. Поскольку в TLB количество записей намного меньше количества страниц в оперативной памяти, промахи при обращении к TLB будут случаться намного чаще, чем настоящие ошибки отсутствия страницы.

Промахи при обращении к TLB могут быть обработаны аппаратным или программным способом. Если разобраться, то на практике разница в производительности между этими двумя подходами невелика, потому что в обоих случаях применяются одни и те же основные операции.

Буфер быстрого преобразования адреса (TLB)
Кэш-память, отслеживающая недавно использованные отображения адресов, чтобы попытаться избежать обращения к таблице страниц.

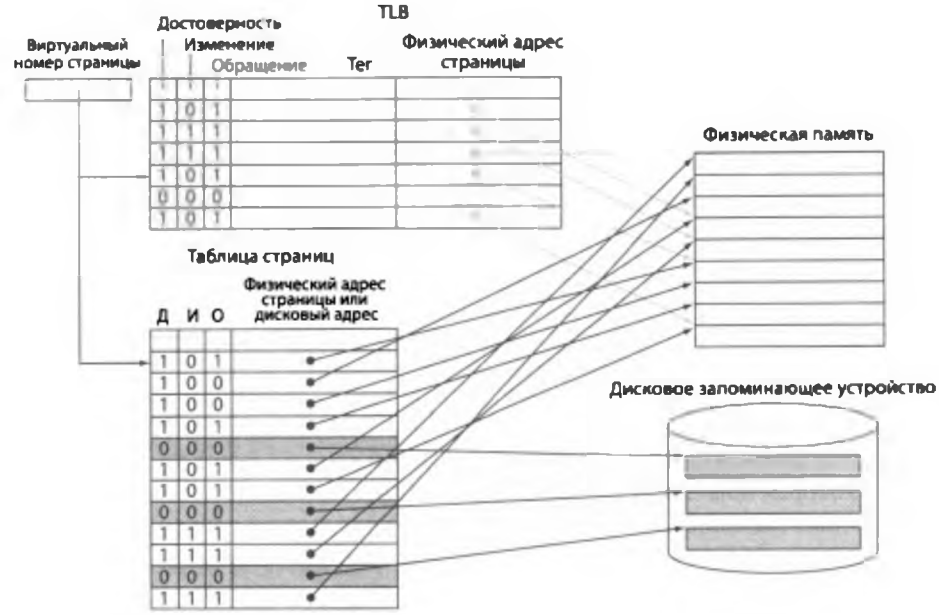


Рис. 5.20. TLB работает как кэш-память таблицы страниц для записей, которые отображаются только на физические страницы. TLB содержит поднабор отображений виртуальных страниц на физические, имеющиеся в таблице страниц. Отображения TLB показаны другим цветом. Поскольку TLB является кэш-памятью, у нее должно быть поле тега. Если в TLB нет записи, соответствующей странице, нужно обращаться к таблице страниц. Таблица страниц либо предоставит физический номер страницы (который может быть использован для создания записи TLB), либо покажет, что страница находится на диске, в таком случае возникнет ошибка отсутствия страницы. Поскольку таблица страниц содержит запись для каждой виртуальной страницы, поле тега ей не нужно; иными словами, в отличие от TLB, таблица страниц *не является* кэш-памятью

После того как при промахе во время обращения к TLB отсутствующий результат преобразования извлечен из таблицы страниц, нам нужно будет выбрать запись в TLB, которая будет заменена. Поскольку в составе записи TLB есть биты изменения и обращения, то при замене записи эти биты нужно скопировать обратно в таблицу страниц. Эти биты являются единственной частью записи TLB, которая может быть изменена. Использование отложенной записи, то есть копирование этих записей обратно при промахе, а не тогда, когда в них была произведена запись, — очень эффективное решение, поскольку ожидается, что коэффициент промахов при обращении к TLB будет небольшим. На некоторых системах используются другие технологии, приблизительно похожие на использование битов обращения и изменения, исключающие необходимость записи в TLB, кроме внесения новой записи при промахе.

У TLB могут быть следующие типовые параметры:

- размер TLB: 16–512 записей;
- размер блока: 1–2 записи таблицы страниц (обычно по 4–8 байт каждая);

- время попадания: 0,5–1 тактовый цикл;
- издержки промаха: 10–100 тактовых циклов;
- коэффициент промахов: 0,01%–1%.

Конструкторы используют в буферах TLB широкое разнообразие технологий ассоциативности. На некоторых системах используются небольшие, полностью ассоциативные буферы TLB, потому что у полностью ассоциативного отображения меньше коэффициент промахов; более того, поскольку TLB имеет небольшой размер, стоимость полностью ассоциативного отображения не слишком высока. В других системах используются более крупные TLB, зачастую с небольшой степенью ассоциативности. При полностью ассоциативном отображении выбрать запись для замены становится труднее, потому что реализация аппаратной LRU-схемы обходится слишком дорого. Более того, поскольку промахи при обращении к TLB случаются намного чаще, чем ошибки отсутствия страницы, и поэтому должны обрабатываться более дешевыми средствами, мы не можем использовать для этого слишком накладный программный алгоритм, допустимый при обработке ошибок отсутствия страниц. В результате этого многие системы обеспечивают поддержку произвольного выбора заменяемой записи. Схемы замены более подробно будут рассмотрены в разделе 5.5.

Буфер TLB процессора Intrinsity FastMATH

Чтобы понять, как работают эти конструкторские замыслы на настоящем процессоре, давайте более пристально рассмотрим TLB процессор Intrinsity FastMATH. Система памяти использует страницы объемом 4 Кбайт и 32-разрядное адресное пространство; таким образом, виртуальные номера страниц, как показано в верхней части рис. 5.21, имеют длину 20 разрядов. Физические адреса имеют те же размеры, что и виртуальные. Буфер TLB содержит 16 записей, является полностью ассоциативным и совместно используется при обращениях к инструкциям и к данным. Каждая запись имеет ширину 64 разряда и содержит 20-разрядный тег (который для данной записи TLB является виртуальным номером страницы), соответствующий физический номер страницы (также 20 разрядов), бит достоверности, бит изменения и другие регистрационные биты.

На рис. 5.21 показан TLB и одно из устройств кэш-памяти, а на рис. 5.22 показаны шаги обработки запроса на чтение или запись. Когда происходит промах при обращении к TLB, аппаратное обеспечение MIPS сохраняет номер страницы, к которой было обращение, в специальном регистре и генерирует исключение. Это исключение активизирует операционную систему, которая обрабатывает промах программным способом. Для определения физического адреса отсутствующей страницы подпрограмма промаха TLB индексирует таблицу страниц, используя номер страницы в виртуальном адресном пространстве и регистр таблицы страниц, который указывает на начальный адрес таблицы страниц активного процесса. Используя специальный набор системных инструкций, который способен обновить TLB, операционная система помещает физический адрес из таблицы страниц в TLB. Промах при обращении к TLB отнимает около 13 тактовых циклов, при ус-

ловии, что код и запись таблицы страниц находятся соответственно в кэш-памяти инструкций и в кэш-памяти данных. (Код MIPS TLB будет показан далее в разделе «Обработка промахов при обращении к TLB и ошибок отсутствия страницы».) Настоящая ошибка отсутствия страницы происходит в том случае, если запись в таблице страниц не содержит достоверного физического адреса. Оборудование задает индекс, указывающий на рекомендуемую для замены запись; рекомендуемая запись выбирается произвольно.

Для запросов на запись существуют дополнительные трудности: а именно в TLB должен быть проверен бит доступа для записи. Этот бит мешает программе вести запись в страницы, к которым у нее есть допуск только для чтения. Если программа пытается вести запись, а бит доступа к записи сброшен, выдается исключение. Бит доступа для записи является частью механизма защиты, который будет вскоре рассмотрен.

Объединение виртуальной памяти, буферов TLB и устройств кэш-памяти

Наша виртуальная память и система кэш-памяти работают вместе в рамках одной иерархии, поэтому данные не могут быть в кэш-памяти, пока они не будут присутствовать в оперативной памяти. Операционная система помогает обслуживать эту иерархию путем сбрасывания на диск содержимого любой страницы из кэш-памяти, когда она решит переместить эту страницу на диск. В то же самое время операционная система вносит изменения в таблицу страниц и в TLB, чтобы попытка обращения к любым данным в перемещенной странице приводила к генерированию ошибки отсутствия страницы.

При самых благоприятных обстоятельствах виртуальный адрес транслируется буфером TLB и отправляется в кэш-память, где соответствующие данные разыскиваются, извлекаются и отправляются процессору. В наихудшем случае обращение может потерпеть промах во всех трех компонентах иерархии памяти: в TLB, в таблице страниц и в кэш-памяти. Более подробно это взаимодействие рассмотрено в следующем упражнении.

Упражнение

Вся операция в иерархии памяти

Иерархия памяти (рис. 5.21) включает в себя TLB и кэш-память, обращение к памяти может столкнуться с тремя разными типами промахов: промахом при обращении к TLB, ошибкой отсутствия страницы и промахом при обращении к кэш-памяти. Рассмотрите все комбинации этих трех событий, при которых случается одно из них или сразу несколько событий (всего имеется семь возможных вариантов). Для каждой возможности установите, может ли это событие произойти на самом деле и при каких обстоятельствах.

Ответ

В табл. 5.4 показаны все комбинации и рассмотрены возможности их появления в реальных устройствах.

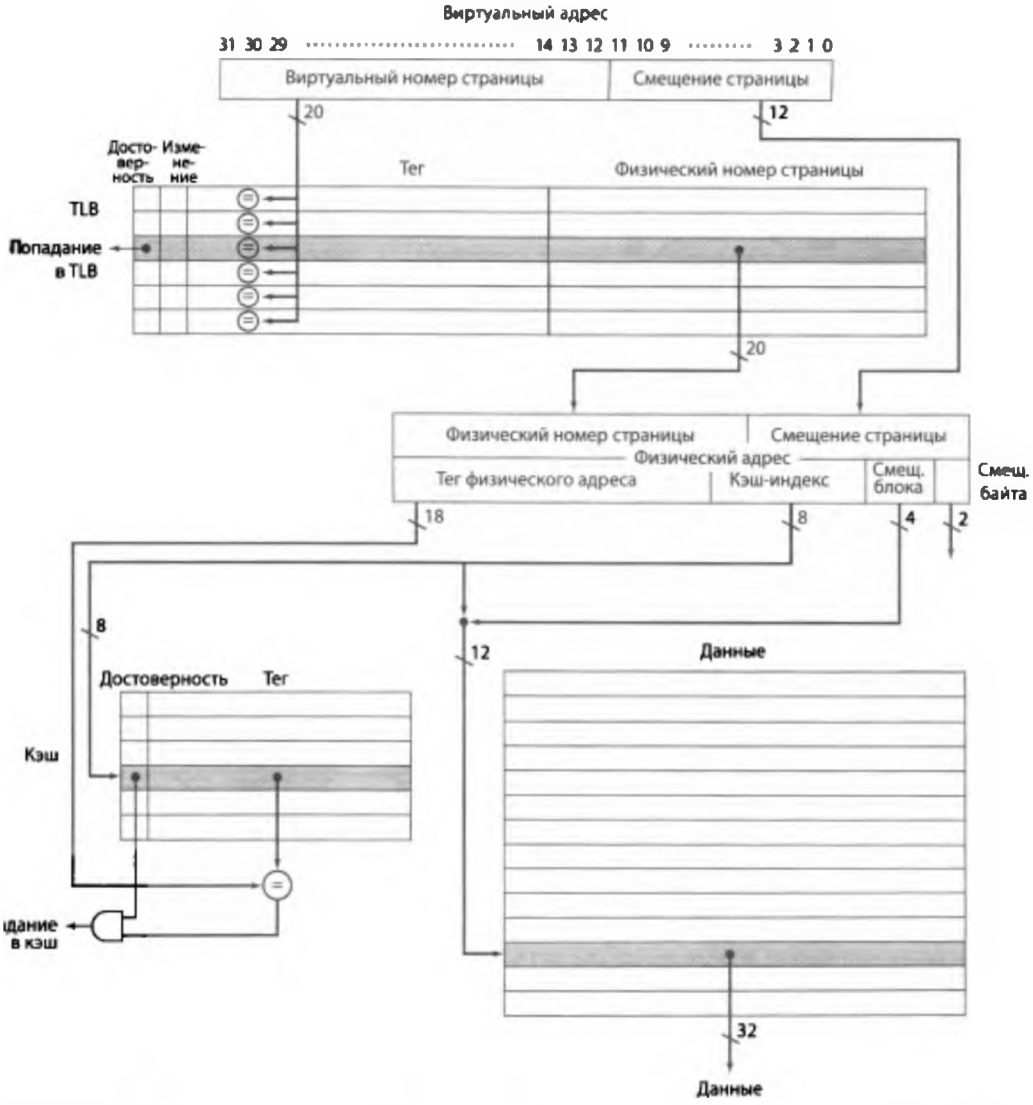


Рис. 5.21. TLB и кэш-память осуществляют процесс перехода от виртуального адреса к элементу данных в Intrinsity FastMATH. На этом рисунке показана организация TLB и кэш-памяти данных при размере страницы, равном 4 Кбайт. Эта схема показывает процесс чтения, а на рис. 5.22 дается описание обработки записи. Заметьте, что в отличие от показанного на рис. 5.9, RAM-блоки тега и данных разделены. За счет адресации длинной, но узкой RAM-памяти данных мы выбираем нужное слово в блоке с помощью индекса кэш-памяти, объединенного со смещением блока, не используя мультиплексор 16:1. В то время как кэш-память имеет непосредственное отображение, TLB обладает полной ассоциативностью. Реализация полностью ассоциативного буфера TLB требует, чтобы каждый тег TLB сравнивался с виртуальным номером страницы, поскольку нужная запись может быть в любом месте TLB. (См. уточнение, касающееся CAM-памяти в разделе 5.3, в подразделе «Расположение блока в кэш-памяти».) Если бит достоверности соответствующей записи установлен, обращение завершается попаданием в TLB, и биты и разряды из физического номера страницы вместе с разрядами из смещения страницы формируют индекс, который используется для доступа к кэш-памяти

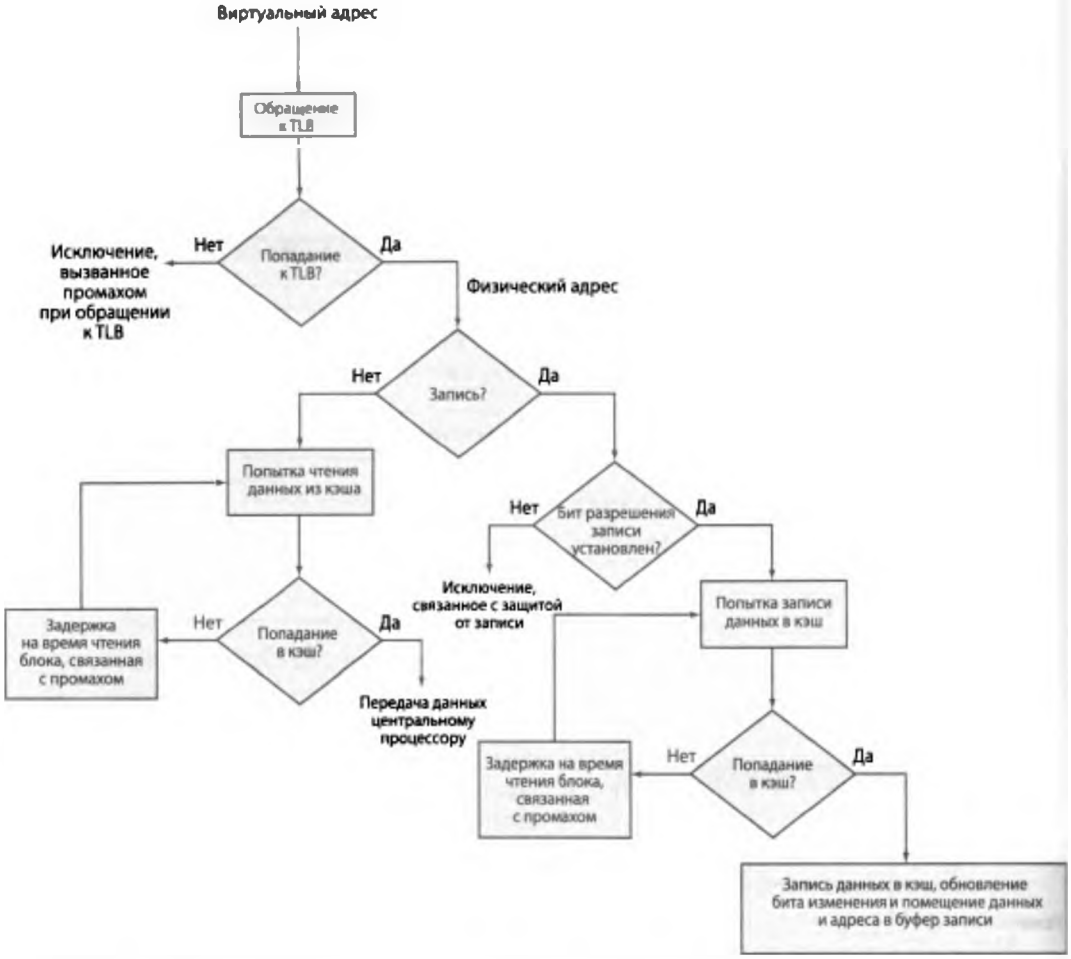


Рис. 5.22. Обработка чтения или сквозной записи в Intrinsity FastMATH TLB и в кэш-памяти. Если TLB генерирует попадание, к кэш-памяти можно обратиться с имеющимся физическим адресом. Для чтения кэш-память генерирует попадание или промах и предоставляет данные или принуждает к задержке на время доставки данных из памяти. Если осуществляется запись, часть записи в кэш-памяти при попадании переписывается, и данные отправляются в буфер записи, при условии, что ведется сквозная запись. Промах при записи равноценен промаху при чтении, за исключением того, что блок после чтения из памяти подвергается изменению. Технология отложенной записи требует установки для блока кэш-памяти бита изменения, а в буфер записи весь блок загружается только при промахе чтения или промахе записи, если заменяемый блок подвергался изменению. Учтите, что попадание при обращении к TLB и попадание при обращении к кэш-памяти — это независимые события, но попадание при обращении к кэш-памяти может произойти только после попадания при обращении к TLB, это означает, что данные должны быть в памяти. Взаимоотношения между промахами при обращении к TLB и промахами при обращении к кэш-памяти будут рассмотрены чуть позже в следующем упражнении и в упражнениях в конце главы

Таблица 5.4. Возможные комбинации событий в TLB, в системе виртуальной памяти и в кэш-памяти. Из представленных комбинаций три абсолютно нереальны, а одна возможна (попадание в TLB, попадание в виртуальную память, но промах при обращении к кэш-памяти), но никогда не возникает

TLB	Таблица страниц	Кэш	Возможно ли это? Если да, то при каких обстоятельствах?
Попадание	Попадание	Промах	Возможно, хотя обращение к таблице страниц на самом деле никогда не происходит, если произошло попадание при обращении к TLB
Промах	Попадание	Попадание	При обращении к TLB произошел промах, но запись нашлась в таблице страниц; после повторения попытки данные обнаруживаются в кэш-памяти
Промах	Попадание	Промах	При обращении к TLB произошел промах, но запись нашлась в таблице страниц; после повторения попытки обнаруживается, что данные в кэш-памяти отсутствуют
Промах	Промах	Промах	После промаха при обращении к TLB следует ошибка отсутствия страницы; после повторной попытки должен произойти промах при обращении за данными к кэш-памяти
Попадание	Промах	Промах	Это невозможно: в TLB не может быть результата преобразования, если страницы нет в памяти
Попадание	Промах	Попадание	Это невозможно: в TLB не может быть результата преобразования, если страницы нет в памяти
Промах	Промах	Попадание	Это невозможно: данные не могут находиться в кэш-памяти, если страницы нет в оперативной памяти

Уточнение. Таблица 5.4 составлена исходя из того, что все адреса памяти преобразованы в физические адреса до обращения к кэш-памяти. При такой организации кэш-память проиндексирована по физическим адресам и теги также созданы на основе физических адресов (и индекс кэш-памяти, и тег имеют физические, а не виртуальные адреса). В такой системе продолжительность обращения к памяти, при условии попадания при обращении к кэш-памяти, должна включать как время обращения к TLB, так и время обращения к кэш-памяти; разумеется, эти обращения могут быть подвергнуты конвейеризации.

В качестве альтернативного варианта процессор может индексировать кэш-память по адресам, которые полностью или частично являются виртуальными. Это называется **виртуально адресуемой кэш-памятью**, в которой используются теги, являющиеся виртуальными адресами, следовательно, такая кэш-память является *виртуально индексированной* и *использующей виртуальные теги*. В такой кэш-памяти оборудование преобразования адресов (TLB) при обычном обращении к кэш-памяти не используется, потому что обращение осуществляется с помощью виртуальных адресов, которые не нуждаются в преобразовании в физические адреса. TLB таким образом устраняется с критического пути,

Виртуально адресуемая кэш-память Кэш-память, обращение к которой осуществляется по виртуальному, а не по физическому адресу.

уменьшая латентность кэш-памяти. Но если происходит промах при обращении к кэш-памяти, процессору нужно преобразовать адрес в физический адрес для извлечения блока для кэш-памяти из оперативной памяти.

Когда обращение к кэш-памяти осуществляется по виртуальному адресу и страницы совместно используются программами (которые могут к ним обращаться по разным виртуальным адресам), может возникать **неоднозначность (aliasing)**. Она возникает в том случае, когда у одного и того же объекта имеется два имени, в данном случае — два виртуальных адреса для одной и той же страницы. Неоднозначность создает проблему, поскольку слово на такой странице может быть кэшировано в двух разных местах, каждое из которых соотносится с разными виртуальными адресами. Такая неоднозначность может позволить одной программе записать данные без уведомления другой программы о том, что данные были изменены. Полностью виртуально адресуемая кэш-память либо вводит конструктивные ограничения для кэш-памяти и TLB, чтобы сократить неоднозначность, либо требует от операционной системы и, возможно, от пользователя что-либо предпринимать, чтобы неоднозначность не возникала.

Распространенным компромиссом между этими двумя конструкциями является кэш-память, имеющая виртуальную индексацию, иногда с использованием только лишь части адреса, составляющей смещение страницы, которая на самом деле является физическим адресом, поскольку она не подвергается преобразованию, но использует физические теги. Такие конструкции *с виртуальной индексацией, но физической природой тегов* пытаются получить преимущества производительности виртуально индексированной кэш-памяти с преимуществами более простой архитектуры **физически адресуемой кэш-памяти**. Например, в этой кэш-памяти отсутствуют проблемы неоднозначности. На рис. 5.21 предполагается использование страницы размером 4 Кбайт, но на самом деле ее размер составляет 16 Кбайт, что позволяет процессору Intrinsity FastMATH использовать эту хитрость. Для ее осуществления нужно точно скоординировать минимальный размер страницы, размер кэш-памяти и степень ассоциативности.

Реализация защиты при использовании виртуальной памяти

Возможно, наиболее важной функцией виртуальной памяти является возможность совместного использования единой оперативной памяти сразу несколькими процессами, обеспечивающая защиту памяти в среде этих процессов и операционной системы. Механизм защиты должен обеспечить при совместном использовании оперативной памяти несколькими процессами, чтобы один отступивший от правил процесс не смог вести запись в адресное пространство другого пользовательского процесса или операционной системы, будь то преднамеренно или непреднамеренно. Бит разрешения записи в TLB может

Неоднозначность
Ситуация, при которой доступ к одному и тому же объекту осуществляется по двум адресам; может возникать в виртуальной памяти при наличии двух виртуальных адресов для одной и той же физической страницы.

Физически адресуемая кэш-память
Кэш-память, адресуемая по физическому адресу.

защитить страницу от записи. Без этого уровня защиты компьютерные вирусы могли бы получить еще большее распространение.

Интерфейс аппаратного и программного обеспечения

Чтобы позволить операционной системе реализовать защиту в системе виртуальной памяти, оборудование должно предоставить как минимум три перечисленные ниже основные возможности.

1. Поддержка по крайней мере двух режимов, показывающих, когда запущенный процесс является пользовательским процессом, или процессом операционной системы, который называется по-разному: процессом **супервизора**, процессом **ядра** или *исполнительным* процессом.
2. Предоставить часть состояния процессора, которую пользовательский процесс сможет прочитать, но не сможет вести в нее запись. Сюда включается бит режима пользователь-супервизор, который предписывает, в каком режиме находится процессор, в пользовательском или супервизорском, указатель на таблицу страниц и TLB. Для записи в эти элементы операционная система использует специальные инструкции, доступные только в режиме супервизора.
3. Предоставить механизмы, посредством которых процессор может переходить из пользовательского режима в режим супервизора и обратно. Переход в первом из этих направлений обычно осуществляется с помощью исключения **системного вызова**, реализуемого в виде специальной инструкции (в наборе инструкций MIPS это `syscall`), которая передает управление в специальное место в супервизорском кодовом пространстве. Как и при всех других исключениях, значение счетчика команд из точки системного вызова сохраняется в счетчике команд исключений (EPC), и процессор переводится в режим супервизора. Для возвращения из исключения в режим пользователя используется инструкция возвращения из исключения (`ERET`), которая переключает процессор в пользовательский режим и осуществляет переход по адресу, хранящемуся в EPC.

Благодаря использованию этого механизма и хранению таблиц страниц в адресном пространстве операционной системы, операционная система может вносить изменения в таблицы страниц, в то же время защищая их от изменений со стороны процесса пользователя, гарантируя, что этот процесс сможет получить доступ только к памяти, предоставленной ему операционной системой.

Режим супервизора
Также называется режимом ядра. Режим, свидетельствующий о том, что запущенный процесс является процессом операционной системы.

Системный вызов
Специальная инструкция, передающая управление из пользовательского режима в специальное место в супервизорском кодовом пространстве, инициируя механизм обработки исключения в процессе.

Нужно также не дать процессу возможность читать данные другого процесса. Например, нежелательно, чтобы студенческая программа могла читать оценки, находящиеся в памяти процессора. Как только начинается совместное использование памяти, нужно предоставить процессу возможность защиты его данных как

от чтения, так и от записи со стороны других процессов; в противном случае совместное использование оперативной памяти станет палкой о двух концах!

Вспомним, что у каждого процесса имеется свое собственное виртуальное адресное пространство. Таким образом, если операционная система поддерживает такую организацию таблиц страниц, при которой независимые виртуальные страницы отображаются на разобщенные физические страницы, один процесс не сможет получить доступ к данным других процессов. Разумеется, это также требует, чтобы пользовательский процесс не имел возможности изменять отображение таблицы страниц. Операционная система может гарантировать безопасность, если она не позволяет пользовательскому процессу изменять его собственные таблицы страниц. Но операционная система должна иметь возможность вносить изменения в таблицы страниц. Помещая таблицы страниц в защищенное адресное пространство операционной системы, мы выполняем оба требования.

Когда процесс хочет делиться информацией неким ограниченным способом, операционная система должна ему помочь, поскольку доступ к информации другого процесса требует изменения в таблице страниц обращающегося к этой информации процесса. Для ограничения совместного доступа только в рамках чтения может использоваться бит доступа к записи, и, подобно всему остальному содержимому таблицы страниц, этот бит может быть изменен только операционной системой. Чтобы разрешить другому процессу, скажем, P1, читать страницу, принадлежащую процессу P2, процесс P2 должен попросить операционную систему создать запись в таблице страниц для виртуальной страницы в адресном пространстве процесса P1, которая указывает на ту же физическую страницу, к которой хочет предоставить совместное использование процесс P2. Операционная система может использовать бит защиты от записи для предотвращения записи данных со стороны процесса P1, если это было требованием со стороны процесса P2. Любые биты, определяющие права доступа к странице, должны быть включены как в таблицу страниц, так и в TLB, поскольку обращение к таблице страниц осуществляется только при промахе обращения к TLB.

Уточнение. Когда операционная система решает вместо процесса P1 запустить процесс P2 (что называется **переключением контекста** или *переключением процесса*), она должна перекрыть доступ процесса P2 к таблицам страниц процесса P1, поскольку это поставит защиту под угрозу. Если TLB не используется, достаточно изменить регистр таблицы страниц, чтобы он указывал на таблицу страниц процесса P2 (а не на таблицу страниц процесса P1). При использовании TLB нужно очистить записи TLB, принадлежащие процессу P1, как для защиты данных процесса P1, так и для того, чтобы заставить TLB загрузить данные процесса P2. Если показатель переключения процессов слишком высок, все это может быть весьма неэффективным. Например, процесс P2 может загрузить только лишь несколько записей TLB перед тем, как операционная система снова переключится на процесс P1. К сожалению, процесс P1 затем обнаружит, что все его TLB-записи были стерты, и будет нести издержки промахов при обращении к TLB, снова загружая эти записи. Эта проблема возникает потому, что процессами P1 и P2 используются

Переключение контекста
Изменение внутреннего состояния процессора, позволяющее разным процессам использовать процессор, и включающее сохранение состояния с целью возвращения к текущему выполняемому процессу.

одинаковые виртуальные адреса, и мы должны очищать TLB во избежание путаницы, связанной с этими адресами.

Обычно альтернативой является расширение виртуального адресного пространства путем добавления *идентификатора процесса* или *идентификатора задачи*. Процессор Intrinsity FastMATH имеет для этой цели восьмиразрядное поле идентификатора адресного пространства (address space ID – ASID). Это небольшое поле идентифицирует текущий выполняемый процесс; оно сохраняется в регистре, загружаемом операционной системой при переключении процессов. Идентификатор процесса объединяется с тег-частью TLB, чтобы попадание при обращении к TLB происходило только в том случае, если совпадали и номер страницы, и идентификатор процесса. Эта комбинация избавляет от необходимости очищать TLB, за исключением весьма редких случаев.

Сходные проблемы могут возникать и для кэш-памяти, поскольку на момент переключения процесса кэш-память будет содержать данные выполняемого процесса. Эти проблемы возникают для физически адресуемой и виртуально адресуемой кэш-памяти по-разному и имеют множество различных решений, таких как идентификаторы процессов, используемых для того, чтобы гарантировать, что процесс получает свои собственные данные.

Обработка промахов при обращении к TLB и ошибок отсутствия страницы

Хотя преобразование виртуальных адресов в физические с помощью TLB, при условии попадания в нее, осуществляется довольно просто, обработка промахов при обращении к TLB и ошибок отсутствия страницы является куда более сложной задачей. Промах при обращении к TLB возникает в том случае, когда в TLB отсутствует запись, соответствующая виртуальному адресу. Промах при обращении к TLB может свидетельствовать об одной из двух возможностей:

1. Страница присутствует в памяти, и нужно лишь создать недостающую TLB-запись.
2. Страницы нет в памяти, и нужно передать управление операционной системе для обработки ошибки отсутствия страницы.

Как узнать, с каким из этих двух обстоятельств пришлось столкнуться? При обработке промаха при обращении к TLB будет вестись поиск записи в таблице страниц для доставки ее в TLB. Если у соответствующей записи таблицы страниц сброшен бит достоверности, значит, соответствующая страница не присутствует в памяти, и мы имеем дело с ошибкой отсутствия страницы, а не просто с промахом при обращении к TLB. Если бит достоверности установлен, мы можем просто извлечь нужную запись.

Промах при обращении к TLB может быть обработан программно или аппаратно, поскольку для этого потребуется выполнить всего лишь небольшую последовательность операций, копирующих достоверную запись таблицы страниц из памяти в TLB. MIPS традиционно ведет обработку промахов при обращении

к TLB программным способом. Запись таблицы страниц берется из памяти, после чего инструкция, вызвавшая промах при обращении к TLB, выполняется еще раз. При повторном выполнении произойдет попадание при обращении к TLB. Если запись в таблице страниц показывает, что страницы в памяти нет, то на этот раз будет получено исключение, связанное с ошибкой отсутствия страницы.

Обработка промаха при обращении к TLB или ошибка отсутствия страницы требует использования механизма исключений для прерывания активного процесса, передачи управления операционной системе с последующим возобновлением выполнения прерванного процесса. Ошибка отсутствия страницы будет обнаружена в какой-нибудь из тактовых циклов, используемых для обращения к памяти. Для перезапуска инструкции после обработки ошибки отсутствия страницы должно быть сохранено значение счетчика команд для той инструкции, которая вызвала ошибку отсутствия страницы. Как и в главе 4, для хранения этого значения используется счетчик команд исключения (EPC).

Кроме того, промах при обращении к TLB или исключение, вызванное ошибкой отсутствия страницы, должны быть подтверждены к концу того же тактового цикла, в котором происходило обращение к памяти, чтобы в следующий тактовый цикл началась обработка исключения, а не продолжилось нормальное выполнение инструкции. Если ошибка отсутствия страницы в этот тактовом цикле распознана не была, инструкция загрузки может переписать значение регистра, что обернется провалом при попытке перезапуска инструкции. Например, рассмотрим инструкцию `lw $1,0($1)`: компьютер должен быть способен предотвратить наступление стадии конвейера, на которой осуществляется запись; в противном случае он не сможет должным образом перезапустить инструкцию, поскольку содержимое регистра `$1` будет уничтожено. Такие же сложности возникнут и при выполнении инструкций сохранения. При возникновении ошибки отсутствия страницы реальное завершение записи в память нужно предотвратить; обычно это делается путем снятия сигнала на линии управления записью в память.

Таблица 5.5. Регистры управления MIPS. Предполагается, что они находятся в сопроцессоре 0, и, следовательно, для их чтения используется инструкция mfc0, а для их записи используется инструкция mtc0

Регистр	Номер регистра сопроцессора 0	Описание
EPC	14	Откуда будет осуществлен перезапуск после исключения
Cause	13	Причина исключения
BadVAddr	8	Адрес инструкции, вызвавшей исключение
Index	0	Считываемое или записываемое место в TLB
Random	1	Псевдослучайное место в TLB
EntryLo	2	Физический адрес страницы и флаги
EntryHi	10	Виртуальный адрес страницы
Context	4	Адрес таблицы страниц и номер страницы

Интерфейс аппаратного и программного обеспечения

В промежуток времени между началом работы обработчика исключения в операционной системе и моментом, когда операционная система сохранит все состояние процесса, операционная система особенно уязвима. Например, если в момент обработки в операционной системе первого исключения случится какое-нибудь другое исключение, блок управления перепишет значение счетчика команд исключения, не позволяя вернуться к той инструкции, которая вызвала ошибку отсутствия страницы! Нужно предотвратить эту аварийную ситуацию путем запрещения и **разрешения исключений**. При возникновении первого исключения процессор устанавливает бит, запрещающий все другие исключения; это может произойти одновременно с установкой процессором бита режима супервизора. Затем операционная система сохранит достаточную долю информации о состоянии, которая позволит ей все восстановить, если случится еще одно исключение, а именно, значение регистров счетчика команд исключения (EPC) и причины исключения (Cause). EPC и Cause являются двумя специальными регистрами управления, помогающими обрабатывать исключения, промахи при обращении к TLB и ошибки отсутствия страницы; остальные регистры показаны в табл. 5.5. Затем операционная система может вновь разрешить исключения. Эти шаги гарантируют то, что исключения не заставят процессор утратить какое-нибудь состояние и из-за этого потерять возможность возобновления выполнения прерванной инструкции.

Как только операционной системе станет известен виртуальный адрес, вызвавший ошибку отсутствия страницы, она должна выполнить следующие три шага:

1. Отыскать запись в таблице страниц, используя виртуальный адрес, и найти место на диске той страницы, к которой было обращение.
2. Выбрать заменяемую физическую страницу; если выбранная страница подвергалась изменениям, она должна быть записана обратно на диск перед тем, как в эту физическую страницу будет помещена новая виртуальная страница.
3. Запустить чтение, чтобы доставить страницу, к которой было обращение, с диска в выбранную физическую страницу.

Разумеется, этот последний шаг займет миллионы тактовых циклов процессора (и это время удвоится, если заменяемая страница подвергалась изменениям); соответственно, операционная система выберет, как правило, другой процесс для выполнения процессором, пока не завершится обращение к диску. Поскольку операционная система сохранила состояние процесса, она может свободно передать управление процессором другому процессу.

Когда чтение страницы с диска завершится, операционная система может восстановить состояние процесса, который изначально вызвал

Разрешение исключения
Также называется разрешением прерывания. Сигнал или действие, управляющее реакцией процессора на исключение. Необходим для предотвращения возникновения исключений в течение интервалов времени, предшествующих гарантированному сохранению процессором состояния, необходимого для возобновления выполнения инструкции.

ошибку отсутствия страницы, и выполнить инструкцию, возвращающую из исключения. Эта инструкция переключит процессор из режима ядра в режим пользователя, а также восстановит значение счетчика команд. Затем пользовательский процесс перезапустит инструкцию, потерпевшую неудачу, успешно обратится к странице и продолжит свое выполнение.

Исключения, связанные с отсутствием страницы при обращении к данным, довольно трудно правильно реализовать в процессоре из-за сочетания трех характеристик:

1. Они происходят в середине инструкции, в отличие от отсутствия страницы при обращении к инструкции.
2. Инструкция не может быть завершена до обработки исключения.
3. После обработки исключения инструкция должна быть перезапущена, как будто ничего не случилось.

Превращение инструкций в **перезапускаемые** с целью обработки исключения и последующего продолжения выполнения инструкции в архитектурах наподобие MIPS дается относительно легко. Поскольку каждая инструкция записывает только один элемент данных и эта запись происходит в конце цикла инструкции, мы можем просто предотвратить завершение инструкции (отменив запись) и перезапустить инструкцию с самого начала.

Давайте более пристально взглянем на MIPS. Когда происходит промах при обращении к TLB, оборудование MIPS сохраняет номер страницы, к которой было обращение, в специальном регистре, который называется `BadVAddr`, и генерирует исключение.

Исключение вызывает операционную систему, которая обрабатывает исключение программным способом. Управление передается на адрес $8000\ 0000_{16}$, в то место, где находится **обработчик** промахов при обращении к TLB. Чтобы определить физический адрес страницы, обращение к которой вызвало промах, подпрограмма обработки промаха при обращении к TLB индексирует таблицу страниц, используя номер страницы виртуального адреса и регистр таблицы страниц, который показывает начальный адрес таблицы страниц активного процесса. Для ускорения этой индексации оборудование MIPS помещает все необходимое в специальный регистр `Context`: старшие 12 разрядов содержат адрес базы таблицы страниц, а следующие 18 разрядов содержат виртуальный адрес ненайденной страницы. Каждая запись таблицы страниц является одним словом, поэтому последние два разряда содержат нули. Таким образом, первые две инструкции копируют содержимое регистра `Context` во временный регистр ядра `$k1`, а затем загружают запись таблицы страниц из этого адреса в регистр `$k1`. Следует напомнить, что регистры `$k0` и `$k1` зарезервированы для операционной системы для использования без сохранения; основной причиной такого соглашения является ускорение работы обработчика промахов при обращении к TLB. Ниже показан MIPS-код для типового обработчика промахов при обращении к TLB:

Перезапускаемая инструкция
Инструкция, которая может возобновить выполнение после обработки исключения без влияния этого исключения на результаты выполнения инструкции.

Обработчик
Название подпрограммы, вызываемой для обработки исключения или прерывания.

```
TLBmiss:
 mfc0   $k1,Context     # копирование адреса элемента таблицы страниц (PTE)
                        # во временный регистр $k1
 lw     $k1, 0($k1)     # помещение PTE во временный регистр $k1
 mtc0   $k1,EntryLo     # помещение PTE в специальный регистр EntryLo
 tlbwr                  # помещение EntryLo в произвольную запись TLB
 eret                   # возвращение из исключения, вызванного промахом при
                        # обращении к TLB
```

Как показано выше, в MIPS имеется специальный набор системных инструкций для обновления TLB. Инструкция `tlbwr` копирует значение из регистра управления `EntryLo` в TLB-запись, выбранную регистром управления `Random`. Этот последний регистр позволяет реализовать произвольную замену, поэтому по сути это автономный счетчик. Обработка промаха при обращении к TLB занимает около десятка тактовых циклов.

Заметьте, что обработчик промаха при обращении к TLB не проверяет, установлен ли в записи таблицы страниц бит достоверности. Поскольку исключение, связанное с промахом обращения к TLB, случается намного чаще, чем ошибка отсутствия страницы, операционная система загружает TLB из таблицы страниц без проверки записи и перезапускает инструкцию. Если запись недостоверна, возникает еще одно, другое исключение, и операционная система обнаруживает ошибку отсутствия страницы. Этот метод ускоряет обработку часто происходящих промахов при обращении к TLB при незначительной издержке производительности для нечастых случаев отсутствия страницы.

Как только процесс, генерирующий ошибку отсутствия страницы, будет прерван, управление будет передано на адрес $8000\ 0180_{16}$, отличающийся от адреса обработчика промахов при обращении к TLB. Это общий адрес для исключений; у обработчика промахов при обращении к TLB имеется специальная точка входа для сокращения издержки при промахе. Операционная система использует регистр причины исключения `Cause` для определения причины исключения. Поскольку исключение связано с ошибкой отсутствия страницы, операционная система знает, что потребуется пространная обработка. Таким образом, в отличие от промаха при обращении к TLB, она сохраняет все состояние активного процесса. Это состояние включает все регистры общего назначения и регистры с плавающей точкой, адрес таблицы страниц, EPC, и регистр причины исключения `Cause`. Поскольку обработчики исключений обычно не используют регистры с плавающей точкой, при передаче управления на общую точку входа их значения не сохраняются, оставляя это занятие лишь для некоторых обработчиков, которым это нужно.

В табл. 5.6 приведен фрагмент MIPS-кода обработчика исключений. Обратите внимание на то, что в нем сохраняется и восстанавливается состояние, обращается внимание на разрешение и запрещение исключений, но для обработки конкретного исключения вызывается код на языке Си.

Виртуальный адрес, ставший причиной ошибки, зависит от того, была ли это ошибка при обращении к инструкции или при обращении к данным. Адрес инструкции, которая сгенерировала ошибку, находится в EPC. Если ошибка произошла при обращении к странице инструкций, EPC содержит виртуальный адрес страницы, вызвавшей ошибку; в противном случае виртуальный адрес, вызвавший

ошибку, должен быть вычислен путем изучения инструкции (чей адрес находится в EPC), чтобы определить базовый регистр и поле смещения.

Таблица 5.6. MIPS-код для сохранения и восстановления состояния при возникновении исключений

Сохранение состояния			
Сохранение регистров общего назначения (GPR)	addi	$k1,$sp, -XCPSIZE	# сохранение пространства стека
			# для состояния
	sw	$sp, XCT_SP($k1)	# сохранение в стеке $sp
	sw	$v0, XCT_V0($k1)	# сохранение в стеке $v0
	...		# сохранение в стеке $v1, $a1,
			# $s1, $t1,...
	sw	$ra, XCT_RA($k1)	# сохранение в стеке $ra
Сохранение hi, lo	mfhi	$v0	# копирование Hi
	mflo	$v1	# копирование Lo
	sw	$v0, XCT_HI($k1)	# сохранение в стеке значения Hi
	sw	$v1, XCT_LO($k1)	# сохранение в стеке значения Lo
Сохранение регистров исключения	mfc0	$a0, $cr	# копирование регистра причины
	sw	$a0, XCT_CR($k1)	# сохранение в стеке значения $cr
	....		# сохранение $v1.....
	mfc0	$a3, $sr	# копирование регистра статуса
	sw	$a3, XCT_SR($k1)	# сохранение в стеке $sr
Установка sp	move	$sp, $k1	# sp = sp - XCPSIZE
Разрешение вложенных исключений			
	andi	$v0, $a3, MASK1	# $v0 = $sr & MASK1,
			# разрешение исключений
	mtc0	$v0, $sr	# $sr = значению, разрешающему
			# исключения
Вызов обработчика исключения, написанного на языке C			
установка $gp	move	$gp, GPINIT	# установка $gp на точку области кучи
Вызов C-кода	move	$a0, $sp	# arg1 = указатель на стек исключений
	jal	xcpt_deliver	# вызов Си-кода для обработки
			# исключения
Восстановление состояния			
Восстановление большинства GPR, hi, lo	move	$at, $sp	# временное значение $sp
	lw	$ra, XCT_RA($at)	# восстановление $ra из стека
	....		# восстановление $t0..... $a1
	lw	$a0, XCT_A0($k1)	# восстановление $a0 из стека
Восстановление регистра статуса	lw	$v0, XCT_SR($at)	# загрузка старого $sr из стека
	li	$v1, MASK2	# установка маски для запрещения
			# исключений
	and	$v0, $v0, $v1	# $v0 = $sr & MASK2, запрещение
			# исключений
	mtc0	$v0, $sr	# установка регистра статуса
Возвращение из обработки исключения			
Восстановление $sp и остальных GPR, которые использовались в качестве временных регистров	lw	$sp, XCT_SP($at)	# восстановление $sp из стека
	lw	$v0, XCT_V0($at)	# восстановление $v0 из стека
	lw	$v1, XCT_V1($at)	# восстановление $v1 из стека
	lw	$k1, XCT_EPC($at)	# копирование старого $epc из стека
	lw	$at, XCT_AT($at)	# восстановление $at из стека
Восстановление ERC и возвращение	mtc0	$k1, $epc	# восстановление $epc
	eret	$ra	# возвращение к прерванной инструкции

Уточнение. Эта упрощенная версия предполагает, что в указателе стека (sp) находится достоверное значение. Чтобы избежать ошибки отсутствия страницы при выполнении этого низкоуровневого кода обработки исключений, MIPS выделяет часть своего адресного пространства, которая не может иметь ошибок отсутствия страниц, эта часть называется **неотображаемой**. Операционная система помещает код точки входа в обработчик исключений и стек обработки исключений в неотображаемую память. Оборудование MIPS переводит виртуальные адреса с $8000\ 0000_{16}$ по $BFFF\ FFFF_{16}$ в физические адреса путем простого игнорирования старших разрядов виртуального адреса, помещая тем самым эти адреса в нижнюю часть физической памяти. Таким образом, операционная система помещает код входа в обработчик исключений и стеки обработки исключений в неотображаемую память.

Уточнение. Код в табл. 5.6 показывает последовательность возвращения из исключения, имеющуюся в MIPS-32. Более старая архитектура MIPS-I использует вместо инструкции eret инструкции rfe и jr.

Уточнение. Для процессоров с более сложными инструкциями, которые могут касаться многих мест в памяти и записывать множество элементов данных, превращение инструкций в перезапускаемые дается значительно труднее. Обработка одной инструкции может сгенерировать в середине инструкции сразу несколько ошибок отсутствия страницы. Например, в процессоре x86 есть инструкция перемещения блока, затрагивающая тысячи слов данных. В таких процессорах инструкции часто не могут быть перезапущены с самого начала, как это делалось с инструкциями MIPS. Вместо этого инструкция должна быть прервана и чуть позже продолжена по ходу своего выполнения. Возобновление выполнения инструкции с середины обычно требует сохранения некоторого специального состояния, обработки исключения и восстановления этого специального состояния. Правильное выполнение этой задачи требует координации между кодом обработки исключения в операционной системе и оборудованием.

Краткие выводы

Виртуальная память — это название уровня в иерархии памяти, который управляет кэшированием между оперативной памятью и диском. Виртуальная память позволяет отдельной программе расширить ее адресное пространство, выходя за пределы оперативной памяти. Наиболее важно то, что виртуальная память поддерживает совместное использование оперативной памяти сразу несколькими одновременно активными процессами с соблюдением защитных мер.

Управление иерархией памяти между оперативной памятью и диском дается нелегко, поскольку цена ошибки отсутствия страницы слишком высока. Для сокращения коэффициента промахов используется ряд технологий:

1. Страницы делают достаточно большими, чтобы воспользоваться преимуществом пространственной локальности и уменьшить коэффициент промахов.
2. Отображение виртуальных адресов на физические, реализуемое с помощью таблицы страниц, делается полностью ассоциативным, чтобы виртуальная страница могла быть помещена в любое место оперативной памяти.

Неотображаемая память
Часть адресного пространства, в которой не может быть ошибок отсутствия страницы.

3. Операционная система использует такие технологии, как LRU и бит обращения, для выбора страницы, подлежащей замене.

Запись на диск обходится дорого, поэтому виртуальная память использует схему с отложенной записью, а также отслеживает, подверглась ли страница изменениям (используя бит изменения), чтобы избежать записи неизменившихся страниц обратно на диск.

Механизм виртуальной памяти предоставляет предобразование адреса из виртуального, используемого программой, в физическое адресное пространство, используемое для обращения к памяти. Преобразование адреса позволяет совместно использовать оперативную память в защищенном режиме и предоставляет ряд дополнительных преимуществ, таких как упрощение выделения памяти. Гарантия защищенности процессов друг от друга требует, чтобы только операционная система могла изменять преобразование адресов, что реализуется за счет предотвращения изменения таблиц страниц со стороны пользовательских программ. Управляемое совместное использование страниц сразу несколькими процессами может быть реализовано с помощью операционной системы и битов доступа в таблице страниц, показывающих, имеет ли пользовательская программа доступ к странице по чтению или записи.

Если процессор должен обращаться к таблице страниц, находящейся в памяти для преобразования каждого адреса, виртуальная память будет обходиться слишком дорого, и кэширование станет бессмысленным! Вместо этого в качестве кэш-памяти для преобразований из таблицы страниц служит буфер TLB. В таком случае адреса транслируются из виртуальных в физические с использованием преобразований, имеющихся в TLB.

Все устройства кэш-памяти, виртуальная память и буферы TLB основаны на общем наборе принципов и линий поведения. Эта общая среда будет рассмотрена в следующем разделе.

Представление о производительности программ

Хотя виртуальная память была изобретена, чтобы позволить небольшому объему памяти работать под видом большого, разница в производительности между диском и памятью означает, что если программа регулярно обращается к объему виртуальной памяти, превышающему объем физической памяти, она будет работать очень медленно. Такая программа будет постоянно заниматься переброской, или свопингом, страниц между памятью и диском, то есть войдет в режим так называемой *пробуксовки*. Возникновение пробуксовки оборачивается аварией, но она случается довольно редко. Если ваша программа начинает пробуксовывать, самым простым решением будет ее запуск на компьютере с более объемной памятью или покупка дополнительных блоков памяти для вашего компьютера. Более сложное решение заключается в пересмотре алгоритма и структуры данных на предмет возможностей изменения локальности и сокращения за счет этого количества одновременно используемых вашей программой страниц. Этот набор наиболее востребованных страниц неформально называется *рабочим набором*.

Более распространенной проблемой производительности являются промахи при обращении к TLB. Поскольку TLB может одновременно справляться только с 32–64 записями, касающимися страниц, программа легко может столкнуться с высоким коэффициентом промахов при обращении к TLB, поскольку процессор может получить непосредственный доступ менее чем к четверти мегабайта: 64 × 4 Кбайт = 0,25 Мбайт. Например, промахи при обращении к TLB часто становятся проблемой для задач сортировки Radix Sort. Пытаясь снизить остроту этой проблемы, большинство компьютерных архитектур сегодня поддерживают переменный размер страниц. Например, вдобавок к стандартным страницам в 4 Кбайт оборудование MIPS поддерживает страницы длиной 16 Кбайт, 64 Кбайт, 256 Кбайт, 1 Мбайт, 4 Мбайт, 16 Мбайт, 64 Мбайт и 256 Мбайт. Следовательно, если программа использует большие размеры страниц, она может напрямую обращаться к большему пространству памяти без промахов при обращении к TLB.

Трудности, с которыми приходится сталкиваться на практике, заставляют операционную систему позволить программам выбрать эти, более крупные размеры страниц. Здесь также более сложным решением по сокращению промахов при обращении к TLB будет пересмотр алгоритма и структуры данных для сокращения рабочего набора страниц; учитывая важность влияния обращений к памяти на производительность и частоту промахов при обращении к TLB, некоторые программы с большими рабочими наборами для достижения этой цели подверглись переработке.

Самопроверка

Соотнесите элемент иерархии, показанный слева, с наиболее подходящей для него фразой, показанной справа:

1. Кэш уровня L1	а. Кэш-память для кэш-памяти
2. Кэш уровня L2	б. Кэш-память для дисков
3. Оперативная память	в. Кэш-память для оперативной памяти
4. TLB	г. Кэш-память для элементов таблицы страниц

5.5. Общая среда для иерархий памяти

На данный момент нам известно, что у разных типов элементов иерархии памяти имеется много общего. Хотя многие из аспектов элементов иерархии памяти отличаются количественно, многое в стратегии и свойствах, определяющих их функционирование, имеет качественные сходства. В табл. 5.7 показано, как могут отличаться друг от друга некоторые количественные характеристики элементов иерархии памяти. Далее в этом разделе будут рассмотрены общие действующие альтернативы для элементов иерархии памяти и то, как ими определяется поведение этих элементов. Их поведение будет исследовано с помощью серии из четырех вопросов, которые возникают между любыми двумя уровнями иерархии памяти, хотя для упрощения нашей задачи будет преимущественно использоваться терминология, применимая к кэш-памяти.

Таблица 5.7. Основные количественные конструктивные параметры, характеризующие основные элементы иерархии памяти, имеющиеся в компьютере. Это типичные значения для представленных уровней по состоянию на 2008 год. Хотя диапазон значений довольно широк, частично это обусловлено тем, что многие значения, сместившиеся со временем, связаны друг с другом, например по мере того, как увеличивался размер кэш-памяти для преодоления больших издержек промахов, также возрастал и размер блоков

Свойство	Типичные значения для устройств кэш-памяти L1	Типичные значения для устройств кэш-памяти L2	Типичные значения для памяти со страничной организацией	Типичные значения для TLB
Общий размер в блоках	250–2000	15 000–50 000	16 000–250 000	40–1024
Общий размер в килобайтах	16–64	500–4000	1 000 000–1 000 000 000	0,25–16
Размер блока в байтах	16–64	64–128	4000–64 000	4–32
Издержки промаха в тактовых циклах	10–25	100–1000	10 000 000–100 000 000	10–1000
Коэффициент промахов (общий для L2)	2%–5%	0,1%–2%	0,00001%–0,0001%	0,01%–2%

Вопрос 1: Куда может быть помещен блок?

Мы уже видели, что для размещения блока в верхнем уровне иерархии может использоваться ряд схем, от непосредственного отображения до множественной и полной ассоциативности. Как уже ранее упоминалось, весь этот ряд схем может рассматриваться как вариации множественно-ассоциативной схемы, где варьируется количество каналов и количество блоков на канал:

Название схемы	Количество каналов	Блоков на канал
Непосредственное отображение	Равно количеству блоков в кэш-памяти	1
Множественная ассоциативность	Равно количеству блоков в кэш-памяти, поделенному на степень ассоциативности	Равно степени ассоциативности (обычно 2–16)
Полная ассоциативность	1	Равно количеству блоков в кэш-памяти

Преимуществом увеличения степени ассоциативности является то, что при этом обычно снижается коэффициент промахов. Уменьшение коэффициента промахов получается за счет снижения промахов при конкуренции за одно и то же место. Вскоре мы рассмотрим данный вопрос более подробно. Сначала давайте посмотрим, насколько существенным будет полученное преимущество. На рис. 5.23

казаны коэффициенты промахов для блоков кэш-памяти нескольких размеров по мере изменения степени ассоциативности от непосредственного отображения до восьмиканальной ассоциативности. Наибольший прирост получен при переходе от непосредственного отображения к двухканальной ассоциативности, что привело к снижению коэффициента промахов между 20% и 30%. По мере роста размера кэш-памяти относительное улучшение от повышения степени ассоциативности растет незначительно. Поскольку общий коэффициент промахов в более крупной кэш-памяти будет ниже, возможности улучшения коэффициента промахов уменьшаются, и абсолютное улучшение коэффициента промахов от повышения степени ассоциативности существенно сужается. Потенциальным недостатком повышения степени ассоциативности, как уже ранее упоминалось, является повышение стоимости и увеличение времени доступа.

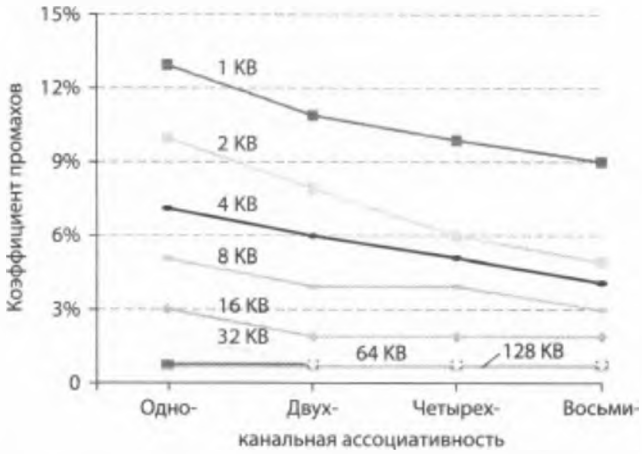


Рис. 5.23. Коэффициенты промахов при обращении к кэш-памяти данных для каждого из восьми размеров кэш-памяти по мере возрастания степени ассоциативности. При существенном преимуществе от перехода с одноканальной (с непосредственным отображением) к двухканальной ассоциативной кэш-памяти преимущества от дальнейшего повышения степени ассоциативности менее значительны (например, 1%–10%-ное улучшение при переходе от двухканальной к четырехканальной ассоциативной кэш-памяти по сравнению с 20%–30%-ным улучшением при переходе от одного к двум каналам). Еще меньшее улучшение происходит при переходе от четырехканальной к восьмиканальной ассоциативности, которая, в свою очередь, достигает практически сопоставимого с полной ассоциативностью коэффициента промахов. Меньшие по объему блоки кэш-памяти получают намного более значительные абсолютные преимущества от повышения степени ассоциативности, поскольку базовый показатель коэффициента промахов для небольших блоков кэш-памяти выше. Как собраны эти данные, объяснено в табл. 5.3

Вопрос 2: Как определяется местоположение блока?

Способ обнаружения блока зависит от схемы размещения блоков, поскольку этим диктуется количество возможных мест его нахождения. Эти схемы можно свести в следующую таблицу:

Ассоциативность	Метод обнаружения	Требуемое количество сравнений
Непосредственное отображение	Индексирование	1
Множественная	Индексирование наборов, поиск среди элементов набора	Равно степени ассоциативности
Полная	Просмотр всех элементов кэш-памяти	Равно размеру кэш-памяти
	Отдельная таблица поиска	0

Выбор между непосредственным отображением, множественной и полной ассоциативностью в любой иерархии памяти будет зависеть от цены промаха по сравнению с ценой реализации ассоциативности, как во времени, так и в необходимости использования дополнительного оборудования. Наличие на микросхеме кэш-памяти L2 позволяет использовать более высокую степень ассоциативности, поскольку время попадания становится не таким критичным, и разработчикам не нужно зависеть от стандартных модулей SRAM в качестве строительных элементов. Полностью ассоциативные устройства кэш-памяти обходятся слишком дорого, за исключением устройств кэш-памяти небольших размеров, где стоимость компараторов не слишком большая и где абсолютное снижение коэффициент промахов имеет наивысшее значение.

В системах виртуальной памяти отдельная страница отображения — таблица страниц — ведется для индексирования памяти. Кроме запоминающего устройства, необходимого для таблицы, использование индексной таблицы требует дополнительных обращений к памяти. Вариант полной ассоциативности для размещения страниц и наличие дополнительной таблицы мотивируются следующими фактами:

1. Полная ассоциативность дает преимущества, поскольку промахи обходятся слишком дорого.
2. Полная ассоциативность позволяет программному обеспечению использовать довольно сложные схемы замены, разрабатываемые для уменьшения коэффициента промахов.
3. Полное отображение может быть легко проиндексировано без использования дополнительного оборудования, поиск при этом не потребуется.

Именно поэтому системы виртуальной памяти почти всегда используют полностью ассоциативное размещение.

Размещение с множественной ассоциативностью часто используется для устройств кэш-памяти и буферов TLB, где при обращении сочетаются индексирование и поиск в небольших наборах данных. Устройства кэш-памяти с непосредственным отображением используются только в небольшом количестве систем из-за их преимуществ во времени доступа и в простоте. Преимущества во времени обращения возникают из-за того, что поиск нужного блока не зависит от сравнения. Выбор такой конструкции зависит от множества деталей реализации, например от того, находится ли устройство кэш-памяти на самом кристалле микропроцессора, от того, какая технология используется для реализации кэш-памяти, и от критичности времени доступа при определении продолжительности тактового цикла процессора.

Вопрос 3: Какой блок должен быть заменен при промахе обращения к кэш-памяти?

Когда промах происходит в ассоциативной кэш-памяти, нужно решить, какой блок заменить. В полностью ассоциативной кэш-памяти кандидатами на замену являются все блоки. Если кэш-память является множественно-ассоциативной, нужно выбрать блок из соответствующего набора. Разумеется, замену легко произвести в кэш-памяти с непосредственным отображением, потому что для нее есть только один кандидат.

Для множественно-ассоциативной и полностью ассоциативной кэш-памяти есть две основные стратегии замены:

- *Произвольная*: блоки-кандидаты выбираются произвольным образом, возможно использование помощи со стороны оборудования. Например, в MIPS поддерживается произвольная замена для промахов при обращениях к TLB.
- *Замена наименее востребованного элемента (LRU)*: заменяется тот блок, который не был востребован длительный период времени.

На практике, LRU — слишком дорогая для реализации стратегия для тех элементов иерархии, степень ассоциативности которых ниже определенного уровня (обычно от двух до четырех), поскольку отслеживание востребованности информации обходится довольно дорого. Даже для четырехканальной ассоциативности, LRU зачастую имеет приблизительный характер, например отслеживается, какая из двух пар блоков наименее востребована (для чего нужен один бит), а затем отслеживается, какой из блоков в каждой паре наименее востребован (для чего требуется по одному биту на пару).

При более высокой степени ассоциативности используется либо приблизительная стратегия LRU, либо произвольная замена. В кэш-памяти алгоритм замены выполняется оборудованием, из чего следует, что схема должна быть проста в реализации. Произвольная замена довольно легко создается в оборудовании, и для двухканальной ассоциативной кэш-памяти произвольная замена имеет коэффициент промахов примерно в 1,1 раза выше, чем LRU-замена. Чем больше становится кэш-память, тем меньше становится коэффициент промахов для обеих стратегий замены, и абсолютная разница становится незначительной. Фактически, произвольная замена может быть иногда лучше простой приблизительной LRU-замены, которую легче реализовать в оборудовании.

В виртуальной памяти некоторые формы LRU-замены всегда приблизительны, поскольку даже незначительное уменьшение коэффициента промахов может играть важную роль, когда цена промаха слишком велика.

Чтобы упростить операционной системе задачу отслеживания наименее востребованных страниц, часто предоставляются биты обращения или эквивалентные им функциональные возможности. Из-за высокой стоимости промахов и их относительной редкости вполне приемлема преимущественно программная аппроксимация этой информации.

Вопрос 4: Что происходит при записи?

Ключевой характеристикой любого элемента иерархии памяти является то, как он справляется с записью. До сих пор рассматривались два основных варианта:

- *Сквозная запись*: информация записывается как в блок кэш-памяти, так и в блок, расположенный в иерархии памяти уровнем ниже (для кэш-памяти — в оперативную память). Эта схема используется в кэш-памяти, показанной на рис. 5.2.
- *Отложенная запись*: информация записывается только в блок кэш-памяти. Измененный блок записывается в нижний уровень иерархии только при его замене. В системах виртуальной памяти всегда используется отложенная запись в силу причин, рассмотренных в разделе 5.4.

У каждой из стратегий записи, сквозной и отложенной, есть свои преимущества. Основные преимущества отложенной записи заключаются в следующем:

- Отдельные слова могут быть записаны процессором с той скоростью, с которой происходит обращение к ним в кэш-памяти, а не в оперативной памяти.
- Несколько записей внутри блока требуют только одной операции записи на уровне иерархии, расположенном на ступень ниже.
- Когда происходит отложенная запись блока, система может эффективно использовать широкополосную передачу данных, поскольку ведется запись сразу всего блока.

Основные преимущества сквозной записи заключаются в следующем:

- Промахи обходятся дешевле, поскольку они никогда не требуют обратной записи блока на уровень иерархии, расположенный на ступень ниже.
- Сквозная запись реализуется проще, чем отложенная, хотя для практичности кэш-память со сквозной записью все же будет нуждаться в использовании буфера записи.

В системах виртуальной памяти целесообразно использовать исключительно стратегию отложенной записи из-за высокой латентности записи на низший уровень иерархии (на диск). Частота генерации операций записи процессором обычно превосходит частоту, с которой система памяти может обрабатывать эти операции записи, даже с учетом физически и логически более широких блоков памяти и пакетных режимов передачи для RAM. По этой причине в современной кэш-памяти самого низкого уровня, как правило, используется отложенная запись.

Общее представление

Разные версии кэш-памяти, буферы TLB и виртуальная память могут изначально выглядеть очень непохожими друг на друга, но они основаны на одних и тех же двух принципах локальности, и с ними можно разобраться, отвечая на следующие четыре вопроса:

Вопрос 1: Куда может быть помещен блок?

Ответ: В одно место (при непосредственном отображении), в несколько мест (при множественной ассоциативности) или в любое место (при полной ассоциативности).

Вопрос 2: Как найти блок?

Ответ: Существует четыре метода: индексация (как в кэш-памяти с непосредственным отображением), ограниченный поиск (как в кэш-памяти с множественной ассоциативностью), полный поиск (как в кэш-памяти с полной ассоциативностью) и отдельная таблица поиска (как в таблице страниц).

Вопрос 3: Какой блок заменить при промахе?

Ответ: Обычно либо наименее востребованный, либо произвольный.

Вопрос 4: Как обрабатываются записи?

Ответ: На каждом уровне иерархии может использоваться либо сквозная, либо отложенная запись.

Три «С»: Интуитивная модель восприятия поведения иерархий памяти

В этом подразделе будет рассмотрена модель, дающая представление об источниках промахов в иерархии памяти и о том, как на промахи повлияют изменения в иерархии. Идеи будут объяснены в понятиях кэш-памяти, хотя эти идеи можно перенести на любой другой уровень иерархии. В данной модели все промахи классифицируются по одной из трех категорий (**три «С»**):

- **Вынужденные промахи** (Compulsory misses). Эти промахи при обращении к кэш-памяти вызваны первым обращением к блоку, который никогда не был в кэш-памяти. Они также называются **промахами начального запуска**.
- **Промахи вместимости** (Capacity misses). Эти промахи при обращении к кэш-памяти происходят в том случае, если кэш-память не может содержать все блоки, необходимые при выполнении программы. Промахи вместимости происходят, когда блоки заменяются, а позже снова извлекаются из памяти.
- **Промахи конфликтов** (Conflict misses). Эти промахи при обращении к кэш-памяти возникают в кэш-памяти с множественной ассоциативностью или в кэш-памяти с непосредственным отображением, когда несколько блоков соревнуются за место в одном и том же наборе. Промахи конфликтов относятся к таким промахам в кэш-памяти с непосредственным отображением или в кэш-памяти с множественной ассоциативностью, которые исчезают в полностью ассоциативной

Модель трех «С»
Модель кэш-памяти, в которой все промахи при обращении к ней классифицируются по одной из трех категорий: вынужденные промахи (compulsory misses), промахи вместимости (capacity misses) и промахи конфликтов (conflict misses).

Вынужденный промах (compulsory miss)
Также называется промахом начального запуска. Промах при обращении к кэш-памяти, вызванный первым обращением к блоку, который никогда не был в кэш-памяти.

Промах вместимости (capacity miss)
Промах при обращении к кэш-памяти, вызванный тем, что кэш-память, даже с полной ассоциативностью, не может содержать все блоки, необходимые для удовлетворения запроса.

Промах конфликта (conflict miss)
Также называется промахом коллизии. Промах при обращении к кэш-памяти, который возникает в кэш-памяти с множественной ассоциативностью или в кэш-памяти с непосредственным отображением, когда несколько блоков соревнуются за один и тот же набор, и который исчезает в полностью ассоциативной кэш-памяти такого же размера

кэш-памяти такого же размера. Эти промахи при обращении к кэш-памяти называются также **промахами коллизий**.

На рис. 5.24 показано деление коэффициента промахов на три источника. Эти источники промахов должны быть непосредственно разрушены путем изменения некоторых аспектов конструкции кэш-памяти. Поскольку промахи конфликтов появляются непосредственно из-за соперничества за один и тот же блок кэш-памяти, повышение степени ассоциативности сокращает количество промахов конфликтов. Но ассоциативность может замедлить доступ, что приведет к снижению общей производительности.

Рис. 5.24. Коэффициент промахов можно разбить на три источника промахов. На этой схеме показан общий коэффициент промахов и его составляющие для диапазона размеров кэш-памяти. Эти данные верны для контрольных задач по работе с целыми числами и числами с плавающей точкой SPEC2000, и получены из того же источника, что и данные, показанные на рис. 5.23. Компонент вынужденных промахов равен 0,006% и не может быть замечен на данной схеме. Следующий компонент касается коэффициента промахов вместительности и зависит от размера кэш-памяти. Компонент, касающийся конфликтов, который зависит как от степени ассоциативности, так и от размера кэш-памяти, показан для диапазона ассоциативности от одноканальной до восьмиканальной. В каждом отдельном случае подписанная часть соответствует возрастанию коэффициента промахов, когда ассоциативность изменяется от следующего, более высокого уровня до подписанного уровня. Например, та часть, которая обозначена как «двухканальная», показывает дополнительные промахи, возникающие, когда кэш-память имеет двухканальную, а не четырехканальную ассоциативность. Таким образом, разница в коэффициенте промахов, получающаяся при сравнении кэш-памяти с непосредственным отображением и полностью ассоциативной кэш-памяти одного и того же размера, складывается из суммы частей, помеченных как «четырехканальная», «двухканальная» и «одноканальная». Разница между восьмиканальной и четырехканальной частью настолько мала, что на этой схеме ее разглядеть довольно трудно

Промахи вместимости могут быть легко уменьшены путем увеличения объема кэш-памяти; и действительно, объем кэш-памяти второго уровня неизменно рос в течение многих лет. Разумеется, когда объем кэш-памяти увеличивается, нужно

же что-то сделать с увеличением времени доступа, которое может привести к снижению общей производительности. А вот объем кэш-памяти первого уровня рос медленно, если вообще можно считать это ростом.

Поскольку вынужденные промахи вызываются первым обращением к блоку, главным способом для кэш-системы снизить количество вынужденных промахов является увеличение размера блока. Тем самым будет уменьшено количество обращений, требуемых для однократного обращения к каждому блоку программы, поскольку программа будет состоять из меньшего количества блоков. Как уже ранее упоминалось, чрезмерное увеличение размера блока может негативно повлиять на производительность, поскольку увеличиваются издержки промахов.

Деление промахов на три «С» является довольно-таки полезной качественной моделью. В реальных конструкциях кэш-памяти многие конструктивные решения влияют друг на друга, и изменение одной характеристики кэш-памяти зачастую будет оказывать влияние сразу на несколько компонентов, определяющих коэффициент промахов. Несмотря на такие недостатки, эта модель является полезным способом постижения производительности различных конструкций кэш-памяти.

Общее представление

Трудности конструирования иерархий памяти состоят в том, что каждое изменение, потенциально уменьшающее коэффициент промахов, может также негативно сказаться на общей производительности, как показано в табл. 5.8. Эта комбинация позитивных и негативных эффектов и составляет привлекательность конструирования иерархии памяти.

Таблица 5.8. Трудности конструирования иерархии памяти

Конструктивное изменение	Влияние на коэффициент промахов	Возможное негативное влияние на производительность
Увеличение размера кэш-памяти	Уменьшение количества промахов вместительности	Возможно увеличение времени доступа
Увеличение степени ассоциативности	Уменьшение коэффициента промахов за счет уменьшения количества промахов конфликтов	Возможно увеличение времени доступа
Увеличение размера блока	Уменьшение коэффициента промахов для более широкого диапазона размеров блока за счет пространственной локализации	Повышение издержек промахов. Очень большие блоки могут увеличить коэффициент промахов

Самопроверка

Какие из следующих утверждений (если таковые имеются) в целом соответствуют действительности?

1. Способов уменьшения вынужденных промахов не существует.
2. В полностью ассоциативной кэш-памяти отсутствуют промахи конфликтов.
3. В уменьшении количества промахов ассоциативность важнее объема.

5.6. Виртуальные машины

Родственной виртуальной памяти и почти такой же старой является идея виртуальных машин — Virtual Machines (VM). Первые такие машины были разработаны в середине 1960-х годов и со временем стали широко использоваться в больших универсальных машинах. Несмотря на практически полное игнорирование виртуальных машин в области однопользовательских компьютеров в 1980-х и 1990-х годах, в последнее время они набрали популярность благодаря следующим факторам:

- Повышению важности изолирования и безопасности в современных системах.
- Промахам в безопасности и надежности стандартных операционных систем.
- Совместному использованию одного и того же компьютера многими пользователями.
- Существенное повышение за последние десятилетия скорости работы процессоров, которое сделало издержки использования виртуальных машин более приемлемыми.

Самое широкое определение виртуальных машин включает по существу все методы эмуляции, предоставляемые таким стандартным программным интерфейсом, как Java VM. В данном разделе наше внимание привлекут виртуальные машины, предоставляющие полноценную среду системного уровня в виде архитектуры набора двоичных инструкций — binary instruction set architecture (ISA). Хотя некоторые виртуальные машины запускают в своей среде наборы ISA отличающиеся от наборов исходного оборудования, мы будет предполагать, что эти наборы всегда соответствуют наборам оборудования. Такие виртуальные машины называются (операционными) *системами виртуальных машин*. Примерами могут послужить IBM VM/370, VMware ESX Server и Xen.

Система виртуальных машин представляет иллюзию того, что у пользователей имеется в распоряжении целый компьютер, включая копию операционной системы. На одном компьютере запускаются несколько виртуальных машин, и он может поддерживать несколько разных операционных систем. На обычной платформе единственная операционная система «владеет» всеми аппаратными ресурсами, но при использовании виртуальной машины аппаратные ресурсы совместно используются несколькими операционными системами.

Программное обеспечение, поддерживающее виртуальные машины, называется *диспетчером виртуальных машин* — virtual machine monitor (VMM), или *гипервизором*; VMM является сердцем технологии виртуальных машин. Лежащая в его основе аппаратная платформа называется хозяином, или *хостом*, а ее ресурсы распределяются между *гостевыми* виртуальными машинами. VMM определяет порядок отображения виртуальных ресурсов на физические: физические ресурсы могут распределяться во времени, по частям или даже эмулироваться программным обеспечением. VMM намного меньше традиционной операционной системы; изолированная часть VMM занимает, наверное, всего лишь около 10 000 строк кода.

Хотя наш интерес к виртуальным машинам в данном случае концентрируется на вопросе повышения защиты, виртуальные машины предоставляют два других коммерчески значимых преимущества:

Управление программным обеспечением. Виртуальные машины предоставляют абстракцию, которая может запустить полный программный пакет, даже включая старые операционные системы наподобие DOS. Обычной средой развертывания могут стать виртуальные машины с запущенными устаревшими операционными системами, множество виртуальных машин с текущими стабильными выпусками операционной системы и некоторые виртуальные машины, на которых тестируются следующие выпуски операционной системы.

Управление оборудованием. Одной из причин использования нескольких серверов является стремление к запуску каждого приложения на совместимой с ним версии операционной системы на отдельных компьютерах, поскольку такое разделение может повысить функциональную надежность. Виртуальные машины позволяют этим отдельным пакетам программного обеспечения запускаться независимо друг от друга и при этом совместно использовать оборудование, заменяя собой несколько серверов. Еще один пример заключается в том, что диспетчеры VMM поддерживают миграцию запущенной виртуальной машины на другой компьютер либо для балансировки загруженности, либо для эвакуации с отказавшего оборудования.

В целом, издержки виртуализации процессора зависят от рабочей нагрузки. Программы пользовательского уровня, зависящие только от быстродействия процессора, имеют нулевые издержки виртуализации: операционная система вызывается довольно редко, поэтому все работает на естественных скоростях. Рабочая нагрузка, предусматривающая интенсивный ввод-вывод данных, обычно также интенсивно использует операционную систему, выполняя множество системных вызовов и привилегированных инструкций, что может привести к высоким издержкам виртуализации. С другой стороны, если рабочая нагрузка, связанная с интенсивным вводом-выводом, является также *зависимой от быстродействия систем ввода-вывода*, издержки виртуализации процессора могут быть полностью скрыты, поскольку процессор часто простаивает в ожидании ввода-вывода.

Издержки определяются как количеством инструкций, которые должны быть эмулированы VMM, так и количеством времени, отнимаемым каждой из них на эмуляцию. Следовательно, когда гостевая виртуальная машина запускает ту же архитектуру набора инструкций (ISA), что и хост-машина, что и предполагается в данном случае, цель архитектуры и VMM заключается в запуске практически всех инструкций непосредственно на исходном оборудовании.

Требования, предъявляемые к диспетчеру виртуальных машин (VMM)

Чем должен заниматься диспетчер виртуальных машин? Он предоставляет программный интерфейс для гостевого программного обеспечения, а также должен изолировать гостей друг от друга и защитить самого себя от гостевого программного обеспечения (включая гостевые операционные системы). К нему предъявляются следующие требования:

- Гостевое программное обеспечение должно вести себя на виртуальной машине точно так же, как если бы оно было запущено на исходном оборудовании, за

исключением поведения, влияющего на производительность или ограничение постоянных ресурсов, совместно используемых несколькими виртуальными машинами.

- Гостевое программное обеспечение не должно иметь возможности непосредственно изменять распределение реальных системных ресурсов.

Для «виртуализации» процессора VMM должен управлять практически всем — обращением к привилегированному состоянию, преобразованием адреса, вводом-выводом, исключениями и прерываниями — даже если гостевая виртуальная машина и текущая запущенная операционная система временно их используют.

Например, в случае прерывания по таймеру VMM приостанавливает текущую запущенную гостевую виртуальную машину, сохраняет ее состояние, обрабатывает прерывание, определяет, какую из гостевых виртуальных машин запустить следующей, и затем загружает ее состояние. Гостевым виртуальным машинам, зависящим от прерывания по таймеру, предоставляется виртуальный таймер и эмулированное прерывание по таймеру со стороны VMM.

Чтобы отвечать за все это, VMM должен иметь более высокий уровень привилегий, чем гостевая виртуальная машина, которая обычно запускается в пользовательском режиме; тем самым также гарантируется, что исключение любой привилегированной инструкции будет обработано VMM. Основные требования к системе виртуальных машин почти идентичны тем перечисленным выше требованиям, которые предъявлялись к виртуальной памяти со страничной организацией:

- как минимум два режима работы процессора, системный и пользовательский;
- привилегированный поднабор инструкций, доступный только в системном режиме, приводящий при выполнении в пользовательском режиме к перехвату; все системные ресурсы должны быть подконтрольны только этим инструкциям.

Поддержка виртуальных машин архитектурой набора инструкций (нехватка инструкций)

Если применение виртуальных машин планировалось при разработке архитектуры набора инструкций, то сократить количество инструкций, которые должны выполняться VMM, и повысить скорость их эмуляции сравнительно несложно. Архитектура, позволяющая виртуальной машине работать непосредственно на оборудовании, получила название *виртуализируемой*, и такое гордое название носит архитектура машины IBM 370.

Но, увы, поскольку виртуальные машины стали рассматриваться для применения на настольных компьютерах и на серверах на основе персональных компьютеров совсем недавно, большинство наборов инструкций были созданы без учета виртуализации. В числе обвиняемых x86 и большая часть представителей RISC-архитектуры, включая ARM и MIPS.

Поскольку VMM должен гарантировать, что гостевая система будет взаимодействовать только с виртуальными ресурсами, обычная гостевая операционная система запускается в качестве программы, работающей в пользовательском ре-

ение поверх VMM. Затем, если гостевая операционная система предпринимает ытку обратиться к информации или изменить эту информацию, связанную сурсами оборудования посредством привилегированной инструкции, — напри-р, прочитать или записать указатель таблицы страниц, — произойдет перехват ередача управления VMM. Диспетчер виртуальных машин затем может внести бходимые изменения в соответствующие реальные ресурсы.

Следовательно, если любая инструкция, пытающаяся прочитать или записать кую чувствительную информацию, попадает в ловушку в процессе ее выполнения пользовательском режиме, VMM может ее перехватить и поддержать виртуаль-ую версию чувствительной информации в соответствии с ожиданиями гостевой ерационной системы.

При отсутствии такой поддержки могут быть приняты другие меры. Диспетчер иртуальных машин должен принять специальные меры предосторожности для бнаружения проблемных инструкций и гарантирования их правильного поведе-ия при выполнении гостевой операционной системой, повышая тем самым слож-ость своего устройства и снижая производительность запущенной виртуальной ашины.

Защита и архитектура набора инструкций

Защита обеспечивается общими усилиями архитектуры и операционных систем, ео разработчики архитектуры должны были внести изменения в некоторые не-удобные детали существующей архитектуры набора инструкций, когда приобрела опулярность виртуальная память. Например, для поддержки виртуальной памяти на IBM 370 разработчики вынуждены были изменить вполне удачную архитектуру набора инструкций IBM 360, которая была анонсирована шестью годами ранее. Подобные поправки были сделаны в наши дни для того, чтобы приспособиться к виртуальным машинам.

Например, x86-инструкция `POPF` загружает регистры флагов с вершины стека в память. Одним из флагов является разрешение прерываний — Interrupt Enable (IE). Если запустить инструкцию POPF в пользовательском режиме, вместо ее перехвата будут просто изменены все флаги за исключением IE. В системном режи-ме будет изменен и флаг IE. Поскольку гостевая операционная система запускается в пользовательском режиме внутри виртуальной машины, это вызывает проблему, поскольку ожидается увидеть измененный флаг IE.

Исторически оборудование больших универсальных машин IBM и VMM про-шло три этапа повышения производительности виртуальных машин:

1. Уменьшение стоимости виртуализации процессора.
2. Уменьшение издержек прерывания, связанных с виртуализацией.
3. Уменьшение издержек прерывания за счет управления прерываниями для нуж-ной виртуальной машины без обращения к VMM.

В 2006 году новые предложения от AMD и Intel были направлены в первую очередь на сокращение издержек виртуализации процессора. Будет интересно посмотреть, сколько поколений архитектуры и модификаций VMM потребуется

для решения по всем трем пунктам, и сколько времени пройдет, пока виртуальные машины XXI века станут такими же эффективными, как универсальные машины IBM и диспетчеры виртуальных машин 1970-х годов.

Уточнение. Кроме виртуализации набора инструкций есть еще одна проблема — виртуализация виртуальной памяти, поскольку каждая операционная система в каждой виртуальной машине управляет своим собственным набором таблиц страниц. Чтобы справиться с этой работой, VMM разделяет понятия *реальной* и *физической памяти* (которые часто считаются синонимами) и делает реальную память отдельным, промежуточным уровнем между виртуальной и физической памятью. (Для названия этих же трех уровней используются также термины *виртуальная память*, *физическая память* и *машинная память*.) Гостевая операционная система отображает виртуальную память на реальную через свои таблицы страниц, а таблицы страниц VMM отображают реальную память гостевой системы на физическую память. Архитектура виртуальной памяти определена либо через таблицы страниц, как в IBM VM/370 и в x86, либо через структуру TLB, как в MIPS.

Вместо того чтобы платить за дополнительный уровень «косвенности» при каждом обращении к памяти, VMM поддерживает теневую таблицу страниц, которая осуществляет непосредственное отображение гостевого виртуального адресного пространства на физическое адресное пространство оборудования. Путем обнаружения всех изменений в гостевой таблице страниц VMM может гарантировать, что используемые оборудованием при преобразовании адресов записи в теневой таблице страниц соответствуют записям окружения гостевой операционной системы, за исключением соответствующих физических страниц, подставляемых вместо реальных страниц в гостевых таблицах. Следовательно, VMM должен перехватывать любые попытки гостевой операционной системы изменить ее таблицы страниц или обратиться к указателю таблицы страниц. Обычно это делается путем защиты от записи гостевых таблиц страниц и перехвата любого обращения к указателю таблицы страниц со стороны гостевой операционной системы. Как отмечалось ранее, последнее действие происходит вполне естественно, если обращение к указателю таблицы страниц является привилегированной операцией.

Последней частью архитектуры, подвергаемой виртуализации, является ввод-вывод. Это самая трудная часть системной виртуализации, что обусловлено постоянно возрастающим количеством устройств ввода-вывода, подключаемых к компьютеру, и растущим разнообразием этих устройств. Еще одной сложностью является совместное использование реальных устройств несколькими виртуальными машинами, и эта сложность усугубляется еще и необходимостью поддержки множества драйверов устройств, которые требуются особенно в том случае, если на одной и той же системе виртуальных машин поддерживаются разные гостевые операционные системы. Иллюзия виртуальной машины может поддерживаться путем предоставления каждой виртуальной машине универсальных версий драйверов устройств ввода-вывода каждого типа и передачи полномочий управления реальным вводом-выводом диспетчеру VMM.

5.7. Использование конечного автомата для управления простой кэш-памятью

Теперь мы можем реализовать управление кэш-памятью, точно так же, как мы создавали систему управления для однотактного и конвейеризированного операционного блока в главе 4. Этот раздел начинается с определения простой кэш-памяти, а затем дается описание конечного автомата — finite-state machines (FSM). В завершение рассматривается FSM контроллера для этой простой кэш-памяти.

Простая кэш-память

Мы собираемся разработать контроллер для простой кэш-памяти. Эта кэш-память имеет следующие основные характеристики:

- непосредственное отображение;
- отложенная запись с выделением для записи;
- размер блока 4 слова (16 байт или 128 бит);
- размер кэш-памяти 16 Кбайт, следовательно, в ней содержится 1024 блока;
- 32-разрядная адресация;
- кэш-память использует для каждого блока бит достоверности и бит изменения.

Пользуясь материалом раздела 5.2, мы можем вычислить поля адреса для кэш-памяти:

- кэш-индекс занимает 10 разрядов;
- смещение блока занимает 4 разряда;
- размер тега составляет 32 – (10 + 4), или 18 разрядов.

Между процессором и кэш-памятью будут использоваться следующие сигналы:

- одноразрядный сигнал Чтения (Read) или Записи (Write);
- одноразрядный сигнал Достоверности (Valid), свидетельствующий о том, что операция проводится именно с кэш-памятью;
- 32-разрядный адрес;
- 32-разрядные данные от процессора к кэш-памяти;
- 32-разрядные данные от кэш-памяти к процессору;
- одноразрядный сигнал Готовности (Ready), свидетельствующий о завершении операции с кэш-памятью.

Следует отметить, что это блокирующая кэш-память, заставляющая процессор ждать, пока она не обработает запрос.

Интерфейс между оперативной и кэш-памятью имеет те же поля, что и интерфейс между процессором и кэш-памятью, за исключением того, что данные имеют ширину 128 разрядов. Расширенный формат памяти не редкость для сегодняшних микропроцессоров, которые работают с 32-разрядными или с 64-разрядными словами, в то время как DRAM-контроллер зачастую работает с 128-разрядами. При-

ведение блока кэш-памяти в соответствие ширине DRAM упрощает конструкцию. Таким образом, используются следующие сигналы:

- одноразрядный сигнал Чтения (Read) или Записи (Write);
- одноразрядный сигнал Достоверности (Valid), свидетельствующий о том, что операция проводится с памятью;
- 32-разрядный адрес;
- 128-разрядные данные от кэш-памяти к оперативной памяти;
- 128-разрядные данные от оперативной памяти к кэш-памяти;
- одноразрядный сигнал Готовности (Ready), свидетельствующий о завершении операции с памятью.

Следует учесть, что на интерфейс с памятью затрачивается нефиксированное количество циклов. Предполагается использование контроллера памяти, который уведомит кэш-память сигналом готовности — Ready, когда закончится чтение памяти или запись в нее.

Перед тем как дать описание контроллера кэш-памяти, нужно рассмотреть конечные автоматы, позволяющие нам управлять операцией, которая может занять несколько тактовых циклов.

Конечные автоматы

Для блока управления однотактного операционного блока мы использовали набор таблиц истинности, определяющих выставления сигналов управления на основе класса инструкции. Для кэш-памяти используется более сложная система управления, поскольку операция может быть представлена серией шагов. Система управления кэш-памяти должна определять как выставленные сигналы для любого из шагов, так и следующий шаг.

Наиболее распространенный многошаговый метод управления базируется на использовании **конечных автоматов**, которые обычно представляются в графическом виде. Конечный автомат состоит из набора состояний и указаний о том, как изменять состояния. Указания определяются **функцией перехода** (next-state function), которая отображает текущее состояние и входные значения в новое состояние. Когда конечные автоматы используются для управления, каждое состояние помимо всего прочего определяет набор выходных сигналов, которые выставляются при нахождении автомата в данном состоянии. Реализация конечного автомата обычно предполагает, что все выходные сигналы, которые не выставлены явным образом, считаются невыставленными. Точно так же, соответствующая операция операционного блока зависит от того факта, что сигнал, который не был

Конечный автомат
Последовательная логическая функция, состоящая из набора входных и выходных сигналов, функции перехода, отображающей текущее состояние и входные значения в новое состояние, и функции выхода, отображающей текущее состояние и, возможно, входные значения для выставления выходных сигналов.

Функция перехода
Комбинационная функция, которая на основании входных данных и текущего состояния определяет следующее состояние конечного автомата.

ставлен явным образом, является невыставленным, если только он не является этой операции безразличным.

Сигналы управления мультиплексором несколько отличаются, поскольку здесь исходит выбор одного из входов в зависимости от того, равны ли они 0 или 1. ким образом, в конечном автомате мы всегда определяем выставление всех ходимых сигналов управления мультиплексором. Когда конечный автомат изуется с использованием логических схем, установка сигнала управления ль может быть значением по умолчанию, не требующим никаких логических ментов.

Конечный автомат может быть реализован с использованием временного регистра, содержащего текущее состояние, и блока комбинационной логики, определяющего и выставляемые сигналы операционного блока, и следующее состояние. рис. 5.25 показано, как может выглядеть подобная реализация.

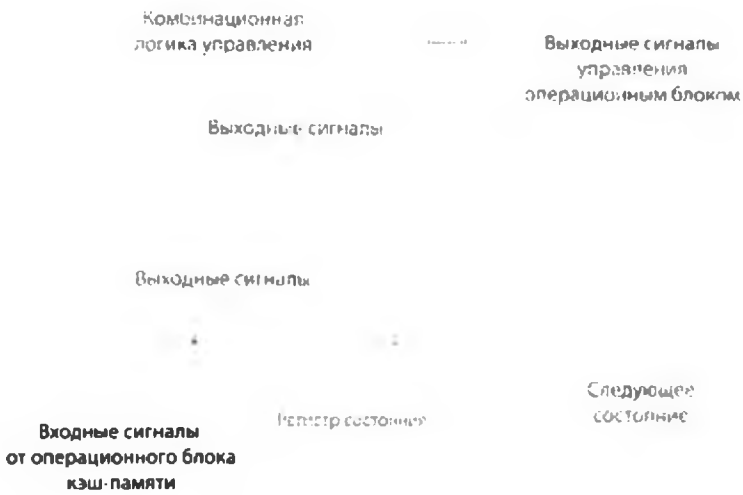


Рис. 5.25. Контроллеры на основе конечных автоматов обычно реализуются с использованием блока комбинационной логики и регистра, хранящего текущее состояние. Выходные сигналы комбинационной логики являются номером следующего состояния и сигналами управления, выставляемыми для текущего состояния. Входными сигналами для комбинационной логики являются текущее состояние и любые входные сигналы, используемые для определения следующего состояния. В данном случае входные сигналы являются битами регистра кода операции (opcode). Учтите, что в конечном автомате, рассматриваемом в данной главе, выходные сигналы зависят только от текущего состояния, а не от входных сигналов. Более подробно это объясняется в следующем уточнении

Уточнение. По типу конечный автомат, рассматриваемый в данной книге, относится к машинам Мура, названным по имени Эдварда Мура (Edward Moore). Его отличительные признаки заключаются в том, что выходные сигналы зависят только от текущего

состояния. Для машины Мура прямоугольник, имеющий надпись «Комбинационная логика управления», может быть разбит на две части. У одной из них имеется управляющий выход и только вход состояния, а у другого имеется только выход следующего состояния (перехода).

Другим типом конечного автомата является машина Мили, названная по имени Джорджа Мили (George Mealy). Машина Мили позволяет использовать для определения выхода и вход, и текущее состояние. Машины Мура обладают потенциальными преимуществами реализации в скорости и размере блока управления. Скоростные преимущества возникают благодаря тому, что управляющие выходы, которые нужны на ранней стадии тактового цикла, не зависят от входов и формируются только на основе текущего состояния. Потенциальный недостаток машины Мура заключается в том, что ей могут понадобиться дополнительные состояния. Например, в ситуациях, при которых два следующих друг за другом состояния отличаются только одним параметром, машина Мили может унифицировать состояния, сделав выход зависимым от входа.

Конечный автомат для контроллера простой кэш-памяти

На рис. 5.26 показаны четыре состояния контроллера простой кэш-памяти:

- *Бездействие*. Это состояние ожидания допустимого запроса от процессора на чтение или запись, который переводит конечный автомат в состояние сравнения тега.
- *Сравнение тега*. Судя по названию, в этом состоянии происходит тестирование с целью обнаружения попадания или промаха запроса на чтение или запись. С помощью индексной части адреса выбирается сравниваемый тег. Если индексная часть допустима и тег-часть адреса соответствует тегу, происходит попадание. Данные либо считываются из выбранного слова, либо записываются в выбранное слово, после чего выставляется сигнал о готовности кэш-памяти (Cache Ready). Если производится запись, устанавливается единичное значение бита изменения. Следует заметить, что при попадании запроса на запись устанавливается также бит достоверности и поле тега. Это может показаться ненужным, но тем не менее осуществляется — тег находится в едином элементе памяти, поэтому для выставления единицы в бите изменения нам также нужно внести изменения в поля достоверности и тега. Если произошло попадание и блок помечен как достоверный, конечный автомат возвращается в состояние бездействия. При промахе сначала обновляется значение тега кэш-памяти, а затем происходит переход либо в состояние отложенной записи, если блок в данном месте имел установленный в единицу бит изменения, либо в состояние распределения, если значение этого бита было равно нулю.
- *Отложенная запись*. В этом состоянии 128-разрядный блок записывается в память с использованием адреса, составленного из тега и индекса кэш-памяти. При этом нужно дождаться сигнала готовности от памяти. Когда запись в память завершится, конечный автомат переходит в состояние распределения.

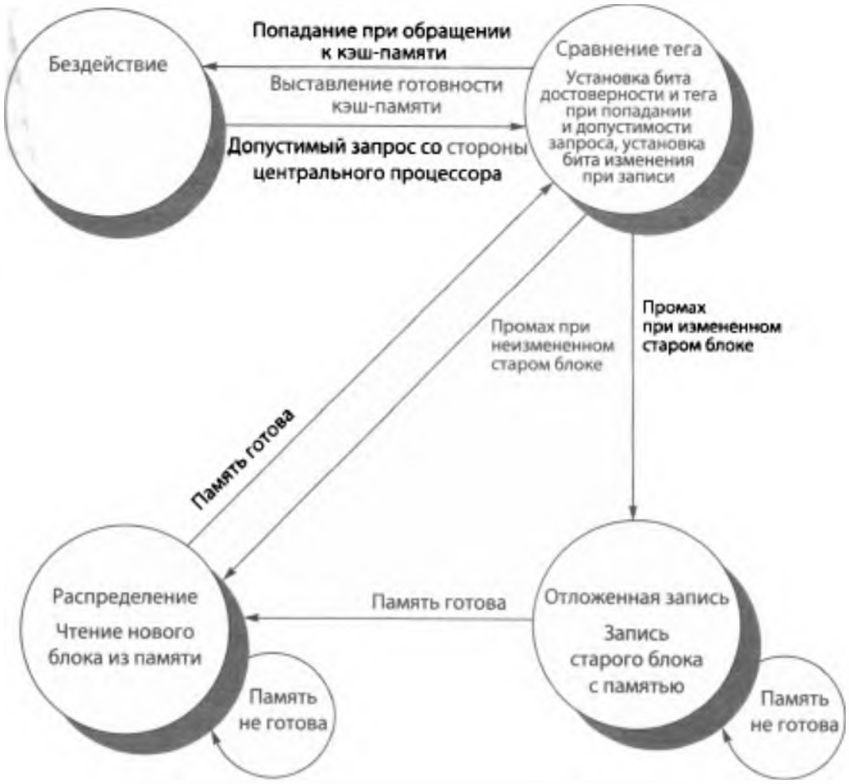


Рис. 5.26. Четыре состояния простого контроллера

- *Распределение*. В этом состоянии из памяти извлекается новый блок. При этом нужно дождаться сигнала готовности от памяти. Когда чтение из памяти завершится, конечный автомат переходит в состояние сравнения тега. Хотя можно было бы перейти к новому состоянию, чтобы завершить операцию, а не использовать снова состояние сравнения тега, существует множество совпадений, включая обновление соответствующего слова в блоке, если было обращение по записи.

Для улучшения производительности эта простая модель легко может быть расширена дополнительными состояниями. Например, в состоянии сравнения тега в одном и том же тактовом цикле осуществляется как сравнение, так чтение или запись данных кэш-памяти. Чтобы уменьшить продолжительность тактового цикла, сравнение и обращение к кэш-памяти часто осуществляются в разных состояниях. Еще один способ оптимизации может заключаться в добавлении буфера записи, чтобы можно было сохранить измененный блок, а затем сначала считать новый блок, чтобы процессору не пришлось ожидать двух обращений к памяти при промахе и замене измененного блока. Потом кэш-память запишет измененный блок из буфера записи, пока процессор будет обрабатывать запрошенные данные.

5.8. Параллелизм и иерархии памяти: целостность данных в кэш-памяти

Учитывая, что многоядерный мультипроцессор подразумевает наличие нескольких процессоров на одном кристалле, вполне вероятно предположить, что эти процессоры совместно используют общее адресное пространство. Кэширование совместно используемых данных создает новую проблему, поскольку вид на память, имеющийся у двух разных процессоров, формируется через их индивидуальные устройства кэш-памяти, это без дополнительных мер предосторожности может привести к тому, что они будут видеть два разных значения в одном и том же месте. Таблица 5.9 иллюстрирует проблему и показывает, как два различных процессора могут получить два разных значения в одном и том же месте в памяти. Это осложнение часто называют проблемой *поддержки целостности данных в кэш-памяти*.

Неформально можно сказать, что система памяти сохраняет целостность, если при любом считывании элемента данных возвращается самое последнее записанное значение этого элемента. Это определение, несмотря на его интуитивную привлекательность, грешит нечеткостью и упрощенностью, в реальности все значительно сложнее. В этом простом определении содержатся два разных аспекта поведения систем памяти, каждый из которых является очень важным для написания программ, корректно использующих общую память. Первый аспект, называемый *целостностью*, определяет, *какие значения* могут быть возвращены в результате чтения.

Таблица 5.9. Проблема целостности данных в отдельном месте в памяти (X), откуда ведется чтение и куда осуществляется запись двумя процессорами (А и Б). Изначально предполагается, что ни одна кэш-память не содержит переменную и что значение в X равно нулю. Также предполагается использование кэш-памяти со сквозной записью; кэш-память с отложенной записью создает дополнительные, но похожие трудности. После того как значение в X было записано процессором А, кэш-память этого процессора и место в памяти содержат новое значение, а кэш-память процессора Б не содержит этого значения — если процессор Б прочитает значение места X, ему будет возвращен нуль!

Отрезок времени	Событие	Содержимое кэш-памяти процессора А	Содержимое кэш-памяти процессора Б	Содержимое памяти в месте X
0				0
1	Процессор А читает X	0		0
2	Процессор Б читает X	0	0	0
3	Процессор А сохраняет 1 в X	1	0	1

Второй аспект, называемый *последовательностью*, определяет, когда записанное значение будет возвращено в результате чтения.

Сначала рассмотрим целостность данных. Система памяти считается целостной, если:

1. Чтение, осуществляемое процессором P из места X, которое следует за записью процессора P в место X, без каких-либо записей в X со стороны других процессоров в промежуток времени между записью и чтением, осуществляемыми процессором P, всегда возвращают значение, записанное процессором P. Таким образом, в показанной выше табл. 5.9, если процессор A был бы должен прочитать данные из места X после отрезка времени 3, он должен был бы увидеть там значение 1.
2. Чтение процессором данных из места X, которое следует за записью в место X, осуществленной другим процессором, возвращает записанное значение, если чтение и запись четко разнесены по времени и между этими двумя обращениями к памяти никаких записей в место X не велось. Таким образом, при ситуации, показанной в табл. 5.9, нам нужен механизм, заменяющий значение 0 в кэш-памяти процессора Б значением 1 после того, как процессор A сохраняет 1 в памяти по адресу X за отрезок времени 3.
3. Запись в одно и то же место ведется последовательно; то есть две операции записи двумя процессорами в одно и то же место видны в том же порядке всеми процессорами. Например, если процессор Б сохраняет 2 в памяти по адресу X после отрезка времени 3, процессоры уже никогда не смогут сначала прочитать значение по адресу X как 2, а чуть позже прочитать его как 1.

Первое свойство просто сохраняет порядок выполнения программы, к примеру наличие этого свойства безусловно ожидается в однопроцессорных системах. Второе свойство определяет свое значение наличием целостного взгляда на память: если процессор может постоянно считывать устаревшее значение данных, можно абсолютно точно заявить о том, что память потеряла свою целостность.

Потребность в *обеспечении последовательности записи* менее очевидна, но не менее важна. Предположим, что последовательность записи не выстроена и процессор P1 записывает данные в место X, а за ним в это же место ведет запись процессор P2. Выстраивание последовательности операций записи гарантирует, что каждый процессор в определенный момент времени увидит запись, сделанную процессором P2. Если мы не выстроим последовательность операций записи, может получиться так, что некоторые процессоры смогут увидеть запись, произведенную процессором P2, первой, а затем увидят запись, произведенную процессором P1, бесконечно придерживаясь значения, записанного процессором P1. Проще всего избавиться от подобных осложнений за счет гарантии того, что все операции записи в одно и то же место будут видны в одном и том же порядке; это свойство называется *обеспечением последовательности записи*.

Основные схемы поддержания целостности данных

В многопроцессорной системе с обеспечением целостности данных кэш-памяти такая кэш-память предоставляет *миграцию* и *репликацию* общих элементов данных:

- *Миграция*. Элемент данных может быть перемещен в локальную кэш-память и использован там в явном виде. Миграция сокращает как латентность доступа к совместно используемому дистанционно распределяемому элементу данных, так и требования по ширине полосы пропускания совместно используемой памяти.
- *Репликация*. Когда происходит одновременное считывание совместно используемых данных, устройства кэш-памяти создают копию элемента данных в локальной кэш-памяти. Репликация сокращает как латентность доступа, так и конкуренцию при чтении общего элемента данных.

Поддержка миграции и репликации настолько важна для производительности при обращении к общим данным, что на многих мультипроцессорных системах внедрен аппаратный протокол для поддержки устройств кэш-памяти с обеспечением целостности данных. Протоколы для поддержки целостности данных в мультипроцессорах называются *протоколами поддержания целостности данных в кэш-памяти*. Ключевой особенностью реализации протокола поддержания целостности данных кэш-памяти является отслеживание состояния любого совместно используемого блока данных.

Наиболее популярным протоколом поддержания целостности данных кэш-памяти является *отслеживание* (snooping). Каждое устройство кэш-памяти, имеющее копию данных из блока физической памяти, также имеет копию статуса общего использования блока, но централизованно это состояние не сохраняется. Все устройства кэш-памяти доступны по какому-нибудь средству обмена данными (шине или сети), и все контроллеры устройств кэш-памяти *следят (snoop)* за передающей средой, чтобы определить, имеется ли доступная копия запрошенного блока на шине или в коммутаторе.

В следующем разделе будет рассмотрена поддержка целостности данных кэш-памяти на основе отслеживания, реализованная на основе общей шины, но для реализации схемы обеспечения целостности на основе отслеживания может использоваться любое средство обмена информацией, которое распространяет промахи при обращении к кэш-памяти на все процессоры. Такое распространение информации на все устройства кэш-памяти упрощает реализацию протоколов отслеживания, но при этом ограничивает их масштабируемость.

Протоколы отслеживания

Один из методов обеспечения целостности данных заключается в гарантии того, что у процессора имеется исключительный доступ к элементу данных до того, как он произведет запись в этот элемент. Протокол такого типа называется *протоко-*

аннулирования записи, поскольку он при записи аннулирует копии в других устройствах кэш-памяти. Исключительный доступ гарантирует отсутствие при записи других, доступных для чтения или записи копий этого элемента: все другие попавшие в устройства кэш-памяти копии элемента аннулируются.

В табл. 5.10 показан пример работы протокола аннулирования для отслеживаемой шины с устройствами кэш-памяти, использующими отложенную запись. Чтобы посмотреть, как этот протокол гарантирует целостность данных, рассмотрим запись, за которой следует чтение со стороны другого процессора: поскольку для записи требуется исключительный доступ, любые копии должны быть аннулированы (согласно названию протокола). Таким образом, когда происходит чтение, возникает промах при обращении к кэш-памяти, и кэш-память вынуждена извлечь новую копию данных. Для записи требуется, чтобы записывающий процессор имел исключительный доступ, препятствуя возможности одновременной записи со стороны любого другого процессора. Если два процессора предпринимают попытку одновременной записи одних и тех же данных, один из них побеждает в соревновании, приводя к аннулированию копии данных другого процессора. Чтобы другой процессор завершил свою запись, он должен получить новую копию данных, которая теперь должна содержать обновленное значение. Следовательно, этот протокол также навязывает обеспечение последовательности записи.

Интерфейс аппаратного и программного обеспечения

Размер блока играет важную роль в поддержании целостности данных кэш-памяти. Возьмем, к примеру, отслеживание кэш-памяти с размером блока, составляющим восемь слов, где одно из слов попеременно записывается и читается двумя процессорами. Большинство протоколов производят между процессорами обмены целыми блоками, увеличивая таким образом пропускную способность, необходимую для поддержания целостности данных.

Большие по размеру блоки могут также стать причиной такого явления, как **ложное совместное использование**: когда две не связанные друг с другом совместно используемые переменные расположены в одном и том же блоке кэш-памяти, процессоры обмениваются целым блоком, даже если эти процессоры обращаются к разным переменным. Во избежание ложного совместного использования программисты и компиляторы должны обратить особое внимание на размещение данных.

Уточнение. Хотя перечисленных ранее трех свойств, обеспечивающих согласованность, вполне достаточно для поддержания целостности данных, не менее важен момент времени, когда записанное значение станет видимым. Чтобы понять почему, обратим внимание на то, что мы не можем требовать, чтобы при чтении значения по адресу X в табл. 5.10 сразу же было видно значение, за-

Ложное совместное использование
Когда две несвязанные друг с другом переменные расположены в одном и том же блоке кэш-памяти и между процессорами происходит обмен целым блоком, даже если процессоры обращаются к разным переменным.

писанное по адресу X каким-нибудь другим процессором. Если, к примеру, запись по адресу X на одном процессоре предшествует слишком раннему чтению по адресу X на другом процессоре, может быть, невозможно будет гарантировать, что чтение вернет значение записанных данных, поскольку эти данные к тому времени могли еще даже не покинуть процессора. Вопрос о том, *когда* именно записанное значение должно стать видимым читателю, определяется *моделью целостности памяти*.

Таблица 5.10. Пример протокола аннулирования, работающего на отслеживаемой шине для одного блока кэш-памяти (X), использующей принцип отложенной записи. Предполагается, что изначально ни в одном из устройств кэш-памяти не содержится значение из адреса X, и значение по адресу X в памяти равно нулю. В содержимом процессора и памяти показывается значение после завершения действия процессора и шины. Пустые места свидетельствуют об отсутствии действия или отсутствии кэширования копии. Когда происходит второй промах при обращении процессора Б к кэш-памяти, процессор А реагирует значением, отменяющим ответ из памяти. Кроме этого обновляется как содержимое кэш-памяти процессора Б, так и содержимое памяти по адресу X. Это обновление памяти, происходящее в момент совместного использования блока, упрощает протокол, но существует возможность отследить принадлежность и провести принудительную обратную запись только при замене блока. Для этого требуется ввести дополнительное состояние под названием «владелец», которое покажет, что блок может использоваться совместно, но процессор-владелец отвечает за обновление кэш-памяти любого другого процессора и памяти, когда он вносит в блок изменение или заменяет этот блок

Действие процессора	Действие шины	Содержимое кэш-памяти процессора А	Содержимое кэш-памяти процессора Б	Содержимое памяти по адресу X
				0
Процессор А читает значение по адресу X	Промах при обращении к кэш-памяти по адресу X	0		0
Процессор Б читает значение по адресу X	Промах при обращении к кэш-памяти по адресу X	0	0	0
Процессор А записывает значение 1 по адресу X	Аннулирование элемента по адресу X	1		0
Процессор Б читает значение по адресу X	Промах при обращении к кэш-памяти по адресу X	1	1	1

Мы делаем следующие два допущения. Во-первых, запись не завершается (не дает возможность произойти следующей операции записи), пока все процессоры не увидят эффект от этой операции записи. Во-вторых, процессор не изменяет порядок любой операции записи относительно любого другого обращения к памяти. Эти два условия означают, что если процессор ведет запись по адресу X, за которым следует адрес Y, любой процессор, который видит новое значение по адресу Y, должен также видеть новое значение по адресу X. Эти ограничения позволяют процессору менять порядок чтения, но заставляют процессор завершить запись в порядке выполнения программы.

Уточнение. Хотя задачи поддержания целостности данных для мультипроцессоров и систем ввода-вывода (см. главу 6) изначально имеют сходный характер, они обладают разными характеристиками. В отличие от систем ввода-вывода, где наличие нескольких копий данных — случай довольно редкий, и является тем самым случаем, от которого стараются избавиться при первой же возможности, программа, запущенная на нескольких процессорах, обычно будет иметь копии одних и тех же данных в нескольких устройствах кэш-памяти.

Уточнение. Кроме протокола отслеживания для поддержания целостности данных кэш-памяти, где происходит распространение статуса совместно используемых блоков, существует протокол на *основе каталога*, который сохраняет статус совместного использования блока физической памяти только в одном месте, называемом каталогом. Поддержание целостности данных с использованием протокола на основе каталога имеет несколько более высокие издержки реализации, чем отслеживание, но может сократить трафик между устройствами кэш-памяти и благодаря этому быть расширено на большее количество процессоров.

5.9. Реальное оборудование: иерархии памяти AMD Opteron X4 (Barcelona) и Intel Nehalem

В данном разделе будет рассмотрена иерархия памяти в двух современных микропроцессорах: the AMD Opteron X4 (Barcelona) и Intel Nehalem. На рис. 5.27 показана фотография кристалла процессора Intel Nehalem, а на рис. 1.9 в главе 1 — фотография кристалла процессора AMD Opteron X4. У обоих кристаллов имеется вторичная и третичная кэш-память на главном кристалле процессора. Такая интеграция сокращает время доступа к кэш-памяти низшего уровня, а также сокращает количество выводов на кристалле, поскольку отпадает необходимость в шине к внешней вторичной кэш-памяти. На обоих кристаллах расположены контроллеры памяти, что сокращает латентность обращения к оперативной памяти.

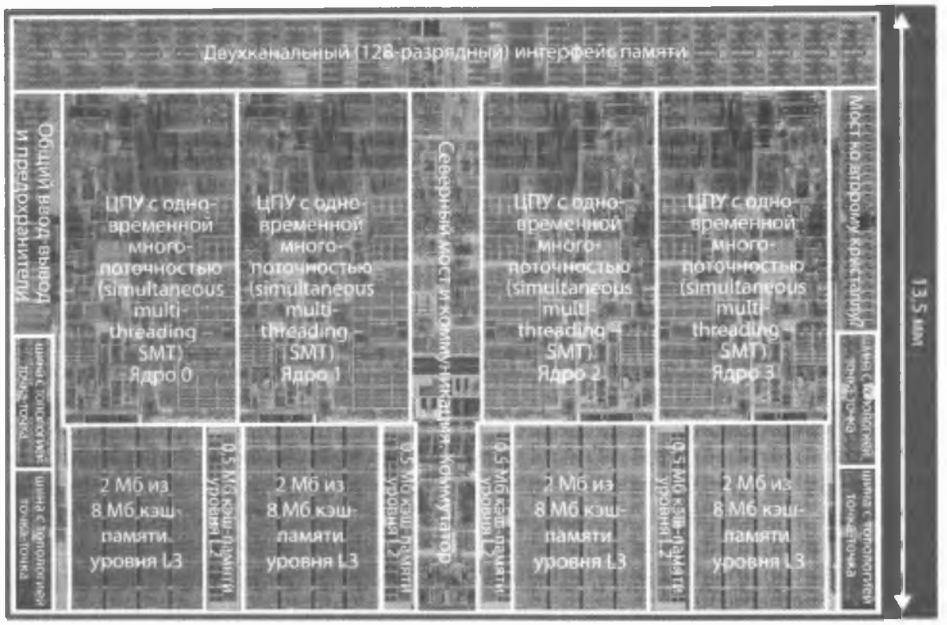


Рис. 5.27. Фотография процессора Intel Nehalem с подписанными компонентами. На этом кристалле 13,5 на 19,6 мм находится 731 млн транзисторов. Он содержит четыре процессора, у каждого из которых имеется своя собственная кэш-память инструкций объемом 32 Кбайт и такая же кэш-память данных уровня L1, а также кэш-память уровня L2 объемом 512 Кбайт. Четыре ядра совместно используют кэш-память уровня L3 объемом 8 Мбайт. К DDR3 DRAM ведут 128-разрядные каналы памяти. Каждое ядро также имеет двухуровневый буфер TLB. Теперь контроллер памяти находится на кристалле, поэтому уже нет отдельной микросхемы северного моста, как в наборе Intel Clovertown

Иерархии памяти Nehalem и Opteron

В табл. 5.11 показана сводка размеров адресов и буферов TLB двух процессоров. Следует отметить, что в AMD Opteron X4 (Barcelona) имеется четыре буфера TLB и что виртуальные и физические адреса не совпадают с размером слова. В X4 задействуется только 48 из потенциально возможных 64 разрядов его виртуального пространства, и 48 из потенциально возможных 64 разрядов его физического адресного пространства. В Nehalem имеется три буфера TLB, виртуальный адрес состоит из 48 разрядов, а физический адрес — из 44 разрядов.

Их устройства кэш-памяти показаны в табл. 5.12. Каждый процессор в X4 располагает своими собственными устройствами кэш-памяти уровня L1 для инструкций и данных, каждое из которых имеет размер 64 Кбайт, и свое собственное устройство кэш-памяти L2 размером 512 Кбайт. Четыре процессора совместно используют единую кэш-память уровня L3 размером 2 Мбайт. У кристалла Nehalem сходная структура, где у каждого процессора имеются свои собственные устройства кэш-памяти уровня L1 для инструкций и данных, каждое размером 32 Кбайт, и свое собственное устройство кэш-памяти уровня L2 размером 256 Кбайт, а четыре процессора совместно используют единую кэш-память уровня L3 размером 8 Мбайт.

Таблица 5.11. Аппаратное обеспечение для преобразования адреса и TLB в Intel Nehalem и AMD Opteron X4. Размер слова устанавливает максимальный размер виртуального адреса, но процессору не нужно использовать все разряды. Оба процессора предоставляют поддержку больших страниц, которые используются, к примеру, операционными системами или системами отображения буфера кадров. Схемы больших страниц избегают большого количества элементов для отображения всегда существующего объекта. Nehalem поддерживает два аппаратно-поддерживаемых потока в каждом ядре (см. раздел 7.5 в главе 7)

Характеристика	Intel Nehalem	AMD Opteron X4 (Barcelona)
Виртуальный адрес	48 разрядов	48 разрядов
Физический адрес	44 разряда	48 разрядов
Размер страницы	4 Кбайт, 2/4 Мбайт	4 Кбайт, 2/4 Мбайт
Организация TLB	1 TLB для инструкций и 1 TLB для данных на каждое ядро	1 L1 TLB для инструкций и 1 L1 TLB для данных на каждое ядро
	Оба L1 TLB-буфера имеют четырехканальную ассоциативность, LRU-замену	Оба L1 TLB-буфера являются полностью ассоциативными, LRU-замена
	L2 TLB имеет четырехканальную ассоциативность, LRU-замену	1 L2 TLB для инструкций и 1 L2 TLB для данных на каждое ядро
	L1 TLB инструкций имеет 128 элементов для небольших страниц, 7 на каждый поток — для больших страниц	Оба L2 TLB-буфера имеют четырехканальную ассоциативность, циклический алгоритм замены
	L1 TLB данных имеет 64 элемента для небольших страниц, 32 — для больших страниц	Оба L1 TLB-буфера имеют 48 элементов
	L2 TLB имеет 512 элементов	Оба L2 TLB-буфера имеют 512 элементов
	Промахи при обращении к TLB обрабатываются аппаратно	Промахи при обращении к TLB обрабатываются аппаратно

В табл. 5.13 приведены показатели CPI, коэффициенты промахов на тысячу инструкций для устройств кэш-памяти уровней L1 и L2, и количество обращений к DRAM на тысячу инструкций для Opteron X4, выполняющего контрольные задачи SPECint 2006. Обратите внимание на высокий уровень корреляции показателей CPI и коэффициентов промахов при обращении к кэш-памяти. Коэффициент корреляции набора показателей CPI и набора коэффициентов промахов при обращениях к кэш-памяти уровня L1 на 1000 инструкций равен 0,97. Хотя мы не располагаем реальным коэффициентом промахов при обращениях к кэш-памяти уровня L3, мы можем сделать выводы об эффективности кэш-памяти уровня L3 за счет сокращения обращений к DRAM по сравнению с коэффициентом промахов при обращениях к кэш-памяти уровня L2. Хотя некоторые программы существенно выигрывают от использования 2-мегабайтной кэш-памяти уровня L3 — h264avc, hmmer и bzip2, — большинство программ не имеют такого выигрыша.

Таблица 5.12. Устройство кэш-памяти первого, второго и третьего уровня в Intel Nehalem и AMD Opteron X4 2356 (Barcelona)

Характеристика	Intel Nehalem	AMD Opteron X4 (Barcelona)
Организация кэш-памяти уровня L1	Раздельные устройства кэш-памяти инструкций и данных	Раздельные устройства кэш-памяти инструкций и данных
Размер кэш-памяти уровня L1	По 32 Кбайт для инструкций и данных на каждое ядро	По 64 Кбайт для инструкций и данных на каждое ядро
Ассоциативность кэш-памяти уровня L1	4-канальная (для инструкций), 8-канальная (для данных) ассоциативность	2-канальная ассоциативность
Алгоритм замены в L1	Приблизительная LRU-замена	LRU-замена
Размер блока в L1	64 байта	64 байта
Стратегия записи в L1	Отложенная запись, пакетная запись	Отложенная запись, пакетная запись
Время попадания в L1 (загрузка-использование)	Сведения недоступны	3 тактовых цикла
Организация кэш-памяти уровня L2	Объединенная (инструкции и данные) на каждое ядро	Объединенная (инструкции и данные) на каждое ядро
Размер кэш-памяти уровня L2	256 Кбайт (0,25 Мбайт)	512 Кбайт (0,5 Мбайт)
Ассоциативность кэш-памяти уровня L2	8-канальная ассоциативность	16-канальная ассоциативность
Алгоритм замены в L2	Приблизительная LRU-замена	Приблизительная LRU-замена
Размер блока в L2	64 байта	64 байта
Стратегия записи в L2	Отложенная запись, пакетная запись	Отложенная запись, пакетная запись
Время попадания в L2	Сведения недоступны	9 тактовых циклов
Организация кэш-памяти уровня L3	Объединенчая (инструкции и данные)	Объединенная (инструкции и данные)
Размер кэш-памяти уровня L3	8192 Кбайт (8 Мбайт), общая	2048 Кбайт (2 Мбайт), общая
Ассоциативность кэш-памяти уровня L3	16-канальная ассоциативность	32-канальная ассоциативность
Алгоритм замены в L3	Сведения недоступны	Замена блока, совместно используемого наименьшим количеством ядер
Размер блока в L3	64 байта	64 байта
Стратегия записи в L3	Отложенная запись, пакетная запись	Отложенная запись, пакетная запись
Время попадания в L3	Сведения недоступны	38 (?) тактовых циклов

Таблица 5.13. Показатели CPI, коэффициенты промахов и обращения к DRAM для иерархии памяти процессора Opteron модели X4 2356 (Barcelona), выполняющего контрольные задачи SPECint2006. К сожалению, счетчики промахов при обращении к кэш-памяти уровня L3 на этом кристалле не работают, поэтому для оценки эффективности кэш-памяти уровня L3 у нас есть только количество обращений к DRAM. Обратите внимание, что эта таблица относится к тем же системам и контрольным задачам, что и табл. 1.5 в главе 1

Название	CPI	Количество промахов при обращениях к кэш-памяти данных уровня L1 на 1000 инструкций	Количество промахов при обращениях к кэш-памяти данных уровня L2 на 1000 инструкций	Количество обращений к DRAM на 1000 инструкций
perl	0,75	3,5	1,1	1,3
bzip2	0,85	11,0	5,8	2,5
gcc	1,72	24,3	13,4	14,8
mcf	10,00	106,8	88,0	88,5
go	1,09	4,5	1,4	1,7
hmmer	0,80	4,4	2,5	0,6
sjeng	0,96	1,9	0,6	0,8
libquantum	1,61	33,0	33,1	47,7
h264avc	0,80	8,8	1.6	0,2
omnetpp	2,94	30,9	27,7	29.8
astar	1,79	16,3	9,2	8,2
xalancbmk	2,70	38,0	15,8	11,4
Median	1,35	13,6	7,5	5,4

Технологии уменьшения издержек промахов

Как у Nehalem, так и у Opteron X4 имеются дополнительные средства оптимизации, позволяющие им уменьшить издержки промахов. Первая из них, как описано в уточнении в разделе «Конструирование системы памяти с поддержкой кэш-памяти», заключается в том, что в случае промаха первым возвращается запрошенное слово. Обе системы позволяют процессору продолжать выполнение инструкций, которые обращаются к кэш-памяти данных во время обработки промаха. Эта технология, которая называется **неблокирующейся кэш-памятью**, часто используется конструкторами, которые стараются скрыть латентность промахов при обращениях к кэш-памяти путем использования процессоров, не придерживающихся порядка выполнения инструкций. В них реализованы две разновидности неблокировки. *Попадание в процессе обработки промаха* раз-

Неблокирующаяся кэш-память
Кэш-память, позволяющая процессору обращаться к кэш-памяти в процессе обработки ранее возникшего промаха

решает дополнительные попадания при обращении к кэш-памяти, а *промах в процессе обработки промаха* позволяет иметь несколько необработанных промахов. Цель первой из них — скрыть часть латентности промаха другой работой, а цель второй — наложить друг на друга латентности двух разных промахов.

Если значительная доля промахов накладываются друг на друга, потребуется система памяти с высокой пропускной способностью, которая может параллельно обработать сразу несколько промахов. В настольных системах память может быть способна получить лишь ограниченные преимущества от этой возможности, но большие серверные системы и мультипроцессоры зачастую обладают системами памяти, способными параллельно обработать более одного необработанного промаха.

Оба микропроцессора осуществляют упреждающую выборку инструкций и имеют встроенный аппаратный механизм упреждающей выборки для доступа к данным. Они смотрят на шаблон промахов при обращении к данным и используют эту информацию, чтобы попытаться предсказать следующий адрес для начала извлечения данных перед тем, как произойдет промах. Такие технологии особенно хорошо работают при доступе в цикле к элементам массивов.

Существенной трудностью, с которой сталкиваются разработчики кэш-памяти, является поддержка таких процессоров, как Nehalem и Opteron X4, которые могут выполнять более одной инструкции, обращающейся к памяти за один тактовый цикл. Несколько запросов могут быть поддержаны в кэш-памяти первого уровня с помощью двух разных технологий. У кэш-памяти может быть несколько портов, что позволит одновременно осуществлять более одного обращения к одному и тому же блоку кэш-памяти. Но многопортовые устройства кэш-памяти зачастую обходятся слишком дорого, поскольку RAM-ячейки в многопортовой памяти должны быть намного больше, чем однопортовые ячейки. Альтернативная схема заключается в разбиении кэш-памяти на банки и разрешении множественных, независимых обращений, если они осуществляются к разным банкам. Эта технология похожа на оперативную память с чередованием адресов (см. рис. 5.10). Кэш-память данных уровня L1 процессора Opteron X4 поддерживает два 128-разрядных чтения за один тактовый цикл и имеет восемь банков.

Nehalem и большинство других процессоров в своей иерархии памяти следуют политике *включения*. Это означает, что копия всех данных в кэш-памяти самого верхнего уровня может быть также найдена в устройствах кэш-памяти нижних уровней. В отличие от этого, процессоры AMD в устройствах кэш-памяти первого и второго уровней следуют политике *исключения*, означающей, что блок кэш-памяти может быть найден только в устройствах кэш-памяти первого или второго уровней, но не на обоих уровнях. Следовательно, при промахе во время обращения к кэш-памяти уровня L1, когда блок извлекается из L2 в L1, заменяемый блок отправляется обратно в кэш-память уровня L2.

Сложные иерархии памяти этих микропроцессоров и большая доля кристаллов, выделенная под устройства кэш-памяти и буферы TLB, свидетельствуют о том, что существенные усилия разработчиков были направлены на ликвидацию разрыва

между длительностью тактового цикла процессоров и временем ожидания при обращении к памяти.

Уточнение. Совместно используемая кэш-память уровня L3 процессора Opteron X4 не всегда следует политике исключения. Поскольку блоки данных в кэш-памяти уровня L3 могут совместно использоваться несколькими процессорами, удаление блока кэш-памяти из L3 происходит только в том случае, если он не используется совместно другими процессорами. Следовательно, протокол кэш-памяти уровня L3 распознает, является ли блок кэш-памяти совместно используемым или используется только одним процессором.

Уточнение. Процессор Opteron X4 не только не следует политике обычного включения, но кроме этого еще имеет нестандартные взаимоотношения между уровнями иерархии памяти. Вместо того чтобы снабжать из памяти кэш-память уровня L2, которая в свою очередь снабжает кэш-память уровня L1, кэш-память уровня L2 содержит только те данные, которые были исключены из кэш-памяти уровня L1. Таким образом, кэш-память уровня L2 может быть названа *кэш-памятью «жертвенных» данных*, поскольку в ней содержатся только те блоки, которые были перемещены из L1 («жертвы»). Аналогично этому, кэш-память уровня L3 является кэш-памятью «жертвенных» данных для L2, состоящей только из блоков, которые были перемещены из L2. Если данные, приведшие к промаху в L1, не будут найдены в L2, но будут найдены в L3, кэш-память уровня L3 поставит данные непосредственно кэш-памяти уровня L1. Следовательно, промах при обращении к L1 может быть обслужен путем попадания в L2, или путем попадания в L3, или из памяти.

5.10. Заблуждения и недоразумения

Как один из наиболее естественных количественных аспектов компьютерной архитектуры, иерархия памяти может показаться наименее уязвимой в плане заблуждений и недоразумений. Мало того, что существует масса весьма распространенных заблуждений и недоразумений, так еще и некоторые из них приводят к существенным негативным последствиям. Начнем с заблуждения, в котором довольно часто пребывают студенты при решении упражнений и сдаче экзаменов.

Недоразумение. *При имитации кэш-памяти упускается из виду байтовая адресация или размер блока кэш-памяти.*

При имитации кэш-памяти (с использованием или без использования компьютера) нужно убедиться в том, что при определении, на какой блок кэш-памяти отображается заданный адрес, в расчет берется эффект байтовой адресации и факт наличия в блоке нескольких слов. Например, если мы располагаем 32-разрядной кэш-памятью с непосредственным отображением и размером блока, равном 4 байтам, байтовый адрес 36 отображается на блок 1 кэш-памяти, поскольку байтовый адрес 36 — это блоковый адрес 9, а (9 mod 8) = 1.

С другой стороны, если адрес 36 это адрес слова, то тогда он отображается на блок (36 mod 8) = 4. Убедитесь в том, что в задаче четко определена база адреса.

Подобным же образом мы должны брать в расчет размер блока. Предположим, у нас есть кэш-память, состоящая из 256 байт, а размер блока составляет 32 байта. В какой блок попадет байтовый адрес 300? Если мы разобьем адрес 300 на поля, будет виден ответ:

Байтовый адрес 300 — это адрес блока

$$\left\lfloor \frac{300}{32} \right\rfloor = 9$$

Количество блоков в кэш-памяти

$$\left\lfloor \frac{256}{32} \right\rfloor = 8$$

Номер блока 9 приходится на блок кэш-памяти номер (9 mod 8) = 1.

В эту ловушку попадаются многие люди, включая авторов (при первоначальных прикидках) и преподавателей, забывших о том, какие адреса подразумеваются в словах, байтах или номерах блоков. Не забудьте об этом заблуждении при решении упражнений.

Недоразумение. *Игнорирование поведения системы памяти при написании программ или при генерации кода в компиляторе.*

Это легко переформулировать: «Программисты могут игнорировать иерархии памяти при написании кода». Проиллюстрируем это с помощью примера, использующего перемножение матриц для завершения сравнения при сортировке, показанной на рис. 5.15.

Вот как выглядит внутренний цикл в версии перемножения матриц из главы 3:

```
for (i=0; i!=500; i=i+1)
   for (j=0; j!=500; j=j+1)
      for (k=0; k!=500; k=k+1)
         x[i][j] = x[i][j] + y[i][k] * z[k][j];
```

При запуске с входными данными, представленными матрицами 500 × 500 с двойной точностью, время выполнения процессором показанного выше цикла на MIPS-процессоре, имеющем 1-мегабайтную вторичную кэш-память, будет примерно вдвое больше по сравнению с тем, когда порядок цикла будет изменен на k, j, i (то есть i станет самым внутренним циклом)! Единственное отличие состоит в том, как программа обращается к памяти и стремится получить эффект от иерархии памяти. Дальнейшая оптимизация компилятора с использованием технологии под

названием *объединение в блоки* может отразиться на времени выполнения, которое уменьшится для этого кода еще в четыре раза!

Заблуждение. *Наличие меньшей степени ассоциативности в совместно используемой кэш-памяти, чем количество ядер или потоков, использующих эту кэш-память.*

Если не принять дополнительные меры, программа, параллельно запущенная на 2^n процессорах или потоках, может запросто распределить структуры данных по адресам, которые будут отображаться на один и тот же набор совместно используемой кэш-памяти уровня L2. Если кэш-память имеет как минимум 2^n-канальную ассоциативность, то этот случайный конфликт будет скрыт от программы оборудованием. Если нет, программист может столкнуться с явно мистической ошибкой выполнения, возникающей фактически из-за конфликта промахов при обращении к L2, при переходе, скажем, с 16-ядерной конструкции на 32-ядерную, при условии, что в обеих конструкциях используется кэш-память уровня L2 с 16-канальной ассоциативностью.

Заблуждение. *Использование среднего времени доступа к памяти для оценки иерархии памяти на процессоре, не придерживающемся порядка выполнения инструкций.*

Если процессор останавливается на время обработки промаха при обращении к кэш-памяти, то вы можете отдельно вычислить время задержки из-за обращения к памяти и время выполнения программы процессором, а следовательно, независимо оценить иерархию памяти, используя среднее время доступа к памяти (см. упражнение «Вычисление среднего времени доступа к памяти»).

Если процессор продолжает выполнение инструкций даже во время обработки промаха, то можно столкнуться с новыми промахами при обращении к кэш-памяти. Единственная точная оценка иерархии памяти — имитация работы в этой иерархии процессора, не придерживающегося порядка выполнения инструкций.

Заблуждение. *Расширение адресного пространства за счет добавления сегментов в верхнюю часть несегментированного адресного пространства.*

В течение 1970-х годов многие программы разрослись до таких размеров, что весь код и данные не могли быть адресованы с помощью всего лишь 16-разрядных адресов. После чего компьютеры были переработаны под 32-разрядные адреса либо посредством несегментированного 32-разрядного адресного пространства (также называемого *линейным адресным пространством*), либо посредством добавления 16 разрядов сегмента к существующему 16-разрядному адресу. С точки зрения маркетинга, добавление сегментов, видимых программой, которые заставляли программиста и компилятор разбивать программу на сегменты, могло решить проблему адресации. К сожалению, всякий раз, когда программа хотела адресовать то, что было больше одного сегмента, например индексы для больших массивов, неограниченные указатели или параметры ссылки, возникала проблема. Кроме этого, добавление сегментов может превратить любой адрес в два слова — одно для номера сегмента, а другое для смещения сегмента, — вызывая проблемы использования адресов в регистрах.

Заблуждение. *Реализация диспетчера виртуальной машины на архитектуре набора команд, которая была разработана без учета виртуализации.*

Многие разработчики 1970-х—1980-х годов не позаботились о гарантиях привилегированности всех инструкций, считывающих или записывающих информацию, связанную с аппаратными ресурсами. Такая беспечность привела к проблемам для диспетчеров виртуальных машин на базе всех этих архитектур, включая x86, которая использовалась нами в качестве примера.

В табл. 5.14 дается описание 18 инструкций, вызывающих проблемы для виртуализации (Robin and Irvine, 2000). Они относятся к двум распространенным классам инструкций:

- чтение регистров управления в пользовательском режиме, которые показывают, что гостевая операционная система запущена на виртуальной машине (например, упомянутая ранее инструкция POPF);
- проверка защиты, как этого требует сегментированная архитектура, в предположении, что операционная система запущена на самом высоком привилегированном уровне.

Таблица 5.14. Сводка из 18 x86-инструкций, вызывающих проблемы для виртуализации (Robin and Irvine, 2000). Первые пять инструкций в верхней части таблицы позволяют программе в пользовательском режиме читать значение регистра управления, например, регистров таблицы дескрипторов, не вызывая при этом перехвата. Инструкция извлечения флагов из стека изменяет регистр управления, содержащий важную информацию, но при этом приводит в пользовательском режиме к молчаливому сбою. Проверка защиты сегментированной архитектуры x86 является западней, расставленной группой инструкций, показанных в нижней части таблицы, поскольку каждая из этих инструкций проверяет уровень привилегированности косвенным образом, как часть выполнения инструкции при чтении регистра управления. Проверка предполагает, что операционная система должна быть на самом высоком привилегированном уровне, что не соответствует случаю использования гостевых виртуальных машин. Только лишь инструкция перемещения в регистр сегмента пытается модифицировать состояние управления, но проверка защиты точно так же мешает ей это сделать

Категория проблемы	Проблемные инструкции x86
Доступ к важным регистрам без перехвата при запуске в пользовательском режиме	Сохранение регистра глобальной таблицы дескрипторов (SGDT) Сохранение регистра локальной таблицы дескрипторов (SLDT) Сохранение регистра таблицы дескрипторов прерываний (SIDT) Сохранение слова состояния машины (SMSW) Помещение флагов в стек (PUSHF, PUSHFD) Извлечение флагов из стека (POPF, POPFD)

Категория проблемы	Проблемные инструкции x86
При обращении к механизмам виртуальной памяти в пользовательском режиме инструкции дают сбой из-за проверок защиты, используемых в архитектуре x86	Загрузка прав доступа из дескриптора сегмента (LAR) Загрузка границы сегмента из дескриптора сегмента (LSL) Проверка возможности чтения дескриптора сегмента (VERR) Проверка возможности записи дескриптора сегмента (VERW) Извлечение из стека в регистр сегмента (POP CS, POP SS, ...) Помещение в стек регистра сегмента (PUSH CS, PUSH SS, ...) Дальний вызов к другому уровню привилегии (CALL) Дальний возврат к другому уровню привилегии (RET) Дальний переход к другому уровню привилегии (JMP) Программное прерывание (INT) Сохранение регистра селектора сегмента (STR) Перемещение в регистры сегмента или из этих регистров (MOVE)

Чтобы упростить реализацию диспетчеров виртуальных машин на x86, и AMD и Intel предложили расширения архитектуры посредством введения нового режима. VT-x предоставляет новый режим выполнения для запущенных виртуальных машин, спланированное определение состояния виртуальной машины, инструкции для быстрого переключения виртуальных машин и большой набор параметров для выбора обстоятельств, при которых вызывается диспетчер виртуальных машин. Всего в режим VT-x добавлено 11 новых инструкций для x86. Такие же предложения были сделаны в технологии AMD Pacifica.

Альтернативой изменению оборудования является небольшая модификация операционной системы с целью избежать использования проблемных частей архитектуры. Такая технология называется *паравиртуализацией*, хорошим примером которой может послужить диспетчер с открытым кодом Xen VMM. Он предоставляет гостевой операционной системе абстракцию виртуальной машины, которая использует только пригодные для виртуализации части физического оборудования x86, на которых работает диспетчер виртуальных машин.

5.11. Заключительные комментарии

Сложность создания системы памяти заключается в том, чтобы она соответствовала более быстродействующим процессорам, ведь исходный материал для оперативной памяти, DRAM-элементов по существу такой же, как в самых быстродействующих компьютерах, так и в самых медленных и дешевых.

Существует принцип локальности, который дает нам шанс преодолеть продолжительную латентность доступа к памяти — и обоснованность этой стратегии демонстрируется на всех уровнях иерархии памяти. Хотя эти уровни иерархии в количественных понятиях выглядят сильно отличающимися друг от друга, в сво-

ей работе они следуют сходным стратегиям и используют одни и те же свойства локальности.

Многоуровневые устройства кэш-памяти позволяют легче использовать дополнительную оптимизацию кэш-памяти благодаря двум обстоятельствам. Во-первых, параметры конструирования кэш-памяти нижнего уровня отличаются от таковых для кэш-памяти первого уровня. Например, поскольку кэш-память нижнего уровня будет иметь значительно больший размер, появляется возможность использования более крупных блоков. Во-вторых, кэш-память нижнего уровня не используется процессором на постоянной основе, в отличие от кэш-памяти первого уровня. Это позволяет нам рассмотреть вопрос задействования низкоуровневой кэш-памяти для какой-нибудь другой работы в период невостребованности, что может быть полезным для предупреждения будущих промахов.

Другой тенденцией является поиск помощи от программного обеспечения. Эффективное управление иерархией памяти с использованием разнообразного преобразования программ и аппаратных средств является главной целью улучшения компиляторов. Исследуются две разные идеи. Одна из них заключается в реорганизации программы, чтобы увеличить ее пространственную и временную локальность. Этот подход фокусируется на циклично-ориентированных программах, использующих в качестве основной структуры данных большие массивы; в качестве типичного примера можно привести задачи линейной алгебры. Путем реструктуризации циклов, обращающихся к массивам, существенно улучшается локальность, а следовательно, может быть достигнута высокая производительность кэш-памяти. Рассмотрение в предыдущем разделе недоразумения, касающегося игнорирования поведения системы памяти при написании программ или при генерации кода в компиляторе, показывает, насколько эффективным может быть даже простое изменение структуры цикла.

Еще одним подходом является **предварительная выборка (предвыборка)**, при которой блок данных переносится в кэш-память еще до того, как к нему последует обращение. Многие микропроцессоры используют аппаратную предвыборку, пытаясь предсказать обращения, которые может быть трудно заметить при использовании программных средств.

Третий подход заключается в использовании специальных инструкций, связанных с кэш-памятью, которые оптимизируют обмен данными с памятью. Например, микропроцессор из раздела 7.10 главы 7 использует оптимизацию: при промахе во время записи содержимое блока из памяти не извлекается, потому что программа собирается записать весь блок. Эта оптимизация существенно сокращает для одного ядра объем обмена данными с памятью.

В главе 7 будет показано, что системы памяти являются центральным вопросом конструирования для параллельных процессоров. Возрастающая важность иерархии памяти в определении производительности системы означает, что эта важная область будет по-прежнему находиться в фокусе как разработчиков, так и исследователей еще несколько лет.

Предвыборка
Технология, в которой блоки данных, которые понадобятся в будущем, переносятся в кэш-память заранее с использованием специальных инструкций, определяющих адрес блока.

5.12. Упражнения

Предоставлены Цзичуань Чангом (Jichuan Chang), Джакобом Леверичем (Jacob Leverich), Кевином Лимом (Kevin Lim) и Партасарати Ранганатаном (Parthasaratny Ranganathan), все они являются представителями компании Hewlett-Packard.

Упражнение 5.1

В данном упражнении будут рассмотрены иерархии памяти для различных приложений, перечисленных в следующей таблице.

а	Просмотр веб-страниц
б	Интерактивные банковские услуги

5.1.1 [10] <5.1> При условии, что в процесс вовлечены и клиент и сервер, сначала назовите системы, используемые клиентом и сервером. Куда могут быть помещены устройства кэш-памяти для ускорения процесса?

5.1.2 [10] <5.1> Разработайте иерархию памяти для системы. Покажите типичный размер и латентность различных уровней иерархии. Какова взаимосвязь между размером кэш-памяти и латентностью доступа?

5.1.3 [15] <5.1> Что собой представляют элементы переноса данных между уровнями иерархии? Какая связь между местом расположения данных, размером данных и латентностью переноса?

5.1.4 [10] <5.1, 5.2> Ширина полосы пропускания канала связи и полоса пропускания при обработке данных на сервере являются двумя важными факторами, которые берутся в расчет при разработке иерархии памяти. Чья полоса пропускания может стать здесь ограничивающим фактором? Как улучшить этот показатель, и какой ценой?

5.1.5 [5] <5.1, 5.8> Теперь рассмотрим ситуацию, при которой к серверу обращаются сразу несколько клиентов. Улучшит ли такое развитие событий пространственную и временную локальность?

5.1.6 [10] <5.1, 5.8> Приведите пример, когда кэш-память может предоставить устаревшие данные. Как подавить или исключить подобные случаи?

Упражнение 5.2

В данном упражнении будут рассмотрены свойства локальности памяти при вычислении матриц. Следующий код написан на языке C, где элементы в одной и той же строке сохранены рядом друг с другом.

а	`for (I=0; I<8000; I++)`
	`  for (J=0; J<8; J++)`
	`  A[I][J]=B[J][0]+A[J][I];`

б	`for (J=0; J<8; J++)`
	`  for (I=0; I<8000; I++)`
	`  A[I][J]=B[J][0]+A[J][I];`

5.2.1 [5] <5.1> Сколько 32-разрядных целых чисел может быть сохранено в 16-байтовой строке кэш-памяти?

5.2.2 [5] <5.1> Обращение к каким переменным демонстрирует временную локальность?

5.2.3 [5] <5.1> Обращение к каким переменным демонстрирует пространственную локальность?

Локальность зависит как от порядка обращения, так и от расположения данных. Те же самые вычисления могут также быть написаны на Matlab, как показано ниже, который отличается от C последовательным сохранением элементов матрицы, принадлежащих одному и тому же столбцу.

а	`for I=1:8000`
	`  for J=1:8`
	`  A(I,J)=B(J,0)+A(J,I);`
	`  end`
	`end`

б	`for J=1:8`
	`  for I=1:8000`
	`  A(I,J) = B(J,0) + A(J,I);`
	`  end`
	`end`

5.2.4 [10] <5.1> Сколько 16-байтных строк кэш-памяти понадобится для сохранения всех 32-разрядных элементов матрицы, к которым происходило обращение?

5.2.5 [5] <5.1> Обращение к каким переменным демонстрирует временную локальность?

5.2.6 [5] <5.1> Обращение к каким переменным демонстрирует пространственную локальность?

Упражнение 5.3

Устройства кэш-памяти играют важную роль в достижении высокой производительности иерархии памяти при работе с процессорами. Ниже показан список 32-разрядных адресных ссылок, заданных в виде адресов слов.

а	1, 134, 212, 1, 135, 213, 162, 161, 2, 44, 41, 221
б	6, 214, 175, 214, 6, 84, 65, 174, 64, 105, 85, 215

5.3.1 [10] <5.2> Определите для каждой из этих ссылок двоичный адрес, тег и индекс при условии использования кэш-памяти с непосредственным отображением, имеющей 16 блоков по одному слову в каждом. Также перечислите попадание или промах каждой ссылки, при условии, что изначально кэш-память была пустой.

5.3.2 [10] <5.2> Определите для каждой из этих ссылок двоичный адрес, тег и индекс при условии использования кэш-памяти с непосредственным отображением, имеющей блоки, состоящие из двух слов и общим размером в восемь блоков.

также перечислите попадание или промах каждой ссылки, при условии, что изначально кэш-память была пустой.

5.3.3 [20] <5.2, 5.3> Вас попросили оптимизировать конструкцию кэш-памяти для заданных ссылок. Возможно создание трех конструкций кэш-памяти с непосредственным отображением, каждая из которых имеет общий размер из восьми слов данных: C1 имеет блоки размером в одно слово, C2 имеет блоки размером в два слова, а C3 имеет блоки размером в четыре слова. Какая конструкция кэш-памяти будет наилучшей в понятиях коэффициента промахов? Если время задержки при возникновении промаха равно 25 циклам и время доступа к C1 составляет 1 цикла, к C2 — 3 цикла, а к C3 — 5 циклов, какая из конструкций кэш-памяти будет наилучшей?

Существует множество разных параметров конструкций, играющих важную роль в общей производительности кэш-памяти. В показанной ниже таблице приведен список параметров для разных конструкций кэш-памяти с непосредственным отображением.

	Размер данных кэш-памяти, Кбайт	Размер блока кэш-памяти, слов	Время доступа к кэш-памяти, циклов
а	64	1	1
б	64	2	2

5.3.4 [15] <5.2> Вычислите общее количество разрядов, необходимое для устройств кэш-памяти, перечисленных в таблице при условии использования 32-разрядных адресов. Используя приведенный общий размер, определите общий размер ближайшей кэш-памяти с непосредственным отображением, имеющей блоки, состоящие из 16 слов такого же или большего размера. Объясните, почему вторая кэш-память, несмотря на больший размер своих данных, может предоставить производительность ниже, чем у первичной кэш-памяти.

5.3.5 [20] <5.2, 5.3> Сгенерируйте серию запросов на чтение, имеющих меньший коэффициент промахов при использовании 2-килобайтной кэш-памяти с двухканальной ассоциативностью, чем у устройств кэш-памяти, перечисленных в таблице. Покажите одно из возможных решений, которое сделает устройства кэш-памяти, перечисленные в таблице, имеющими такой же или более низкий коэффициент промахов, чем у кэш-памяти размером 2 Кбайт. Расскажите о достоинствах и недостатках этого решения.

5.3.6 [15] <5.2> Формула, показанная в начале раздела «Основы кэш-памяти», демонстрирует типичный метод индексации кэш-памяти с непосредственным отображением, а именно (Адрес блока) mod (Количество блоков в кэш-памяти). Предполагая использование 32-разрядной адресации и 1024 блоков в кэш-памяти, рассмотрите другую функцию индексации: (Адрес блока [31:27]) XOR (Адрес блока [26:22]). Возможно ли использование этой функции для индексации кэш-памяти с непосредственным отображением? Если да, то объясните почему и рассмотрите любые изменения, которые может понадобиться внести в кэш-память. Если это невозможно, то объясните почему.

Упражнение 5.4

Для конструкции кэш-памяти с непосредственным отображением, имеющей 32-разрядную адресацию, при обращении к ней используются следующие разряды адреса.

	Тег	Индекс	Смещение
а	31–10	9–4	3–0
б	31–12	11–15	4–0

5.4.1 [5] <5.2> Каков размер строки кэш-памяти (в словах)?

5.4.2 [5] <5.2> Сколько элементов содержит кэш-память?

5.4.3 [5] <5.2> Каково соотношение между общим количеством разрядов, необходимым для реализации такой кэш-памяти и разрядов, используемых для хранения данных?

Начиная с включения питания компьютера были записаны следующие обращения к кэш-памяти с байтовой адресацией.

Address	0	4	16	132	232	160	1024	30	140	3100	180	2180

5.4.4 [10] <5.2> Сколько блоков было заменено?

5.4.5 [10] <5.2> Каков коэффициент попаданий?

5.4.6 [20] <5.2> Дайте список финального состояния кэш-памяти, где каждый достоверный элемент представлен в виде записи <индекс, тег, данные >.

Упражнение 5.5

Вспомним, что у нас имеются две стратегии записи и две стратегии выделения памяти для записи, их комбинация может быть реализована либо в кэш-памяти уровня L1, либо в кэш-памяти уровня L2.

	L1	L2
а	Отложенная запись, выделение для записи	Сквозная запись, без выделения для записи
б	Отложенная запись, выделение для записи	Сквозная запись, выделение для записи

5.5.1 [5] <5.2, 5.5> Для снижения латентности доступа между разными уровнями иерархии памяти используются буферы. Перечислите для этой заданной конфигурации возможные буферы, необходимые между кэш-памятью уровня L1 и кэш-памятью уровня L2 и памятью.

5.5.2 [20] <5.2, 5.5> Дайте описание процедуры обработки промаха при обращении к кэш-памяти уровня L1, рассмотрите задействованные компоненты и возможность замены измененного блока.

5.5.3 [20] <5.2, 5.5>Для конфигурации многоуровневой исключающей кэш-памяти (блок может находиться только в одной кэш-памяти, либо в L1, либо

в L2) опишите процедуру обработки промаха при записи в кэш-память уровня L1, рассматривая задействованные компоненты и возможность замены измененного блока.

Рассмотрите следующую программу и поведение кэш-памяти.

	Чтение данных на 1000 инструкций	Запись данных на 1000 инструкций	Коэффициент промахов при обращении к кэш-памяти инструкций	Коэффициент промахов при обращении к кэш-памяти данных	Размер блока в кэш-памяти данных (в байтах)
а	200	160	0,20%	2%	8
б	180	120	0,20%	2%	16

5.5.4 [5] <5.2, 5.5> Какова минимальная полоса пропускания при чтении и записи для кэш-памяти со сквозной записью при выделении для записи (измеренная в байтах на цикл), необходимая для достижения показателя CPI, равного 2?

5.5.5 [5] <5.2, 5.5> Каковы минимальные полосы пропускания при чтении и записи для кэш-памяти с отложенной записью при выделении для записи, необходимые для достижения показателя CPI, равного 2 при условии, что 30% заменяемых блоков кэш-памяти являются измененными?

5.5.6 [5] <5.2, 5.5> Какова минимальная пропускная способность, необходимая для достижения производительности с показателем CPI = 1,5?

Упражнение 5.6

Медиаприложения, проигрывающие аудио или видеофайлы, являются частью класса рабочих нагрузок, который называется «потоковой» рабочей нагрузкой; то есть они приносят большое количество данных, но основную часть этих данных повторно не используют. Рассмотрите рабочую нагрузку, связанную с видеопотоком, которая последовательно обращается к рабочему набору объемом 512 Кбайт с помощью следующего потока адресов:

```
0, 4, 8, 12, 16, 20, 24, 28, 32, ...
```

5.6.1 [5] <5.5, 5.3> Предположим, что используется кэш-память с непосредственным отображением, имеющая размер 64 Кбайт и 32-разрядную строку. Каким будет коэффициент промахов для показанного выше адресного потока? Насколько этот коэффициент промахов зависит от размера кэш-памяти или рабочего набора? Как можно охарактеризовать промахи, испытываемые данной рабочей нагрузкой, на основе модели трех «С»?

5.6.2 [5] <5.5, 5.1> Вычислите коэффициент промахов для кэш-памяти с размером строки 16 байт, 64 байта и 128 байт. Какую разновидность локальности использует эта рабочая нагрузка?

5.6.3 [10] <5.10> «Предвыборка» — это технология, которая использует шаблоны предсказуемых адресов для эмпирического введения дополнительных

кэш-строк, когда происходит обращение к конкретной строке кэш-памяти. Одним из примеров предвыборки является потоковый буфер, который осуществляет предвыборку последовательно примыкающих кэш-строк в отдельный буфер при вводе конкретной кэш-строки. Если данные находятся в буфере предвыборки, это считается попаданием, и данные перемещаются в кэш-память, после чего проводится предвыборка следующей строки кэш-памяти. Предположим, что используется потоковый буфер с двумя входами и что латентность кэш-памяти имеет значение, позволяющее кэш-строке быть загруженной до того, как завершится обработка предыдущей кэш-строки. Каким будет коэффициент промахов для показанного выше потока данных?

Размер блока кэш-памяти (B) может повлиять как на коэффициент промахов, так и на латентность промаха. Подберите оптимальный размер блока кэш-памяти при следующих коэффициентах промахов, показанных ниже в таблице, для разных размеров блоков при условии, что используется машина с показателем CPI, равным единице, и средним количеством обращений (как к инструкциям, так и к данным) 1,35 на инструкцию.

	8	16	32	64	128
а	8%	3%	1,8%	1,5%	2%
б	4%	4%	3%	1,5%	2%

5.6.4 [10] <5.2> Каков оптимальный размер блока для латентности промаха 20 × B циклов?

5.6.5 [10] <5.2> Каков оптимальный размер блока для латентности промаха 24 + B циклов?

5.6.6 [10] <5.2> Каков оптимальный размер блока для постоянного значения латентности промаха?

Упражнение 5.7

В данном упражнении будут рассмотрены разные способы влияния объема на общую производительность. В общем, время доступа к кэш-памяти пропорционально объему. Предположим, что операция доступа к оперативной памяти занимает 70 нс и эти операции доступа к памяти составляют 36% всех инструкций. В следующей таблице показаны данные для устройств кэш-памяти уровня L1, подключенных к двум процессорам, P1 и P2.

		Размер кэш-памяти L1, Кбайт	Коэффициент промахов при обращении к L1, %	Время попадания при обращении к L1, нс
а	P1	1	11,4	0,62
	P2	2	8,0	0,66
б	P1	8	4,3	0,96
	P2	16	3,4	1,08

5.7.1 [5] <5.3> Какими соответственно будут тактовые частоты процессоров P1 и P2 при условии, что время попадания для устройств кэш-памяти L1 определяется продолжительностями тактовых циклов этих процессоров?

5.7.2 [5] <5.3> Каково среднее время доступа к данным (AMAT) для каждого из процессоров P1 и P2?

5.7.3 [5] <5.3> Каковы будут общие показатели CPI для каждого из процессоров P1 и P2 при условии, что базовый показатель CPI равен 1,0? Какой из процессоров будет работать быстрее?

В следующих трех задачах будет рассмотрено добавление к процессору P1 кэш-памяти уровня L2, чтобы, по-видимому, восполнить ограниченный объем его кэш-памяти уровня L1. При решении этих задач используйте объемы устройств кэш-памяти L1 и показатели времени попадания из предыдущей таблицы. Показанный коэффициент промахов при обращении к кэш-памяти L2 является его локальным коэффициентом промахов.

	Размер кэш-памяти L2	Коэффициент промахов при обращении к L2, %	Время попадания при обращении к L2, нс
а	512 Кбайт	98	3,22
б	4 Мбайт	73	11,48

5.7.4 [10] <5.3> Каков показатель AMAT для P1 при дополнении его кэш-памятью L2? Улучшился или ухудшился показатель AMAT при использовании кэш-памяти L2?

5.7.5 [5] <5.3> Каков общий показатель CPI для P1 при дополнении его кэш-памятью L2 при базовом CPI, равном 1,0?

5.7.6 [10] <5.3> Какой из процессоров быстрее, при условии, что теперь у P1 имеется кэш-память L2? Если быстрее P1, какой коэффициент промахов нужен P2 при обращении к его кэш-памяти L1, чтобы сравняться по производительности с P1? Если быстрее P2, какой коэффициент промахов нужен P1 при обращении к его кэш-памяти L1, чтобы сравняться по производительности с P2?

Упражнение 5.8

В этом упражнении рассматриваются результаты применения различных конструкций кэш-памяти, в частности дается сравнение кэш-памяти, использующей ассоциативность, с кэш-памятью из раздела 5.2, использующей непосредственное отображение. При решении данных задач следует обращаться к таблице адресных потоков, показанной в упражнении 5.3.

5.8.1 [10] <5.3> Используя ссылки из упражнения 5.3, покажите окончательное содержимое трехканальной ассоциативной памяти, имеющей блоки, состоящие из двух слов и общий размер 24 слова. Используйте алгоритм замены LRU. Для каждой ссылки покажите разряды индекса, тега и смещения блока, а также определите событие попадания или промаха.

5.8.2 [10] <5.3> Используя ссылки из упражнения 5.3, покажите окончательное содержимое для полностью ассоциативной кэш-памяти, имеющей блоки, состоя-

щие из одного слова и общий размер восемь слов. Используйте алгоритм замены LRU. Для каждой ссылки покажите разряды индекса, тега и смещения блока, а также определите событие попадания или промаха.

5.8.3 [15] <5.3> Используя ссылки из упражнения 5.3, определите, каким будет коэффициент промахов для полностью ассоциативной кэш-памяти, имеющей блоки, состоящие из двух слов, и общий размер восемь слов, если используется алгоритм замены LRU. Каким будет коэффициент промахов при использовании алгоритма замены MRU (most recently used — последнего по времени использования)? И наконец, определите наилучший коэффициент промахов для этой кэш-памяти при использовании любой стратегии замены.

Многоуровневое кэширование является важной технологией преодоления ограничений объема, который может предоставить кэш-память первого уровня, при сохранении скорости этой кэш-памяти. Рассмотрим процессор, имеющий следующие параметры:

	Базовый показатель CPI при отсутствии задержек обращения к памяти	Скорость процессора, ГГц	Время доступа к оперативной памяти, нс	Коэффициент промахов на инструкцию при обращении к кэш-памяти первого уровня, %	Кэш-память второго уровня, скорость при непосредственном отображении, циклов	Глобальный коэффициент промахов при использовании кэш-памяти второго уровня с непосредственным отображением, %	Кэш-память второго уровня, скорость при восьмиканальной ассоциативности, циклов	Глобальный коэффициент промахов при использовании кэш-памяти второго уровня с восьмиканальной ассоциативностью, %
а	2,0	3	125	5	15	3,0	25	1,8
б	2,0	1	100	4	10	4,0	20	1,6

5.8.4 [10] <5.3> Вычислите показатель CPI для процессора из таблицы при использовании:

1) только кэш-памяти первого уровня;
2) кэш-памяти второго уровня с непосредственным отображением;
3) кэш-памяти второго уровня с восьмиканальной ассоциативностью.

Как изменятся эти показатели, если удвоится время обращения к оперативной памяти? Будут ли они снижены наполовину?

5.8.5 [10] <5.3> Иерархия кэш-памяти может состоять более чем из двух уровней. Имея показанный выше процессор с кэш-памятью второго уровня с непосредственным отображением, конструктор хочет добавить кэш-память третьего уровня, обращение к которой займет 50 циклов и уменьшит глобальный коэффициент про-

махов до 1,3%. Будет ли при этом обеспечена лучшая производительность? Какими, в целом, будут достоинства и недостатки добавления кэш-памяти третьего уровня?

5.8.6 [20] <5.3> На устаревших процессорах, таких как Intel Pentium или Alpha 21264, кэш-память второго уровня была внешней (расположенной на другом кристалле) по отношению к основному процессору и кэш-памяти первого уровня. Хотя такая конструкция позволяет иметь более крупную по объему кэш-память второго уровня, латентность доступа к кэш-памяти была существенно выше, а полоса пропускания обычно была уже, поскольку кэш-память второго уровня работает с более низкой тактовой частотой. Предположим, что кэш-память второго уровня объемом 512 Кбайт, расположенная на отдельном кристалле, имеет общий коэффициент промахов 4%. Насколько большой должна быть кэш-память, чтобы сравняться по производительности с показанной в таблице кэш-памятью второго уровня с непосредственным отображением, если каждые дополнительные 512 Кбайт кэш-памяти снижают общий коэффициент промахов на 0,7%, и кэш-память имеет общее время доступа, равное 50 циклам?

Упражнение 5.9

Для высокопроизводительных систем, например для таких, которые используют индексацию базы данных по принципу B-дерева, размер страницы определяется главным образом исходя из размера данных и производительности диска. Предположим, что индексная страница B-дерева при записях фиксированной длины заполнена на 70%. Полезность страницы определяется глубиной ее B-дерева, вычисляемой как $\log_2$(количество_записей). В следующей таблице, показанной для 16-байтных записей и 10-летнего старого диска с латентностью 10 мс и скоростью переноса данных 10 Мбайт/с, оптимальный размер страницы составляет 16 Кбайт.

Размер страницы, Кбайт	Полезность страницы, или глубина B-дерева, количество сохраненных доступов к диску	Стоимость доступа к индексной странице, мс	Соотношение полезности к стоимости
2	6,49 (или $\log_2(2048/16\times0,7)$)	10,2	0,64
4	7,49	10,4	0,72
8	8,49	10,8	0,79
16	9,49	11,6	0,82
32	10,49	13,2	0,79
64	11,49	16,4	0,70
128	12,49	22,8	0,55
256	13,49	35,6	0,38

5.9.1 [10] <5.4> Каким будет наилучший размер страницы, если теперь записи будут состоять из 128 байтов?

5.9.2 [10] <5.4> Каким будет наилучший размер страницы, если при таких же условиях, как в упражнении 5.9.1, страницы будут заполнены наполовину?

5.9.3 [20] <5.4> Каким будет наилучший размер страницы, если при таких же условиях, как в упражнении 5.9.2, использовать более современный диск с латентностью 3 мс и скоростью переноса данных 100 Мбайт/с? Объясните, почему будущим серверам будет лучше иметь более крупные страницы.

Сохранение в DRAM «наиболее часто востребованных» (или «горячих») страниц может уменьшить количество обращений к диску, но как при этом определить точное значение «наиболее часто востребованных» для заданной системы? Специалисты по обработке данных используют соотношение стоимости обращения к DRAM и к диску для определения порогового количества раз повторного использования горячих страниц. Стоимость обращения к диску равна стоимости диска, деленной на количество обращений в секунду, в то время как стоимость хранения страницы в DRAM равна стоимости мегабайта DRAM, деленной на размер страницы. Типичные цены на DRAM и диск и типичные размеры страниц базы данных для нескольких моментов времени приведены в следующей таблице.

Год	Стоимость DRAM, $/Мбайт	Размер страницы, Кбайт	Стоимость диска, $/диск	Коэффициент обращений к диску, обращений/с
1987	5000	1	15000	15
1997	15	8	2000	64
2007	0,05	64	80	83

5.9.4 [10] <5.1, 5.4> Каково пороговое количество раз повторного использования для этих трех поколений технологии?

5.9.5 [10] <5.4> Каково пороговое количество раз повторного использования, если мы продолжаем использовать один и тот же размер страницы, равный 4 Кбайт? Какая тенденция здесь наблюдается?

5.9.6 [20] <5.4> Какие другие факторы могут быть изменены для продолжения использования того же размера страниц (избегая при этом необходимости переписывания программного обеспечения)? Рассмотрите вероятность этих изменений при существующих технологиях и тенденциях изменения стоимости.

Упражнение 5.10

Согласно описаниям, приведенным в разделе 5.4, виртуальная память использует таблицы страниц для отслеживания отображения виртуальных адресов на физические адреса. В данном упражнении показано, как такая таблица может обновляться по мере обращения к адресам. В следующей таблице показан поток виртуальных адресов, подмеченный в системе. Предполагается, что используются страницы размером 4 Кбайт, полностью ассоциативный буфер TLB, имеющий четыре элемента, и полноценная LRU-замена. Если страницы должны быть доставлены с диска, происходит увеличение на единицу следующего наибольшего номера страницы.

а	4095, 31272, 15789, 15000, 7193, 4096, 8912
б	9452, 30964, 19136, 46502, 38110, 16653, 48480

Буфер TLB

Бит достоверности	Тег	Физический номер страницы
1	11	12
1	7	4
1	3	6
0	4	9

Таблица страниц

Бит достоверности	Физическая страница, или нахождение на диске
1	5
0	Диск
0	Диск
1	6
1	9
1	11
0	Диск
1	4
0	Диск
0	Диск
1	3
1	12

5.10.1 [10] <5.4> Используя адресный поток из таблицы и показанное исходное состояние TLB и таблицы страниц, покажите итоговое состояние системы. А также составьте список для каждого обращения с указанием, чем оно закончилось: попаданием при обращении к TLB, попаданием при обращении к таблице страниц или ошибкой отсутствия страницы.

5.10.2 [15] <5.4> Выполните упражнение 5.10.1 еще раз, но теперь используйте страницы размером не 4 Кбайт, а 16 Кбайт. В чем будут преимущества от использования более крупного размера страниц? А в чем будут заключаться недостатки?

5.10.3 [15] <5.3, 5.4> Покажите итоговое содержимое TLB при использовании двухканальной ассоциативности. Также покажите содержимое TLB при использовании непосредственного отображения. Обсудите важность использования TLB для достижения высокой производительности. Как можно обрабатывать обращения к виртуальной памяти без использования TLB?

Существует ряд параметров, влияющих на общий размер таблицы страниц. Ниже приведен ряд ключевых параметров таблицы страниц.

	Размер виртуального адреса, бит	Размер страницы, Кбайт	Размер записи таблицы страниц, байт
а	32	4	4
б	64	16	8

5.10.4 [5] <5.4> Используя параметры, показанные в таблице, вычислите общий размер таблицы страниц для системы, выполняющей пять приложений, которые используют половину всей доступной памяти.

5.10.5 [10] <5.4> Используя параметры, показанные в таблице, вычислите общий размер таблицы страниц для системы, выполняющей пять приложений, которые используют половину всей доступной памяти, при условии, что используется двухуровневая таблица страниц с 256 записями. Предположим, что каждая запись в основной таблице страниц имеет длину 6 байт. Вычислите минимальный и максимальный объем требуемой памяти.

5.10.6 [10] <5.4> Разработчик кэш-памяти хочет увеличить размер виртуально индексированного, имеющего физические теги устройства кэш-памяти, имеющей объем 4 Кбайт. Используя размеры страниц, перечисленные в таблице, определите, есть ли возможность создания кэш-памяти объемом 16 Кбайт с непосредственным отображением, при условии, что каждый блок состоит из двух слов. Насколько разработчик увеличил бы размер данных в кэш-памяти?

Упражнение 5.11

В данном упражнении будет изучена пространственно-временная оптимизация для таблицы страниц. В следующей таблице показаны параметры системы виртуальной памяти.

	Виртуальный адрес (разряды)	Установленная физическая DRAM-память, Гбайт	Размер страницы, Кбайт	Размер записей в таблице страниц, байт
а	32	4	8	4
б	64	16	4	8

5.11.1 [10] <5.4> Сколько записей необходимо иметь в одноуровневой таблице страниц? Сколько физической памяти необходимо для хранения таблицы страниц?

5.11.2 [10] <5.4> Использование многоуровневой таблицы страниц может сократить потребление таблицами страниц физической памяти за счет хранения в ней только лишь активных записей. Сколько уровней таблиц страниц понадобится в данном случае? И сколько обращений к памяти нужно для преобразования адреса, если при обращении к TLB произошел промах?

5.11.3 [15] <5.4> Для дальнейшей пространственно-временной оптимизации может быть использована инвертированная таблица страниц. Сколько записей должно быть сохранено в таблице страниц? Сколько в обычном и наихудшем варианте необходимо обращений к памяти для обслуживания промахов при обращениях к TLB, при условии реализации хэш-таблицы?

В следующей таблице показано содержимое буфера TLB с четырьмя элементами.

Идентификатор входа	Бит достоверности	Виртуальный адрес страницы	Бит изменения	Защита	Физический адрес страницы
1	1	140	1	RW	30
2	0	40	0	RX	34
3	1	200	1	RO	32
4	1	280	0	RW	31

5.11.4 [5] <5.4> При каком сценарии бит достоверности элемента 2 был бы установлен в нуль?

5.11.5 [5] <5.4> Что случится, когда инструкция будет вести запись в страницу с физическим адресом 30? Когда буфер TLB, управляемый программным способом, будет быстрее такого же буфера, управляемого аппаратным способом?

5.11.6 [5] <5.4> Что случится, когда инструкция будет вести запись в страницу с виртуальным адресом xxx?

Упражнение 5.12

В данном упражнении будет изучено влияние стратегии замены на коэффициент промахов. Предположим, что используется двухканальная кэш-память с четырьмя блоками. Возможно, для решения задач пригодится таблица, подобная той, что была показана в упражнении «Промахи и количество каналов ассоциативности в кэш-памяти» подраздела «Уменьшение коэффициента промахов в кэш-памяти за счет более гибкого размещения блоков» раздела 5.3, показанная ниже для адресной последовательности «0, 1, 2, 3, 4».

Адрес блока памяти, к которому осуществляется обращение	Попадание или промах	Заменяемый блок	Содержимое блоков кэш-памяти после обращения			
			Набор 0	Набор 0	Набор 1	Набор 1
0	Промах		Пам[0]			
1	Промах		Пам[0]		Пам[1]	
2	Промах		Пам[0]	Пам[2]	Пам[1]	
3	Промах		Пам[0]	Пам[2]	Пам[1]	Пам[3]
4	Промах	0	Пам[4]	Пам[2]	Пам[1]	Пам[3]
...						

В следующей таблице показана адресная последовательность.

	Адресная последовательность
а	0, 2, 4, 0, 2, 4, 0, 2, 4
б	0, 2, 4, 2, 0, 2, 4, 0, 2

5.12.1 [5] <5.3, 5.5> Сколько попаданий будет при этой адресной последовательности в случае использования LRU-замены?

5.12.2 [5] <5.3, 5.5> Сколько попаданий будет при этой адресной последовательности в случае использования MRU-замены (most recently used — последнего по времени использования)?

5.12.3 [5] <5.3, 5.5> Сымитируйте произвольную стратегию замены путем подбрасывания монеты. Например, «орел» будет означать замену первого блока, а «решка» — замену второго блока из набора. Сколько попаданий будет при данной адресной последовательности?

5.12.4 [10] <5.3, 5.5> По какому адресу должна проводиться замена для увеличения количества попаданий? Сколько попаданий будет при такой адресной последовательности, если следовать этой «оптимальной» стратегии?

5.12.5 [10] <5.3, 5.5> Объясните, почему возникают трудности при реализации такой стратегии замены в кэш-памяти, которая была бы оптимальной для всех адресных последовательностей.

5.12.6 [10] <5.3, 5.5> Предположим, что вы можете принять решение относительно каждого обращения к памяти, хотите вы или нет кэшировать запрошенный адрес. Какое влияние это может оказать на коэффициент промахов?

Упражнение 5.13

Для поддержки нескольких виртуальных машин необходима двухуровневая виртуализация памяти. Каждая виртуальная машина должна контролировать отображение виртуального адреса (VA) на физический адрес (PA), а гипервизор должен отображать физический адрес (PA) каждой виртуальной машины на действительный машинный адрес (MA). Для ускорения таких отображений программный метод, называемый теневой страничной организацией, дублирует таблицы страниц каждой виртуальной машины в гипервизоре и перехватывает изменения отображений VA на PA с целью поддержания целостности обеих копий. Чтобы избавиться от сложности теневых таблиц страниц, аппаратный метод, называемый вложенной таблицей страниц (или расширенной таблицей страниц), явным образом поддерживает два класса таблиц страниц (VA→PA и PA→MA) и может обходить эти таблицы исключительно аппаратно.

Рассмотрим следующую последовательность операций.

1) Создание процесса; 2) Промах при обращении к TLB; 3) Отсутствие страницы; 4) Переключение контекста.

5.13.1 [10] <5.4, 5.6> Что случится при заданной последовательности операций при использовании теневой таблицы страниц и вложенной таблицы страниц соответственно?

5.13.2 [10] <5.4, 5.6> Сколько обращений к памяти понадобится для обслуживания промаха при обращении к TLB при использовании обычной таблицы страниц в сравнении с вложенной таблицей страниц при условии использования

четырехуровневой таблицы страниц как в гостевой, так и во вложенной таблице страниц на машине x86?

5.13.3 [15] <5.4, 5.6> Какие из следующих показателей играют самую важную роль для теневой таблицы страниц: коэффициент промахов при обращении к TLB, латентность TLB-промаха, коэффициент ошибок отсутствия страницы и латентность обработки ошибок отсутствия страницы? Какие из показателей играют важную роль для вложенной таблицы страниц?

Параметры теневой системы организации страниц показаны в следующей таблице.

TLB-промахи на 1000, инструкций	Латентность TLB-промахов для вложенной таблицы страниц	Количество ошибок отсутствия страниц на 1000 инструкций	Издержки ошибки отсутствия страницы при теневой организации
0,2	200 циклов	0,001	30 000 циклов

5.13.4 [10] <5.6> Каков показатель CPI при использовании теневых таблиц страниц по сравнению с использованием вложенных таблиц страниц (если брать в расчет только издержки виртуализации таблицы страниц) для контрольной задачи, имеющей в обычных условиях показатель CPI, равный 1?

5.13.5 [10] <5.6> Какие технологии могут быть использованы для сокращения вынужденных издержек, возникающих при использовании теневых таблиц страниц?

5.13.6 [10] <5.6> Какие технологии могут быть использованы для сокращения вынужденных издержек, возникающих при использовании вложенных таблиц страниц?

Упражнение 5.14

Одно из самых больших препятствий для широкомасштабного использования виртуальных машин заключается в издержках производительности, связанных с запуском виртуальной машины. В показанной ниже таблице перечислены различные параметры производительности и поведение приложений.

	Базовый показатель CPI	Привилегированные обращения ОС на 10 000 инструкций	Влияние на производительность перехватов гостевой ОС, циклов	Влияние на производительность перехватов диспетчера виртуальных машин (VMM), циклов	Обращений на ввод-вывод данных на 10 000 инструкций	Время доступа к вводу-выводу (включая время на перехват гостевой ОС), циклов
а	2	100	20	150	20	1000
б	1,5	110	25	160	10	1000

5.14.1 [10] <5.6> Вычислите показатель CPI для вышеперечисленных систем, при условии отсутствия обращений к устройствам ввода-вывода. Каким будет показатель CPI при удвоении влияния на производительность VMM? При сокращении этого влияния вдвое? Каким будет наибольший показатель издержек на перехват управления VMM, если компания по производству программного обеспечения для виртуальных машин хочет получить 10%-ное снижение производительности?

5.14.2 [10] <5.6> Обращения к устройствам ввода-вывода зачастую оказывают большое влияние на общую производительность системы. Вычислите показатель CPI машины, используя показанные выше характеристики производительности при условии использования системы, не прошедшей виртуализацию. Вычислите показатель CPI еще раз, но теперь уже при использовании виртуализированной системы. Как эти показатели CPI изменяются, если количество обращений к устройствам ввода-вывода снижается в системе наполовину? Объясните, почему на приложения, связанные с вводом-выводом, виртуализация оказывает весьма незначительное влияние.

5.14.3 [30] <5.4, 5.6> Проведите сравнение и найдите отличия в идеях виртуальной памяти и виртуальных машин. Как можно сравнить цели каждой из этих идей? Какие достоинства и недостатки существуют у этих идей? Перечислите те случаи, при которых желательно применить виртуальную память, и те случаи, при которых желательно использовать виртуальные машины.

5.14.4 [20] <5.6> В разделе 5.6 рассматривается виртуализация на основе предположения, что виртуализированная система запускает ту же архитектуру набора инструкций, которая используется на исходном оборудовании. Но одним из возможных применений виртуализации является эмуляция архитектуры набора инструкций, не совпадающей с исходными архитектурами. В качестве примера может послужить QEMU, эмулирующая различные архитектуры, такие как MIPS, SPARC и PowerPC. Назовите ряд трудностей, связанных с подобной виртуализацией. Возможно ли такое, что эмулируемая система будет работать быстрее, чем на машине со своей родной архитектурой набора инструкций?

Упражнение 5.15

В данном упражнении будет использован блок управления для контроллера кэш-памяти для процессора с буфером записи. Для разработки собственных конечных автоматов в качестве отправной точки используйте конечный автомат, показанный на рис. 5.26. Предположим, что контроллер предназначен для простой кэш-памяти с непосредственным отображением, рассмотренной в подразделе «Простая кэш-память» раздела 5.7, но с добавлением буфера записи емкостью в один блок.

Следует напомнить, что буфер записи служит временным хранилищем, чтобы процессору не приходилось ждать двух обращений к памяти в случае промаха при обращении к измененному блоку. Вместо обратной записи измененного блока перед чтением нового блока он помещает измененный блок в буфер и тут же приступает к чтению нового блока. Затем измененный блок может быть записан в оперативную память, в то время когда процессор занят другой работой.

5.15.1 [10] <5.5, 5.7> Что должно произойти, если процессор выдал запрос, вызвавший *попадание* при обращении к кэш-памяти, в то время как блок записывался обратно в оперативную память из буфера памяти?

5.15.2 [10] <5.5, 5.7> Что должно произойти, если процессор выдал запрос, вызвавший *промах* при обращении к кэш-памяти, в то время как блок записывался обратно в оперативную память из буфера памяти?

5.15.3 [30] <5.5, 5.7> Разработайте конечный автомат, рассчитанный на использование буфера записи.

Упражнение 5.16

Поддержка целостности кэш-памяти касается того, как несколько процессоров видят заданный блок кэш-памяти. В следующей таблице показаны два процессора и их операции чтения-записи применительно к двум разным словам блока кэш-памяти X (изначально X[0] = X[1] = 0).

	P1	P2
а	X[0] ++; X[1] = 4;	X[0] = 2; X[1] ++;
б	X[0] ++; X[1] += 3;	X[0] = 5; X[1] -2;

5.16.1 [15] <5.8> Перечислите возможные значения заданного блока кэш-памяти для реализации подходящего протокола поддержки целостности данных кэш-памяти. Назовите как минимум еще одно возможное значение блока, если протокол не обеспечивает целостность данных кэш-памяти.

5.16.2 [15] <5.8> Укажите для протокола отслеживания допустимую последовательность операций для каждого набора «процессор – кэш-память» для завершения вышеперечисленных операций чтения-записи.

5.16.3 [10] <5.8> Какое количество промахов в лучшем и в худшем случае необходимо при обращении к кэш-памяти для завершения перечисленных инструкций чтения-записи.

Целостность памяти касается того, как выглядят несколько элементов данных. В следующей таблице показаны два процессора и их операции чтения-записи над разными блоками кэш-памяти (A и B изначально имеют значение 0).

	P1	P2
а	A = 1; B = 2; A++; B++; C	C = B; D = A;
б	A = 1; B + = 2; A++; B-4;	C = B; D = A;

5.16.4 [15] <5.8> Перечислите возможные значения C и D для реализации, гарантирующей соблюдение положений целостности данных, перечисленных в начале раздела 5.8.

5.16.5 [15] <5.8> Назовите как минимум еще одну возможную пару значений для C и D, если упомянутые положения не обеспечиваются.

5.16.6 [15] <5.2, 5.8> Какие комбинации упрощают реализацию протокола для разных комбинаций стратегий записи и стратегий выделения памяти в кэш для записи?

Упражнение 5.17

Микропроцессоры Barcelona и Nehalem относятся к микропроцессорам, реализованным на одном кристалле (chip multiprocessor — CMP), имеющим несколько ядер с их собственными устройствами кэш-памяти. Конструкция кэш-памяти уровня L2 в CMP использует интересные компромиссы. В следующих таблицах показаны коэффициенты промахов и латентности попадания для двух контрольных задач для сравнения конструкций с индивидуальными устройствами для каждого ядра и общим устройством кэш-памяти уровня L2. Предположим, что промах при обращении к кэш-памяти уровня L1 происходит один раз на каждые 32 инструкции.

	Индивидуальная, %	Общая, %
Коэффициент промахов на инструкцию при выполнении контрольной задачи A	0,30	0,12
Коэффициент промахов на инструкцию при выполнении контрольной задачи B	0,06	0,03

В следующей таблице показаны латентности попадания.

	Индивидуальная кэш-память	Общая кэш-память	Память
а	6	12	120
б	8	20	120

5.17.1 [15] <5.10> Какая из конструкций кэш-памяти больше подходит для каждой контрольной задачи? Воспользуйтесь данными для обоснования своих выводов.

5.17.2 [15] <5.10> Латентность общей кэш-памяти увеличивается вместе с размером CMP. Выберите наилучшую конструкцию, если латентность общей кэш-памяти L2 удвоилась. Пропускная способность устройств, находящихся вне кристалла, становится узким местом по мере увеличения количества ядер. Выберите наилучшую конструкцию, если время ожидания внешней памяти удваивается.

5.17.3 [10] <5.10> Рассмотрите все за и против общей и индивидуальной кэш-памяти уровня L2 для однопоточных, многопоточных и мультипрограммных рабочих нагрузок и пересмотрите их для случая использования внутренней кэш-памяти уровня L3.

5.17.4 [15] <5.10> Предположим, что базовый показатель CPI для обеих контрольных задач равен 1 (при идеальной кэш-памяти L2). Насколько возрастает производительность благодаря общей кэш-памяти уровня L2 при условии использования неблокирующейся кэш-памяти? Какого улучшения можно достичь за счет использования индивидуальной кэш-памяти L2?

5.17.5 [10] <5.10> Предположим, что в новых поколениях процессоров количество ядер удваивается каждые 18 месяцев. Насколько нужно будет повысить пропускную способность памяти, находящейся вне кристалла для процессора 2012 года, чтобы обеспечить такой же уровень производительности применительно к одному ядру?

5.17.6 [15] <5.10> Если рассматривать всю иерархию памяти, какие из способов оптимизации могут увеличить количество одновременных промахов?

Упражнение 5.18

В данном упражнении будет показано определение регистрационного журнала веб-сервера и изучена оптимизация кода для увеличения скорости обработки журнала. Структура данных для журнала определяется следующим образом:

```
struct entry {
  int  srcIP;              // удаленный IP-адрес
  char URL[128];           // запрашиваемый URL (например, «GET index.html»)
  long long refTime;       // время обращения
  int  status;             // статус соединения
  char browser[64];        // название браузера клиента
} log [NUM_ENTRIES];
```

Некоторые функции обработки журнала выглядят следующим образом.

а	topK_sourceIP();
б	peak_hour(int status); // пиковый период заданного состояния

5.18.1 [5] <5.11> К каким полям записи журнала будет обращаться данная функция его обработки? Сколько промахов при обращении к кэш-памяти на одну запись вызовет в среднем данная функция при условии использования 64-байтных блоков кэш-памяти и отсутствия предвыборки?

5.18.2 [10] <5.11> Как можно перестроить структуру данных для повышения эффективности кэш-памяти и локальности доступа? Покажите свой код определения структуры.

5.18.3 [10] <5.11> Приведите пример другой функции обработки журнала, которой больше подойдет другая организация структуры данных. Если обе функции играют важную роль, то как можно переписать программу, чтобы поднять общую производительность? Подкрепите свои рассуждения фрагментами кода и данных.

Для следующих задач воспользуйтесь данными из контрольных задач «Cache Performance for SPEC CPU2000 Benchmarks» (www.cs.wisc.edu/multifacet/misc/spec2000cache-data/) для пар контрольных задач, показанных в следующей таблице.

а	apsi/facerec
б	perlbmk/ammp

5.18.4 [10] <5.11> Для разных типов промахов (вынужденных, связанных с вместимостью и связанных с конфликтом) укажите, насколько снизятся коэффициенты промахов для каждой контрольной задачи при условии использования 64-килобайтных устройств кэш-памяти с переменной степенью ассоциативности?

5.18.5 [10] <5.11> Выберите степень ассоциативности, которая должна быть применена кэш-памятью данных уровня L1 объемом 64 Кбайт, совместно используемой обеими задачами. Если кэш-память уровня L1 должна иметь непосредственное отображение, выберите степень ассоциативности для кэш-памяти уровня L2, имеющей размер 1 Мбайт.

5.18.6 [20] <5.11> Приведите пример в таблице коэффициентов промахов, где увеличение степени ассоциативности действительно повышает коэффициент промахов. Сконструируйте конфигурацию кэш-памяти и поток обращений для демонстрации этого явления.

Ответы на вопросы для самопроверки

Раздел 5.1: 1 и 4. (Ответ 3 неверен, поскольку стоимость иерархии памяти варьируется от компьютера к компьютеру, но в 2008 году наивысшая стоимость обычно относится к DRAM.)

Раздел 5.2: 1 и 4. Более низкие издержки промахов могут позволить применение блоков меньшего размера, потому что не приходится амортизировать столь большое время ожидания, а более высокая пропускная способность памяти обычно ведет к укрупнению блоков, потому что издержки промахов становятся лишь немногим выше.

Раздел 5.3: 1.

Раздел 5.4: 1-а, 2-в, 3-б, 4-г.

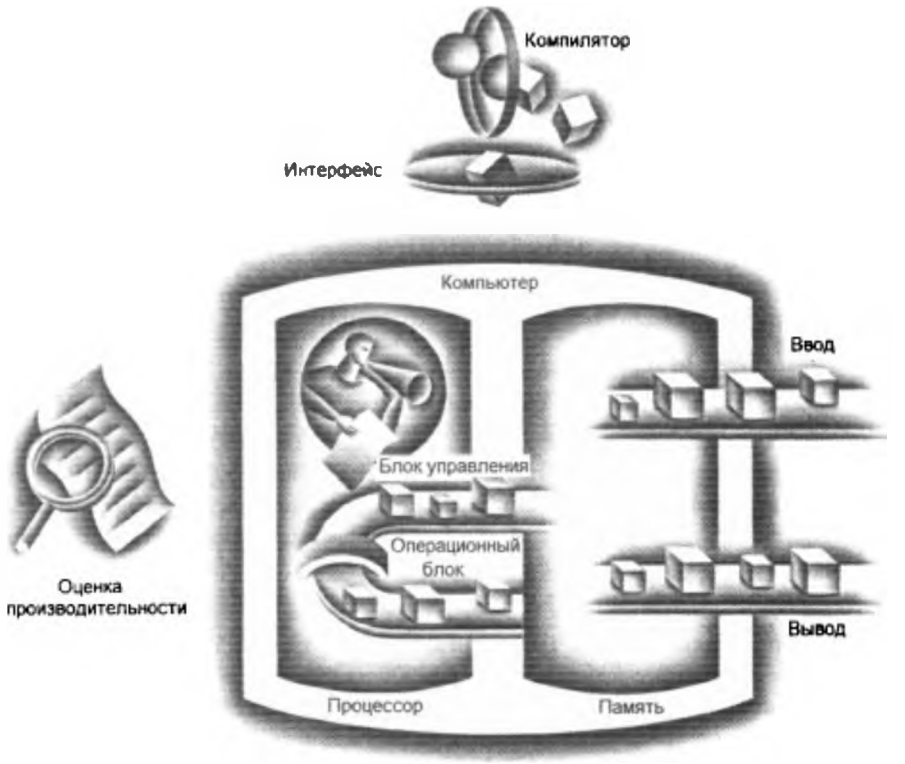
Раздел 5.5: 2. (Вынужденные промахи могут быть уменьшены за счет увеличения размера блоков и за счет предвыборки, так что утверждение 1 не соответствует действительности.)

Глава 6

Хранение информации и другие темы, связанные с вводом-выводом

Сочетание полосы пропускания и емкости... позволяет получить быстрый и надежный доступ к постоянно расширяющейся коллекции содержимого на быстрорастущих в объеме дисках и... в хранилищах Интернета.

Джордж Гилдер



6.1. Введение

Если зависание компьютера и необходимость его перезапуска людей просто раздражает, то авария запоминающего устройства с потерей информации приводит их в ярость. Поэтому стандарт надежности для средств хранения информации намного выше, чем для средств вычисления. Сети также рассчитаны на сбои при передаче данных, включая ряд механизмов для обнаружения сбоев и восстановления после них. Поэтому в системах ввода-вывода обычно уделяется намного больше внимания надежности и стоимости, а в процессорах и памяти основное внимание уделяется производительности и стоимости.

При разработке систем ввода-вывода также должны учитываться возможности расширения и разнообразия устройств, что не характерно для процессоров. Возможности расширения зависят от емкости запоминающих устройств, еще одного конструктивного параметра для устройств ввода-вывода; для выполнения задач системам могут понадобиться минимально допустимые емкости запоминающих устройств.

Хотя для устройств ввода-вывода производительность играет не самую важную роль, здесь тоже не все так просто. Например, при работе с некоторыми устройствами основное внимание уделяется латентности доступа, а при работе с другими устройствами критическим параметром считается пропускная способность. Кроме того, производительность зависит от многих аспектов системы: характеристик устройства, связью между устройством и остальной системой, иерархии памяти и операционной системы. Все компоненты, от отдельных устройств ввода-вывода и до процессора и системного программного обеспечения, будут влиять на надежность, расширяемость и производительность задач, включающих ввод-вывод. На рис. 6.1 показана структура простой системы с ее устройствами ввода-вывода.

Устройства ввода-вывода отличаются невероятным разнообразием. Для организации этого разнообразия используются три характеристики:

- *Поведение*. Ввод (однократное чтение), вывод (только запись, без возможности чтения) или сохранение (может подвергаться повторному считыванию и обычно повторной записи).
- *Партнер*. Либо человек, либо машина на другом конце устройства ввода-вывода, либо поставка данных на вход, либо чтение данных на выходе.
- *Скорость передачи данных*. Пиковая скорость, с которой данные могут быть перенесены между устройством ввода-вывода и оперативной памятью или процессором. При проектировании систем ввода-вывода полезно знать максимальные требования, предъявляемые устройством.

Например, клавиатура в качестве устройства ввода используется с *пиковой скоростью передачи данных* около 10 байт в секунду. В табл. 6.1 показаны некоторые устройства ввода-вывода, подключаемые к компьютерам.

Четыре важных устройства ввода-вывода: мышь, графический дисплей, диски и сети были коротко рассмотрены в главе 1. В данной главе будут более подробно рассмотрены запоминающие устройства и связанные с ними вопросы.

Рис. 6.1. Обычный набор устройств ввода-вывода. Связи между устройствами ввода-вывода, процессором и памятью исторически называются шинами, хотя это название означает общие параллельные проводники и большинство соединений ввода-вывода сегодня склоняются к последовательным линиям связи. Как будет показано в данной главе, при обмене данными между устройствами и процессором используются как прерывания, так и протоколы обмена информацией

Порядок оценки производительности устройств ввода-вывода зависит от способа его применения. В некоторых обстоятельствах основное внимание уделяется полосе пропускания системы. В таких случаях наиболее важным показателем будет ширина полосы пропускания. Даже ширина полосы пропускания может быть оценена двумя разными способами:

1. Какой объем данных можно переместить через систему за определенное время?
2. Сколько операций ввода-вывода может быть выполнено за единицу времени?

Какая из оценок производительности лучше, может зависеть от конкретных обстоятельств. Например, во многих мультимедийных приложениях большинство запросов ввода-вывода касается длинных потоков данных, и важной характеристикой становится ширина полосы пропускания при передаче данных. При иных обстоятельствах может понадобиться обработка большого количества непродолжительных, не связанных между собой обращений к устройству ввода-вывода. Примером подобной среды может послужить офис по обработке налоговой информации в U.S. National Income Tax Service (NITS). Эта служба главным образом занимается обработкой большого количества форм в отведенные для этого сроки; каждая налоговая форма сохраняется отдельно и имеет сравнительно небольшой размер. Ей вполне может подойти система, ориентированная на передачу больших файлов, но система ввода-вывода, способная поддерживать одновременную пере-

дачу множества небольших файлов, может обойтись намного дешевле и может справиться с обработкой миллионов налоговых форм намного быстрее.

Таблица 6.1. Разнообразие устройств ввода-вывода. Эти устройства могут различаться по предназначению: для ввода, для вывода или для хранения информации, по своим партнерам по связи (людям или другим компьютерам) и по пиковой скорости передачи данных. Скорости передачи данных охватывают восемь порядков. Заметьте, что сеть может быть как устройством ввода, так и устройством вывода, но не может быть использована для хранения информации. Скорости передачи данных для устройств всегда приводятся по основанию 10, поэтому 10 Мбит/с = 10 000 000 бит/с

Устройство	Поведение	Партнер	Скорость передачи данных, Мбит/с
Клавиатура	Ввод	Человек	0,0001
Мышь	Ввод	Человек	0,0038
Голосовой ввод	Ввод	Человек	0,2640
Звуковой ввод	Ввод	Машина	3,0000
Ввод сканера	Ввод	Человек	3,2000
Голосовой вывод	Вывод	Человек	0,2640
Звуковой вывод	Вывод	Человек	8,0000
Лазерный принтер	Вывод	Человек	3,2000
Графический дисплей	Вывод	Человек	800,0000–8000,0000
Кабельный модем	Ввод или вывод	Машина	0,1280–6,0000
Сеть / локальная сеть	Ввод или вывод	Машина	100,0000–10000,0000
Сеть / беспроводная локальная сеть	Ввод или вывод	Машина	11,0000–54,0000
Оптический диск	Хранение	Машина	80,0000–220,0000
Магнитная лента	Хранение	Машина	5,0000–120,0000
Флэш-память	Хранение	Машина	32,0000–200,0000
Магнитный диск	Хранение	Машина	800,0000–3000,0000

В других приложениях основное внимание уделяется времени отклика, которое, если вспомнить, является общим временем, затраченным на выполнение конкретной задачи. Если **запросы ввода-вывода** слишком длинные, время отклика будет во многом зависеть от пропускной способности, но во многих средах большинство обращений будут короткими, и система ввода-вывода снизит латентность каждого обращения и предоставит наилучшее время отклика. На однопользовательских машинах, например на настольных компьютерах и ноутбуках, время отклика является ключевой характеристикой производительности.

Запросы ввода-вывода
Чтение с устройств ввода-вывода или запись в эти устройства.

Большое количество приложений, особенно те, что используются для вычислений на обширном коммерческом рынке, требуют как высокой пропускной способности, так и короткого времени отклика. В качестве примера можно привести банкоматы, системы приема заказов и отслеживания имеющегося количества, файловые серверы и веб-серверы. В таких средах внимание уделяется и времени, которое затрачивается на каждую задачу, и количеству задач, обрабатываемых в секунду. Количество запросов к банкомату, которое можно обработать за час, не играет роли, если каждый такой запрос займет 15 минут – вы просто растеряете всех клиентов! Точно так же, если вы сможете быстро обработать каждый запрос к банкомату, но одновременно сможете обработать лишь небольшое количество запросов, вы не сможете обслуживать большое количество банкоматов, или же стоимость компьютерного оборудования, приходящаяся на каждый банкомат, будет слишком высока.

Таким образом, от безотказности и стоимости систем ввода-вывода зависят три класса: настольных компьютеров, серверов и встроенных компьютеров. Настольные компьютеры и встроенные системы фокусируются главным образом на времени отклика и многообразии устройств ввода-вывода, а серверные системы – на пропускной способности и расширяемости устройств ввода-вывода.

6.2. Безотказность, надежность и готовность

Пользователи требуют безотказных запоминающих устройств, но как это можно определить? В компьютерной индустрии одним обращением к словарю дело не обойдется. После массы дебатов было выработано следующее стандартное определение (Laprie, 1985):

Безотказность компьютерных систем заключается в качестве предоставления услуг, вызывающем доверие к той или иной службе. Предоставление системой услуг заключается в наблюдаемом реальном поведении этой системы со стороны другой системы (или систем), взаимодействующих с пользователями данной системы. У каждого компонента также есть определенное идеальное поведение, а технические требования к предоставлению услуг являются согласованным описанием ожидаемого поведения. Отказ системы возникает в том случае, когда реальное поведение отклоняется от предписанного.

Таким образом, чтобы определить безотказность, нужно обратиться к техническим требованиям ожидаемого поведения. Тогда пользователи увидят переходы системы между двумя состояниями предоставления услуг относительно технических требований к службе (сервису):

1. *Предоставление услуг*, когда сервис предоставляется в соответствии с требованиями.
2. *Прекращение предоставления услуг*, когда качество сервиса отличается от предъявляемых к нему требований.

Переходы из состояния 1 в состояние 2 вызываются *отказами*, а переходы из состояния 2 в состояние 1 называются *восстановлениями*. Отказы могут быть постоянными или периодическими. Последние являются более сложным случаем, потому что проблему труднее диагностировать, когда система колеблется между двумя состояниями. Куда проще установить причину постоянных отказов. Эти определения приводят к двум связанным с ними понятиям: надежности и готовности.

Надежность определяется продолжительностью непрерывного предоставления услуг — или, что то же самое, временем наработки на отказ. Следовательно, *наработка на отказ* (mean time to failure, MTTF) дисков, показанная в табл. 6.2, является показателем надежности. Родственным понятием является *частота отказов за год* (annual failure rate, AFR), представляющая собой процент устройств, отказ которых ожидается в течение года при заданном показателе MTTF. Перебой в эксплуатации измеряется *средним временем на восстановление* (mean time to repair, MTTR). *Среднее время безотказной работы* (mean time between failures, MTBF) является простой суммой MTTF + MTTR. Хотя термин MTBF получил довольно широкое распространение, в большинстве случаев более подходящим термином является MTTF.

Готовность является мерой предоставления услуг относительно переходов между двумя состояниями — предоставления и прекращения. Статистическая готовность имеет следующее количественное определение:

$$\text{Готовность} = \frac{\text{MTTF}}{(\text{MTTF} + \text{MTTR})}.$$

Обратите внимание на то, что надежность и готовность являются количественно определяемыми оценками, а не синонимами безотказности.

Какова же причина отказов? В табл. 6.2 приводится сводка из нескольких статей, в которой собраны данные о причинах отказов компьютерных и телекоммуникационных систем. Разумеется, весьма частой причиной отказов является человеческий фактор.

Для увеличения MTTF можно повысить качество компонентов или сконструировать системы для продолжения работы при наличии отказавших компонентов. Следовательно, отказ должен быть определен, сообразуясь с ситуацией. Отказ компонента может не привести к отказу системы. Чтобы четче обозначить эти различия, для определения отказа компонента используется термин *дефект* (fault). Существует три способа повышения MTTF:

1. *Предотвращение дефектов*. Предупреждение возникновения дефектов за счет совершенства конструкции.
2. *Отказоустойчивость*. Использование избыточности, применяемое главным образом к отказам аппаратуры, которое позволяет успешно завершить предоставление услуг в соответствии с предъявляемыми техническими требованиями, несмотря на возникший дефект. В разделе 6.9 описываются подходы, основанные на применении RAID-массивов, предназначенные для придания устройству хранения данных безотказности за счет отказоустойчивости.

Прогнозирование дефектов. Предсказание возникновения и формирования дефектов применительно к отказам оборудования и программного обеспечения, позволяющее производить замену компонента еще до возникновения дефекта.

Таблица 6.2. Сводные данные по исследованию причин отказов. Хотя собрать данные относительно того, являются ли причинами отказов сами операторы, довольно трудно, поскольку зачастую записи о причинах отказов делаются самими операторами, эти исследования все же содержат подобные данные. Часто встречались и другие категории причин, такие как природные катаклизмы, но их доля была крайне мала. Первые две строки были взяты из классической статьи Джима Грэя (Jim Gray, 1990), которая до сих пор широко цитируется по прошествии двух десятков лет после сбора этих данных. Следующие две строки взяты из статьи Мэрфи (Murphy) и Гента (Gent), написанной после изучения причин отказов за определенные периоды времени в системах VAX («Measuring system and software reliability using an automated data collection process», Quality and Reliability Engineering International 11:5, сентябрь–октябрь 1995, 341–53). Пятая и шестая строки показывают результат изучения Федеральной комиссией связи отказов в коммутируемой телефонной сети общего пользования — U.S. public switched telephone network, опубликованный Куном (Kuhn) («Sources of failure in the public switched telephone network», IEEE Computer 30:4, апрель 1997, 31–36) и Пэтти Энрикез (Patty Enriquez). Изучение трех интернет-служб предоставлено Оппенгеймером, Ганапатом и Паттерсоном (Oppenheimer, Ganapath, Patterson, 2003)

Оператор	Программа	Оборудование	Система	Год сбора данных
42%	25%	18%	Центр обработки данных (Tandem)	1985
15%	55%	14%	Центр обработки данных (Tandem)	1989
18%	44%	39%	Центр обработки данных (DEC VAX)	1985
50%	20%	30%	Центр обработки данных (DEC VAX)	1993
50%	14%	19%	Коммутируемая телефонная сеть общего пользования — U.S. public telephone network	1996
54%	7%	30%	Коммутируемая телефонная сеть общего пользования — U.S. public telephone network	2000
60%	25%	15%	Интернет-службы	2002

Сокращение среднего времени на восстановление (MTTR) может поспособствовать готовности не меньше, чем увеличение наработки на отказ (MTTF). Например, инструментальные средства для обнаружения дефектов, проведения диагностики и ремонта могут помочь людям, программным средствам и оборудованию сократить время, необходимое для восстановления.

Самопроверка

Что из следующих утверждений справедливо с точки зрения безотказности:

1. Если система соответствует стандарту, то все ее компоненты выполняют все, что от них ожидается.
2. Готовность является количественным показателем, выражающимся в процентах времени, в течение которого система выполняет все, что от нее ожидается.
3. Надежность является количественным показателем непрерывного предоставления системой соответствующих услуг.
4. В наши дни главным источником отказов является программное обеспечение.

6.3. Дисковое запоминающее устройство

Как указывалось в главе 1, магнитные диски основаны на использовании вращающейся пластины с магнитным покрытием и перемещающихся головок чтения-записи для обращения к диску. Дисковое запоминающее устройство является **энергонезависимым**, данные на нем остаются даже после отключения питания. Магнитный диск содержит набор пластин (1–4), каждая из которых имеет две пригодные для записи дисковые поверхности. Пакет пластин вращается со скоростью от 5400 до 15 000 оборотов в минуту и имеет диаметр от одного до трех с половиной дюймов. Каждая дисковая поверхность поделена на концентрические окружности, называемые **дорожками**. Обычно на каждой поверхности находится от 10 000 до 50 000 дорожек. Каждая дорожка в свою очередь разбита на **сектора**, в которых содержится информация; каждая дорожка может иметь от 100 до 500 секторов. Размер сектора обычно составляет 512 байт, хотя есть инициатива увеличить размер сектора до 4096 байт. Последовательность, записываемая на магнитный носитель, состоит из номера сектора, промежутка, информации для этого сектора, включая код коррекции ошибки, промежутка, номера следующего сектора и т. д.

Изначально все дорожки имели одно и то же количество секторов и, следовательно, одно и то же количество битов. С введением в начале 1990-х годов зональной поразрядной записи (zone bit recording, ZBR) дисковые накопители стали использовать разное количество секторов (а следовательно, и битов) на дорожку, исключив интервал между постоянным количеством битов. Благодаря ZBR увеличилось количество битов на внешних дорожках, и, как следствие, увеличился объем данных на дисковых накопителях.

В главе 1 было показано, что для чтения и записи информации головки чтения-записи должны перемещаться, чтобы находиться над нужным местом. Головки дисков для каждой поверхности соединены друг с другом и перемещаются единым блоком, поэтому каждая го-

Энергонезависимость
Свойство устройства хранения информации, позволяющее данным сохранять их значения даже при отключении питания.

Дорожка
Одна из нескольких тысяч концентрических окружностей, составляющих поверхность магнитного диска.

Сектор
Один из сегментов, составляющих дорожку магнитного диска; сектор является наименьшим блоком данных, который считывается с диска или записывается на него.

ловка находится над одной и той же дорожкой каждой поверхности. Для ссылки на все дорожки, находящиеся под головками в заданной точке всех поверхностей, используется термин *цилиндр*.

Для доступа к данным операционная система должна провести диск через процесс, состоящий из трех шагов. Первый шаг заключается в позиционировании головки над нужной дорожкой. Эта операция называется **поиском**, а время перемещения головки на нужную дорожку называется *временем поиска*.

Производители дисков в своих руководствах дают сведения о минимальном времени поиска, максимальном времени поиска и среднем времени поиска. Первые два из них измерить нетрудно, но среднее время открыто для широкой интерпретации, потому что оно зависит от расстояния поиска. Производители решили вычислять среднее время поиска в виде суммы времени на все возможные поисковые операции, разделенной на количество таких операций. Среднее время поиска обычно рекламируется в промежутке между 3 и 13 мс, но, в зависимости от приложения и диспетчеризации запросов к диску, реальное среднее время поиска может составлять всего лишь от 25% до 33% от рекламируемого, благодаря локальности обращений к диску. Эта локальность возникает как от последовательного характера обращений к одному и тому же файлу, так и от того, что операционная система пытается путем диспетчеризации сгруппировать подобные обращения.

Как только головка достигнет нужной дорожки, следует дождаться, пока под головку чтения-записи не попадет нужный сектор. Время этого ожидания называется **латентностью вращения** или **задержкой вращения**. Средняя латентность до получения нужной информации составляет половину времени оборота диска. Поскольку диск вращается со скоростью от 5400 до 15 000 оборотов в минуту, средняя латентность вращения находится между следующими значениями:

$$\frac{0{,}5 \text{ оборота}}{5400 \text{ об/мин}} = \frac{0{,}5 \text{ оборота}}{5400 \text{ об/мин}\left(60 \dfrac{\text{секунд}}{\text{в минуте}}\right)} = 0{,}0056 \text{ с} = 5{,}6 \text{ мс}$$

и

$$\frac{0{,}5 \text{ оборота}}{15\,000 \text{ об/мин}} = \frac{0{,}5 \text{ оборота}}{15\,000 \text{ об/мин}\left(60 \dfrac{\text{секунд}}{\text{в минуте}}\right)} = 0{,}0020 \text{ с} = 2{,}0 \text{ мс}.$$

Последний компонент обращения к диску, *время передачи данных*, является временем передачи блока битов. Время передачи является функцией от размера сектора, скорости вращения и плотности записи на дорожке. Скорость передачи данных в 2008 году была в диапазоне от 70 до 125 Мбайт/с. Одна из сложностей заключается в том, что большинство дисковых контроллеров имеют встроенную кэш-память, в которой хранятся передаваемые сектора; ско-

Поиск
Процесс позиционирования головки чтения-записи над нужной дорожкой диска.

Латентность вращения
Также называется задержкой вращения. Время, необходимое для подкрутки под головку чтения-записи нужного сектора; обычно за него принимается время полуоборота диска.

рость передачи данных из кэш-памяти, как правило, выше, и в 2008 году могла достигать 375 Мбайт/с (3 Гбит/с). Сегодня большинство передач данных имеют длину в несколько секторов.

Дисковый контроллер обычно занимается всеми тонкостями управления диском и переноса данных между диском и памятью. Контроллер добавляет последний компонент времени доступа к диску, *время контроллера*, являющееся издержками, вносимыми контроллером в осуществление доступа ввода-вывода. Среднее время выполнения операции ввода-вывода будет состоять из этих четырех временных показателей плюс время ожидания, вытекающее из того, что диск используется другими процессами.

Упражнение

Время чтения диска

Каким будет среднее время чтения или записи 512-байтного сектора для обычного диска со скоростью вращения 15 000 об/мин? Заявленное среднее время поиска равно 4 мс, скорость передачи данных — 100 Мбайт/с, а издержки контроллера — 0,2 мс. Предположим, что диск простаивает, поэтому время ожидания отсутствует.

Ответ

Среднее время доступа к диску равно среднему времени поиска + средняя задержка вращения + время переноса данных + издержка контроллера. При использовании заявленного среднего времени поиска ответ будет следующим:

$$4{,}0 \text{ мс} = \frac{0{,}5 \text{ оборота}}{15\,000 \text{ об/м}} + \frac{0{,}5 \text{ Кбайт}}{100 \text{ Мбайт/с}} + 0{,}2 \text{ мс} = 4{,}0 + 2{,}0 + 0{,}005 + 0{,}2 = 6{,}2 \text{ мс}.$$

Если измеренное среднее время поиска составляет 25% от заявленного среднего времени, ответ будет следующим:

$$1{,}0 \text{ мс} + 2{,}0 \text{ мс} + 0{,}005 \text{ мс} + 0{,}2 \text{ мс} = 3{,}2 \text{ мс}$$

Обратите внимание на то, что при учете измеренного среднего времени поиска, а не заявленного времени поиска, латентность вращения может стать наибольшим компонентом времени доступа.

Плотность записи на диске растет на протяжении последних 50 лет. Как показано на рис. 6.2, рост плотности и уменьшение физического размера диска просто невероятны. Усилия множества разработчиков дисков привели к широкому разнообразию дисковых накопителей. В табл. 6.3 показаны характеристики четырех магнитных дисков. В 2008 году эти диски от одного производителя имели стоимость хранения одного гигабайта, равную от 30 центов до 5 долларов. На более широком рынке стоимость обычно была в диапазоне от 20 центов до 2 долларов за гигабайт в зависимости от размеров, интерфейса и производительности.

В то время как диски продолжат свое существование в обозримом будущем, точка зрения о том, как определяется местонахождение блоков по номерам, не отличается постоянством. В модели «сектор—дорожка—цилиндр» предполагается, что соседние блоки находятся на одной и той же дорожке, обращение к блокам

ного и того же цилиндра из-за отсутствия времени поиска занимает меньше времени и одни дорожки находятся ближе других. Основой разбиения стало повышение уровня интерфейсов. «Разумные» высокоуровневые интерфейсы, такие как **ATA** и **SCSI**, требуют наличия микропроцессора в составе диска, что приводит к оптимизации производительности.

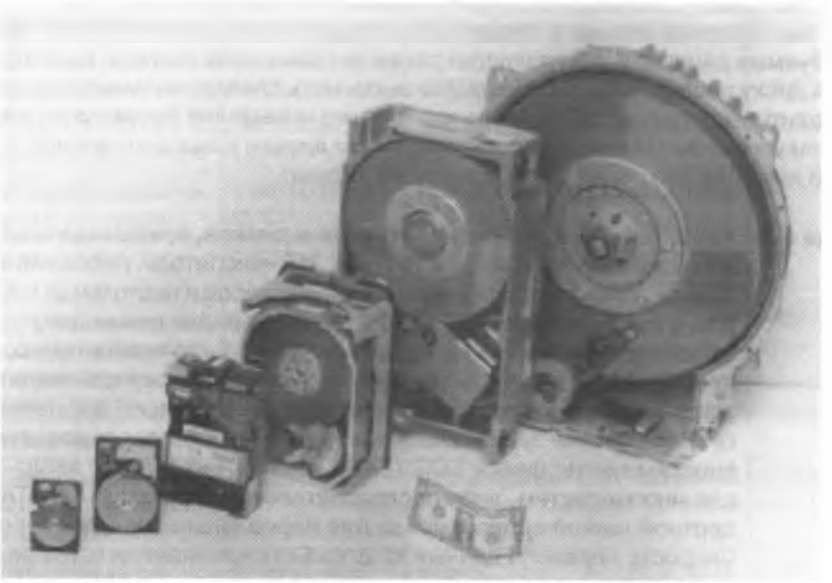


Рис. 6.2. Шесть магнитных дисков диаметрами от 14 до 1,8 дюйма. Изображенные на рисунке диски появились более 15 лет назад, и поэтому они не обладают вместимостью современных дисков таких же диаметров. Тем не менее на этой фотографии точно изображены их физические размеры. Самый широкий диск, DEC R81, содержит четыре 14-дюймовых пластины и способен хранить 456 Мбайт информации. Он был произведен в 1985 году. Диск диаметром 8 дюймов был произведен компанией Fujitsu в 1984 году и хранит 130 Мбайт на шести пластинах. Диск Micropolis RD53 имеет пять 5,25-дюймовых пластин и хранит 85 Мбайт. У диска IBM 0361 также пять пластин, но их диаметр равен 3,5 дюйма. Этот диск, выпущенный в 1988 году, хранит 320 Мбайт. В 2008 году у 3,5-дюймового диска, обладавшего наибольшей плотностью записи, имелись две пластины, и на том же пространстве хранился 1 Тбайт информации, в результате произошло увеличение плотности почти в 3000 раз! Диск Conner CP 2045, произведенный в 1990 году, имеет две 2,5-дюймовые пластины, содержащие 40 Мбайт. Самый маленький диск на этой фотографии — это Integral 1820. Этот диск был выпущен в 1992 году, он имеет одну пластину диаметром 1,8 дюйма и хранит 20 Мбайт информации

Для ускорения последовательных передач эти высокоуровневые интерфейсы делают организацию дисков больше похожей на ленты, чем на устройства произвольного доступа. Логические блоки выстроены по поверхности в виде серпантина в стремлении захватить все секторы, записанные с одинаковой плотностью битов. Следовательно, последовательные блоки могут находиться на разных дорожках. Пример будет

Интерфейс ATA (Advanced Technology Attachment)
Популярный в персональных компьютерах (PC) набор команд, используемый в качестве стандарта для устройств ввода-вывода.

Интерфейс SCSI (Small Computer Systems Interface)
Набор команд, используемый в качестве стандарта для устройств ввода-вывода.

показан на рис. 6.11 при рассмотрении заблуждения относительно популярной модели «сектор—дорожка—цилиндр».

Уточнение. Эти высокоуровневые интерфейсы позволяют добавлять к контроллерам дисков устройства кэш-памяти, обеспечивающие быстрый доступ к недавно считанным данным по запрошенным процессором передачам. В них используется сквозная запись, и они не обновляются в случае промахов при записи, а также часто включают алгоритмы предварительного извлечения, осуществляющие попытки предвидения востребуемых данных. В контроллерах также используется очередь команд, позволяющая диску решать, в каком порядке их выполнять для максимального увеличения производительности при соблюдении правильного поведения. Разумеется, такие возможности усложняют оценку производительности диска и повышают важность выбора рабочей нагрузки при проведении сравнения дисков.

Таблица 6.3. Характеристики четырех магнитных дисков, произведенных одной и той же компанией в 2008 году. Три накопителя, показанные в трех левых столбцах, предназначены для серверов и настольных компьютеров, а накопитель из последнего столбца предназначен для ноутбуков. Обратите внимание на то, что диаметр у третьего накопителя составляет всего 2,5 дюйма, но он относится к разряду высокопроизводительных устройств с наивысшей надежностью и наименьшим временем поиска. Показанные здесь диски оборудованы либо последовательными версиями интерфейса SCSI (SAS), стандартной шиной ввода-вывода для многих систем, либо последовательной версией ATA (SATA), стандартной шиной ввода-вывода для персональных компьютеров (PC). Скорости переноса данных из устройств кэш-памяти в 3–5 раз выше, чем скорости переноса данных с поверхности диска. Намного меньшая стоимость гигабайта у 3,5-дюймового накопителя SATA в основном обусловлена слишком высоким уровнем конкуренции на рынке PC, хотя разница в показателях производительности — количество вводов-выводов в секунду благодаря более высокой скорости вращения и меньшее время поиска при использовании SAS — существует. Срок службы этих дисков составляет пять лет. Учтите, что заявленный показатель MTTF предполагает номинальное напряжение питания и температуру. Срок службы диска может существенно сократиться при температуре и вибрации, не выдерживаемых на уровне эксплуатационных норм. Для получения дополнительной информации об этих дисках обратитесь на веб-сайт Seagate по адресу www.seagate.com

Характеристики	Seagate ST33000655SS	Seagate ST31000340NS	Seagate ST973451SS	Seagate ST9160821AS
Диаметр диска, дюймы	3,50	3,50	2,50	2,50
Емкость форматированного диска, Гбайт	147	1000	73	160
Количество дисковых поверхностей, головок	2	4	2	2
Скорость вращения, об/мин	15 000	7200	15 000	5400
Размер внутренней кэш-памяти диска, Мбайт	16	32	16	8

Характеристики	Seagate ST33000655SS	Seagate ST31000340NS	Seagate ST973451SS	Seagate ST9160821AS
Внешний интерфейс, пропускная способность, Мбайт/с	SAS, 375	SATA, 375	SAS, 375	SATA, 150
Скорость переноса данных в непродолжительном режиме, Мбайт/с	73–125	105	79–112	44
Минимальное время поиска (чтение/запись), мс	0,2/0,4	0,8/1,0	0,2/0,4	1,5/2,0
Среднее время поиска, мс	3,5/4,0	8,5/9,5	2,9/3,3	12,5/13,0
Среднее время наработки на отказ (MTTF), часов	1 400 000 при 25°C	1 200 000 при 25°C	1 600 000 при 25°C	—
Частота отказов за год (AFR), %	0,62%	0,73%	0,55%	—
Минимальное количество контактов магнитных головок с поверхностью диска при пуске-останове	—	50 000	—	>600 000
Гарантийный срок, лет	5	5	5	5
Невосстанавливаемых ошибок на количество считанных битов	<1 сектора на 10^{16}	<1 сектора на 10^{15}	<1 сектора на 10^{16}	<1 сектора на 10^{14}
Температура, ударная нагрузка (рабочая)	5°–55°C, 60 G	5°–55°C, 63 G	5°–55°C, 60 G	0°–60°C, 350 G
Размер, дюймы; вес, фунты	1,0× 4,0×5,8; 1,5	1,0×4,0×5,8; 1,4	0,6×2,8×3,9; 0,5	0,4×2,8×3,9; 0,2
Потребляемая мощность: работа/простой/ожидание, Вт	15/11/—	11/8/1	8/5,8/—	1,9/0,6/0,2
Гбайт/дюйм^3, Гбайт/Вт	6; 10	43; 91	11; 9	37; 84
Цена в 2008 г., $, $/Гбайт	~ 250; ~ 1,70	~ 275; ~ 0,30	~ 350; ~ 5,00	~ 100; ~ 0,60

Самопроверка

Какое из следующих утверждений справедливо для дисковых накопителей:

1. 3,5-дюймовые диски выполняют больше операций ввода-вывода в секунду, чем 2,5-дюймовые диски.
2. 2,5-дюймовые диски зачастую имеют наиболее высокий показатель количества гигабайт на ватт потребляемой мощности.
3. Последовательное чтение содержимого диска большой емкости занимает несколько часов.
4. Чтение содержимого диска большой емкости с использованием произвольных секторов по 512 байт занимает несколько месяцев.

6.4. Флэш-накопители

Многие пытались изобрести технологию, позволяющую найти замену дискам, претерпев при этом множество неудач: ПЗС-память, ЦМД-память и голографическая память так и не оправдали надежд. Ко времени предполагаемой поставки новой технологии диски, в соответствии со сделанными ранее предсказаниями, подвергались усовершенствованиям, цены на них существенно падали, и бросивший вызов продукт терял свою рыночную привлекательность.

Первым наиболее вероятным соперником стала флэш-память. Эта полупроводниковая память является такой же энергонезависимой, как и диски, но ее латентность в 100–1000 раз ниже, чем у диска, и, кроме того, она меньше по габаритам, обладает более высокой энергоэффективностью и ударопрочностью. Не менее важным обстоятельством является широкий рынок, открывшийся для флэш-памяти благодаря ее использованию в сотовых телефонах, цифровых камерах и MP3-плеерах, и инвестирование средств в ее совершенствование. В последнее время стоимость одного гигабайта флэш-памяти падала на 50% в год. В 2008 году цена гигабайта флэш-памяти была от 4 до 10 долларов, или в 2–40 раз выше, чем у дисков, и в 5–10 раз ниже, чем у DRAM-памяти. В табл. 6.4 сравниваются три устройства, основанные на флэш-памяти.

Таблица 6.4. Характеристики трех запоминающих устройств на основе флэш-памяти. Пакет стандартов CompactFlash был предложен корпорацией Sandisk в 1994 году для карт PCMCIA-ATA портативных PC-компьютеров. Поскольку этот пакет следует ATA-интерфейсу, он имитирует дисковый интерфейс, включая команды поиска, логические дорожки и т. д. Изделие RiDATA имитирует SATA-интерфейс 2,5-дюймового диска

Характеристики	Kingston SecureDigital (SD) SD4/8 GB	Transend Type I CompactFlash TS16GCF133	RiDATA Solid State Disk 2,5 дюйма SATA
Емкость отформатированного носителя, Гбайт	8	16	32
Байт на сектор	512	512	512
Скорость переноса данных (при чтении-записи), Мбайт/с	4	20/18	68/50
Энергопотребление при работе / в режиме ожидания, Вт	0,66/0,15	0,66/0,15	2,1/—
Размер: высота × ширина × длина, дюймы	0,94 × 1,26 × 0,08	1,43 × 1,68 × 0,13	0,35 × 2,75 × 4,00
Вес, г	2,5	11,4	52
Среднее время наработки на отказ, часов	> 1 000 000	> 1 000 000	> 4 000 000
Гбайт/куб. дюйм, Гбайт/Вт	84; 12	51; 24	8; 16
Лучшая цена (в 2008 году), $	~ 30	~ 70	~ 300

Несмотря на более высокую, чем у дисков, стоимость гигабайта, флэш-память популярна в мобильных устройствах, в частности потому, что она имеет меньший объем. В результате однодюймовые жесткие диски исчезли из некоторых рынков встроенных устройств. Например, в 2008 году MP3-плеер Apple iPod Shuffle продавался по 50 долларов и содержал 1 Гбайт памяти, а самый маленький диск содержал 4 Гбайт памяти и продавался чуть ли не дороже всего MP3-плеера.

Флэш-память — это *электрически стираемое программируемое постоянное запоминающее устройство, ЭСППЗУ* (electrically erasable programmable read-only memory, EEPROM). Первая флэш-память, называемая *NOR-флэш*, из-за схожести запоминающей ячейки со стандартным NOR-элементом, была прямым конкурентом других EEPROM-устройств и имела произвольную адресацию, как и любая другая память. Несколько лет спустя более высокая плотность записи была предложена памятью *NAND-флэш*, но эта память могла считываться и записываться только целыми блоками, поскольку в ней были удалены электрические цепи, необходимые для произвольного доступа. Гигабайт в памяти NAND-флэш стоит намного дешевле, и она куда более популярна, чем NOR-флэш; все устройства, показанные в табл. 6.4, используют память NAND-флэш. В табл. 6.5 приводится сравнительная характеристика основных параметров NOR и NAND флэш-памяти.

Таблица 6.5. Сравнение характеристик NOR и NAND флэш-памяти по состоянию на 2008 год. В этих устройствах допускается чтение байтов и 16-разрядных слов

Характеристики	NOR флэш-память	NAND флэш-память
Обычное применение	BIOS-память	флэш-накопитель USB
Минимальный размер доступа, байт	512	2048
Время чтения, мс	0,08	25
Время записи, мс	10,00	1500 для стирания + 250
Пропускная способность при чтении, Мбайт/с	10	40
Пропускная способность при записи, Мбайт/с	0,4	8
Износ, записей на ячейку	100 000	от 10 000 до 100 000
Лучшая цена/Гбайт (на 2008 год), $	65	4

В отличие от дисков и DRAM-памяти, но подобно другим изделиям, изготовленным по EEPROM-технологиям, биты флэш-памяти подвержены износу (см. табл. 6.5). Для преодоления подобных ограничений многие изделия NAND флэш-памяти включают контроллер для разнесения зоны записи путем изменения отображения блоков, многократно подвергавшихся записи, и их замены реже использующимися блоками. Эта технология называется *выравниванием степени износа*. Благодаря данной технологии такие потребительские товары, как сотовые телефоны, цифровые камеры, MP3-плееры или флэш-накопители, вряд ли превысят ограничения на количество записей во флэш-память. Такие контроллеры сни-

жают потенциальную производительность флэш-памяти, но они необходимы в том случае, когда высокоуровневое программное обеспечение не управляет выравниванием износа блоков. Но эти контроллеры могут повысить выход полезной продукции, не отображая те ячейки памяти, которые имеют производственные дефекты.

Ограничения по циклам записи являются одной из причин невысокой популярности применения флэш-памяти в ноутбуках и серверах. Тем не менее в 2008 году были проданы первые ноутбуки с флэш-памятью вместо жестких дисков со значительной скидкой, чтобы предложить более быструю загрузку, меньшие габариты и большую продолжительность работы от аккумуляторов. Как показано в табл. 6.4, существуют также устройства флэш-памяти, изготовленные в форм-факторе стандартного диска. Если объединить обе идеи накопителей, то *гибридные жесткие диски* будут включать, скажем, гигабайт флэш-памяти, чтобы ноутбуки быстрее загружались и чтобы экономилась энергия, позволяя дискам чаще находиться в выключенном состоянии.

В ближайшие годы флэш-память будет успешно конкурировать с жесткими дисками во многих устройствах с автономным питанием. По мере увеличения емкости и продолжающегося снижения стоимости гигабайта будет интересно увидеть, сможет ли более производительная и энергоэффективная флэш-память прижиться также на настольных компьютерах и серверах.

Самопроверка

Какое из следующих утверждений справедливо для флэш-памяти:

1. Подобно DRAM, флэш-память является полупроводниковым устройством памяти.
2. Подобно дискам, флэш-память не теряет информацию при отключении питания.
3. Время доступа при чтении NOR флэш-памяти такое же, как у DRAM.
4. Пропускная способности при чтении у NAND флэш-памяти такая же, как у диска.

6.5. Соединение процессоров, памяти и устройств ввода-вывода

В компьютерной системе различные подсистемы должны иметь взаимные интерфейсы. Например, память и процессор нуждаются в связи точно так же, как процессор и устройства ввода-вывода. Многие годы эта связь осуществлялась с помощью *шины*. Шина — это общий канал связи, использующий один набор проводников для соединения нескольких подсистем. Двумя основными преимуществами шинной организации являются универсальность и низкая стоимость. За счет определения единой схемы связи можно без особого труда добавлять новые устройства, а периферийные устройства можно даже перемещать между компьютерными системами, использующими один и тот же тип шины. Кроме того, шины рентабельны, поскольку один и тот же набор проводников используется совместно сразу несколькими способами.

Основной недостаток шины заключается в том, что она создает для связи узкое место, которое может ограничить максимальную пропускную способность ввода-вывода. Когда ввод-вывод должен проходить через единственную шину, пропускная способность этой шины ограничивает максимальную пропускную способность ввода-вывода. Разработка шинной системы, способной отвечать потребностям процессора, а также подключать к машине большое количество устройств ввода-вывода, представляет собой весьма трудную задачу.

Шины традиционно подразделяются на **шины процессор-память** и *шины ввода-вывода*. Шины процессор-память имеют небольшую протяженность, как правило, высокую скорость передачи данных и согласуются с системой памяти для обеспечения максимальной пропускной способности по маршруту «память—процессор». В отличие от них, шины ввода-вывода могут быть длиннее, иметь большее разнообразие типов подключенных к ним устройств и часто работают с широким диапазоном пропускной способности этих устройств. Шины ввода-вывода обычно не имеют прямого интерфейса с памятью, но используют для подключения к памяти шину процессор-память либо **монтажную шину** (backplane bus). Появлялись и другие специализированные шины, например графические, имеющие различные характеристики.

Одной из причин трудности разработки шины является то, что максимальная скорость шины во многом ограничена физическими факторами: длиной шины и количеством устройств. Эти физические ограничения не дают запустить шину с произвольной скоростью. Кроме того, разработка шин усложняется необходимостью поддержки широкого спектра устройств с сильно отличающейся латентностью и пропускной способностью.

Из-за трудностей запуска передачи данных на высокой скорости по множеству параллельных проводников, вызванных расфазировкой и отражением сигналов, промышленность перешла от параллельных общих шин к высокоскоростным последовательным двухточечным соединениям с коммутаторами. Поэтому такие сети ввода-вывода в большинстве случаев заменили в наших системах шины ввода-вывода.

В результате в данном издании этот раздел пришлось пересмотреть, чтобы придать особое значение общей задаче подключения устройств ввода-вывода, процессоров и памяти, а не фокусироваться исключительно на шинах.

Шина процессор-память
Короткая шина, как правило, высокоскоростная, соединяющая процессор и память, согласованная с системой памяти для получения максимальной пропускной способности на маршруте «память—процессор».

Монтажная шина
Шина, разработанная для того, чтобы дать возможность процессорам, памяти и устройствам ввода-вывода сосуществовать на единой шине.

Транзакция ввода-вывода
Последовательность операций, связанных с обменом данными, которая включает запрос и может включать ответ, каждый из которых может переносить данные. Транзакция инициируется единственным запросом и может занять множество отдельных операций, связанных с шиной.

Общие вопросы подключения

Рассмотрим обычную **транзакцию ввода-вывода**. Транзакция включает в себя две части: отправку адреса и получение или отправку данных. Шинные транзакции обычно определяются тем, что они делают с памятью. Транзакция *чтения* переносит данные *из* памяти (либо

к процессору, либо к устройству ввода-вывода), а транзакция *записи* записывает данные *в* память. Очевидно, что эта терминология может сбить с толку. Чтобы избежать этого, мы попробуем использовать понятия *ввод* и *вывод*, всегда определяемые с позиции процессора: операция ввода осуществляет ввод данных из устройства в память, где процессор может их прочитать, а операция вывода занимается выводом данных в устройство из памяти, куда процессор их записывает.

Соединения ввода-вывода служат в качестве способа расширения машины и подключения новых периферийных устройств. Для облегчения этих соединений компьютерная промышленность разработала несколько стандартов. Эти стандарты служат в качестве технических характеристик для производителей компьютеров, а также для производителей периферийных устройств. Стандарт гарантирует разработчику компьютера, что периферийные устройства будут доступны новой машине, а создателю периферийных устройств — что пользователи смогут подключить свое новое оборудование. В табл. 6.6 приведена сводка основных характеристик пяти популярных стандартов ввода-вывода: Firewire, USB, PCI Express (PCIe), Serial ATA (SATA) и Serial Attached SCSI (SAS). Они используются для подключения различных устройств, от клавиатуры до цифровых камер и дисков, к настольному компьютеру.

Традиционные шины являются **синхронными**. Это означает, что шина имеет в цепях управления тактовый генератор и использует для связи фиксированный протокол, зависящий от тактовых импульсов. Например, для осуществления чтения из памяти может использоваться протокол, согласно которому в течение первого тактового цикла передается адрес и команда чтения и используются линии управления для обозначения типа запроса. Затем в течение пятого тактового цикла от памяти может быть запрошен ответ в виде слова данных. Протокол такого типа может быть легко реализован в виде небольшого конечного автомата. Поскольку протокол уже предопределен и привлекает лишь небольшое количество логических операций, шина может работать на высокой скорости, а логический блок интерфейса — иметь небольшие размеры. Но у синхронных шин имеются два основных недостатка. Во-первых, любое устройство на шине должно работать с той же тактовой частотой. Во-вторых, из-за проблем искажения тактового импульса синхронные шины не могут быть длинными, если они обладают высокой скоростью.

Синхронная шина
Шина, которая имеет в цепях управления тактовый генератор и использует фиксированный протокол обмена данными, зависящий от тактовых импульсов.

Асинхронный обмен данными
Использует для координации вместо тактовых импульсов протокол установления связи. Он может приспосабливаться под широкий диапазон устройств, имеющих разную скорость передачи данных.

Протокол установления связи
Серия шагов, используемых для координации передач по асинхронной шине, в которой отправитель и получатель переходят к следующему шагу только при согласии обеих сторон на завершение текущего шага.

Наличие этих проблем привело к **асинхронному** обмену данными, который не управляется тактовыми импульсами. Ввиду отсутствия синхроимпульсов асинхронная связь может приспособиться под широкий диапазон устройств, а шина может иметь протяженность, не связанную с искажениями синхроимпульсов или с проблемами синхронизации. Приведенные в табл. 6.6 стандарты относятся к асинхронному обмену данными.

Для координации передачи данных между отправителем и получателем в асинхронных шинах используется протокол **установления связи**. Этот протокол состоит из серии шагов, осуществляемых отправителем и получателем с переходом к следующему шагу только в случае согласия обеих сторон. Протокол реализован с помощью дополнительного набора линий управления.

Таблица 6.6. Основные характеристики пяти доминирующих стандартов ввода-вывода. В столбцах предполагаемого использования показано, для чего разработан стандарт — для использования внешних по отношению к компьютеру кабелей или только для использования внутри компьютера коротких кабелей или проводников на печатных платах. PCIe может поддерживать одновременное чтение и запись, поэтому в некоторых публикациях пропускная способность на канал (lane) удваивается на основе предположения о разбиении пропускной способности 50 на 50 % на чтение и запись

Характеристика	Firewire (1394)	USB 2.0	PCI Express	Serial ATA	Serial Attached SCSI
Предполагаемое использование	Внешний	Внешний	Внутренний	Внутренний	Внешний
Количество устройств на канал	63	127	1	1	4
Основная ширина шинных (количество сигналов)	4	2	2 на канал (lane)	4	4
Теоретическая пиковая пропускная способность, Мбайт/с	50 (Firewire 400) или 100 (Firewire 800)	0,2 (низкая скорость) 1,5 (полная скорость), или 60 (высокая скорость)	250 на канал (1x); PCIe-карты поставляются в вариантах 1x, 2x, 4x, 8x, 16x или 32x	300	300
Горячее подключение	Да	Да	Зависит от форм-фактора	Да	Да
Максимальная длина шины (медный провод), м	4,5	5	0,5	1	8
Стандартное название	IEEE 1394. 1394b	USB Implementors Forum	PCI-SIG	SATA-IO	T10 committee

Соединения ввода-вывода процессоров x86

На рис. 6.3 показана система ввода-вывода традиционного персонального компьютера (PC). Процессор соединяется с периферийными устройствами посредством двух основных микросхем. Микросхема рядом с процессором является контроллером-концентратором памяти, ее обычно называют *северным мостом*, а подклю-

ченная к ней микросхема является контроллером-концентратором ввода-вывода, она обычно называется *южным мостом*.

Северный мост по существу является контроллером прямого доступа к памяти (DMA), подключающим процессор к памяти, возможно, к графической карте и к микросхеме южного моста. Южный мост подключает северный мост к множеству шин ввода-вывода. Компании Intel, AMD, NVIDIA и другие предлагают широкий выбор этих микросхем для связи процессора с внешним миром.

В табл. 6.7 приведены три примера наборов микросхем («чипсетов»). Обратите внимание, что компания AMD внедрила микросхему северного моста в процессор Opteron и более поздние продукты, сократив тем самым количество микросхем и латентность доступа к памяти и графическим картам путем упразднения электрических цепей, соединяющих микросхемы. Закон Мура не утрачивает свою силу, и постоянно увеличивающееся количество контроллеров ввода-вывода, которые в прежние времена были доступны в виде дополнительных карт, подключаемых к шинам ввода-вывода, было внедрено в эти чипсеты. Например, AMD Opteron X4 и Intel Nehalem включают северный мост в микропроцессор, а в кристалл южного моста Intel 975 включен RAID-контроллер (см. раздел 6.9).

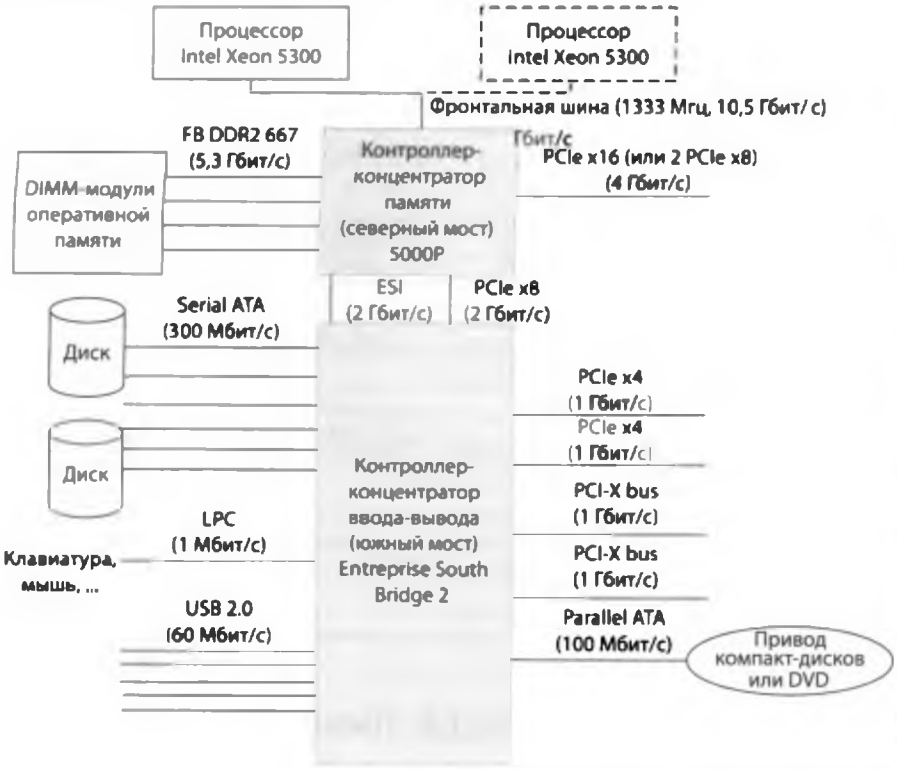


Рис. 6.3. Организация ввода-вывода на сервере Intel, использующем чипсет Intel 5000P. При условии, что обмен данными приходится на операции чтения и записи поровну, пропускную способность на каждую связь по PCIe можно удвоить

Таблица 6.7. Два чипсета ввода-вывода от компании Intel и один чипсет от компании AMD. Обратите внимание на то, что функции северного моста включены в микропроцессор AMD, они также включены и в один из самых новых процессоров Intel Nehalem

	Чипсет Intel 5000P	Чипсет Intel 975X	AMD 580X CrossFiret
Целевой сегмент	Сервер	Рабочий персональный компьютер	Сервер и рабочий персональный компьютер
Фронтальная шина (64 разряда), МГц	1066/1333	800/1066	—
Контроллер-концентратор памяти (северный мост)			
Название изделия	Blackbird 5000P MCH	975X MCH	
Количество выводов	1432	1202	
Тип и скорость памяти	DDR2 FBDIMM 667/533	DDR2 800/667/533	
Шины памяти, ширина	4 × 72	1 × 72	
Количество DIMM, DRAM/DIMM, Гбайт	16, 1/2/4	4, 1/2	
Максимальный объем памяти, Гбайт	64	8	
Возможна ли корректировка ошибок памяти?	Да	Нет	
PCIe / Внешний графический интерфейс	1 PCIe x16 или 2 PCIe x8	1 PCIe x16 или 2 PCIe x8	
Интерфейс южного моста	PCIe x8, ESI	PCIe x8	
Контроллер-концентратор ввода-вывода (южный мост)			
Название изделия	6321 ESB	ICH7	580X CrossFire
Количество выводов	1284	652	549
PCI-шина: ширина, скорость	Две 64-разрядные, 133 МГц	32-разрядная, 33 МГц, 6 главных устройств	—
Порты PCI Express	Три PCIe x4		Два PCIe x16, Четыре PCI x1
Ethernet MAC-контроллер, интерфейс	—	1000/100/10 Мбит	—
Порты USB 2.0, контроллеры	6	8	10
Порты ATA, скорость	Один 100	Два 100	Один 133
Порты Serial ATA	6	2	4
AC-97 аудиоконтроллер, интерфейс	—	Да	Да
Управление вводом-выводом	SMbus 2.0, GPIO	SMbus 2.0, GPIO	ASF 2.0, GPIO

Эта схема соединений ввода-вывода предоставляет возможность электрического соединения устройств ввода-вывода, процессоров и памяти, а также определяет низкоуровневый протокол связи. Поверх этого базового уровня должны быть определены аппаратные и программные протоколы для управления передачей данных между устройствами ввода-вывода и памятью, и для процессора, чтобы определить команды для устройств ввода-вывода. Эти темы рассматриваются в следующем разделе.

Самопроверка

И сети, и шины служат для объединения компонентов. Что из перечисленного ниже является достоверными сведениями об этих устройствах?

1. Сети и шины ввода-вывода практически всегда работают согласно какому-нибудь стандарту.
2. Сети и шины ввода-вывода практически всегда работают в синхронном режиме.

6.6. Организация интерфейса устройств ввода-вывода с процессором, памятью и операционной системой

Порядок передачи слова или блока по набору проводников определяется протоколом шины или сети. Но чтобы фактически произошел перенос данных из устройства в адресное пространство памяти некой пользовательской программы, остается ряд других задач, которые должны быть выполнены. В этом разделе основное внимание уделяется этим задачам и дается ответ на следующие вопросы:

- Как запрос пользователя на ввод-вывод превращается в команду устройства и как эта команда попадает на устройство?
- Как фактически происходит перенос данных в определенное место в памяти или из этого места?
- Какова роль операционной системы?

Из ответов на эти вопросы станет понятно, что главную роль в работе с устройствами ввода-вывода играет операционная система, действующая в качестве интерфейса между оборудованием и программами, выдающими запросы на ввод-вывод.

Обязанности операционной системы предопределяются тремя характеристиками систем ввода-вывода:

1. Системы ввода-вывода совместно используются множеством программ, которые выполняются процессором.
2. Системы ввода-вывода часто используют прерывания (исключения, генерируемые извне) для передачи информации об операциях ввода-вывода. Поскольку прерывания вызывают переход в режим ядра или супервизора, они должны обрабатываться операционной системой.

Низкоуровневое управление устройством ввода-вывода является довольно сложной задачей, потому что здесь требуется управление рядом параллельных событий и потому что требования по правильному управлению устройством бывают сильно детализированными.

Интерфейс аппаратного и программного обеспечения

Ранее перечисленные три характеристики систем ввода-вывода приводят к тому, что операционная система должна предоставлять несколько различных функций:

- Операционная система гарантирует, что пользовательская программа имеет доступ только к той части устройств ввода-вывода, на которую у пользователя есть права. Например, операционная система должна запретить программе чтение или запись файла на диске, если владелец файла не предоставил этой программе доступ к этому файлу. В системах с общими устройствами ввода-вывода защита не может обеспечиваться в том случае, если пользовательские программы могут выполнять непосредственный ввод-вывод.
- Операционная система обеспечивает абстракции устройств, к которым осуществляется обращение, путем предоставления подпрограмм, управляющих низкоуровневыми операциями этих устройств.
- Операционная система обрабатывает прерывания, сгенерированные устройствами ввода-вывода, точно так же, как она обрабатывает исключения, сгенерированные программой.
- Операционная система старается предоставить равный доступ к общим ресурсам ввода-вывода, а также проводит диспетчеризацию доступа, чтобы улучшить пропускную способность системы.

Для выполнения этих функций по поручению пользовательских программ операционная система должна уметь связываться с устройствами ввода-вывода и не давать пользовательской программе связываться с этими устройствами напрямую. Здесь требуются три типа информационного взаимодействия:

- Операционная система должна уметь давать команды устройствам ввода-вывода. В числе этих команд должны быть не только операции вроде чтения и записи, но также и операции, которые должны быть осуществлены на устройстве, например поиск на диске.
- Устройство ввода-вывода должно уметь уведомить операционную систему о том, что оно завершило операцию или обнаружило ошибку. Например, уведомить операционную систему, когда будет завершен поиск.
- Должен быть осуществлен обмен данными между памятью и устройством ввода-вывода. Например, блок, считываемый с диска, должен быть перемещен с диска в память. Как все это делается, будет показано в нескольких следующих подразделах.

Выдача команд на устройства ввода-вывода

Чтобы выдать команду на устройство ввода-вывода, процессор должен уметь обратиться к устройству и предоставить ему одно или несколько командных слов. Для обращения к устройству используются два метода: ввод-вывод с отображением на память и специальные инструкции ввода-вывода. При **вводе-выводе с отображением на память** устройствам ввода-вывода выделяется часть адресного пространства. Чтение или запись с указанием этих адресов интерпретируется как команда устройству ввода-вывода.

Например, для отправки данных на устройство ввода-вывода может использоваться операция записи, в которой данные будут восприняты в качестве команды. Когда процессор выставляет адрес и данные на шине памяти, система памяти игнорирует операцию, поскольку адрес относится к той части адресного пространства, которая выделена для ввода-вывода. А вот контроллер устройства видит операцию, записывает данные и переносит их на устройство как команду. Пользовательские программы отстранены от непосредственного использования операций ввода-вывода, потому что операционная система не предоставляет адресное пространство, выделенное устройствам ввода-вывода, и поэтому адреса защищены преобразованием адресов. Ввод-вывод с отображением на память может также использоваться для передачи данных путем записи или чтения по избранным адресам. Устройство использует адреса для определения типа команды, и данные могут быть предоставлены с помощью записи и получены с помощью чтения. В любом случае в адресе кодируется как идентичность устройства, так и тип переноса данных между процессором и устройством.

Фактически чтение или запись данных при выполнении запроса программы обычно требует нескольких отдельных операций ввода-вывода. Кроме этого процессору между отдельными командами может потребоваться опрашивать состояние устройства, чтобы определить успешное выполнение команды. Например, у простого принтера имеются два регистра устройства ввода-вывода — один для информации о состоянии, а другой для распечатываемых данных. В регистре состояния содержится *бит выполнения* (done bit), который устанавливается принтером после распечатки символа, и *бит ошибки* (error bit), показывающий, что принтер зажевал бумагу или оказался без бумаги. Каждый выводимый на печать байт данных помещается в регистр данных. Затем процессор, перед тем как сможет поместить в буфер следующий символ, должен ждать, пока принтер не установит бит выполнения. Процессор должен также проверить бит ошибки, чтобы отследить возникновение проблемы. Каждая из этих операций требует отдельного обращения к устройству ввода-вывода.

Ввод-вывод с отображением на память
Схема ввода-вывода, при которой устройствам ввода-вывода выделяется часть адресного пространства, и чтение и запись по этим адресам интерпретируются как команды устройству ввода-вывода

Инструкция ввода-вывода
Специальная инструкция, используемая для выдачи команды устройству ввода-вывода, и определяющая как номер устройства, так и слово команды (или местонахождение этого слова в памяти).

Уточнение. Альтернативой вводу-выводу с отображением на память служит использование в процессоре специальных **инструкций ввода-вывода**. Эти инструкции могут определять как

номер устройства, так и слово команды (или местонахождение слова команды в памяти). Процессор передает адрес устройства по набору проводников, которые обычно составляют часть шины ввода-вывода. Текущая команда может быть передана по имеющимся на шине линиям данных. Примерами компьютеров, использующих инструкции ввода-вывода, могут служить Intel x86 и IBM 370. Пользовательские программы могут быть отстранены от непосредственного обращения к устройствам путем запрещения выполнения инструкций ввода-вывода.

Связь с процессором

Процесс периодической проверки битов состояния для определения своевременности следующей операции ввода-вывода, как в предыдущем примере, называется опросом (polling). Опрос является простейшим способом связи устройства ввода-вывода с процессором. Устройство ввода-вывода просто помещает информацию в регистр состояния, а процессор может обратиться к нему и получить информацию. Процессор целиком вовлечен в процесс управления и делает всю работу самостоятельно.

Опрос может использоваться разными способами. Устройства ввода-вывода опрашиваются встраиваемыми приложениями реального времени, поскольку скорости ввода-вывода предопределены, что делает издержки ввода-вывода более предсказуемыми и помогает разобраться в текущем моменте. Как будет показано, это позволяет использовать опрос даже когда скорость ввода-вывода будет несколько более высокой.

Недостаток опроса состоит в том, что на него тратится впустую много процессорного времени, поскольку процессоры работают намного быстрее устройств ввода-вывода. Процессор может прочитать регистр состояния много раз только для того, чтобы обнаружить, что устройство еще не завершило сравнительно более медленную операцию ввода-вывода или что мышь не перемещалась со времени последнего опроса. Когда устройство завершает операцию, мы также должны считать его состояние, чтобы определить успешность этой операции.

Издержки интерфейса, построенного на опросе, стали известны очень давно, что привело к изобретению прерываний для оповещения процессора о том, что устройству нужно уделить внимание со стороны процессора. **Ввод-вывод, управляемый прерываниями**, который используется почти всеми системами, у которых имеется хотя бы небольшое количество устройств, применяет прерывания ввода-вывода, чтобы показать процессору, что устройству ввода-вывода нужно уделить внимание. Когда устройство хочет уведомить процессор, что оно завершило ту или иную операцию или нуждается во внимании, оно заставляет процессор прервать его работу.

Опрос
Процесс периодической проверки состояния устройства ввода-вывода для определения необходимости в его обслуживании.

Ввод-вывод, управляемый прерываниями
Схема ввода-вывода, использующая прерывания, чтобы показать процессору, что устройство ввода-вывода нуждается во внимании.

Прерывания ввода-вывода похожи на исключения, которые рассматривались в главах 4 и 5, но с двумя важными отличиями:

1. По отношению к выполнению инструкций прерывание ввода-вывода является асинхронным. То есть прерывание не связано ни с одной из инструкций и не препятствует завершению инструкции. Это сильно отличается от исключения, связанного с отсутствием страницы, или от исключения, вызванного арифметическим переполнением. Наш блок управления должен только проверять наличие ожидающего решения прерывания ввода-вывода к моменту запуска новой инструкции.
2. Кроме факта возникновения прерывания ввода-вывода, нужно передать еще и дополнительную информацию, например указать выдавшее его устройство. Более того, прерывания представляют устройства, которые могут иметь разные приоритеты и чьи прерывания имеют разную связанную с ними категорию срочности.

Для передачи информации процессору, например об устройстве, выдавшем прерывание, система может использовать либо векторные прерывания, либо регистр причины (Cause), используемый при исключениях. Когда процессор распознает прерывание, устройство может отправить либо адрес вектора, либо поле состояния для помещения в регистре причины. В результате, когда управление будет передано операционной системе, та будет знать об устройстве, выдавшем прерывание, и сможет немедленно детально исследовать устройство. Механизм прерываний избавляет процессор от необходимости опроса устройства, позволяя ему вместо этого сосредоточиться на выполнении программ.

Уровни приоритета прерываний

Чтобы справиться с разными приоритетами устройств ввода-вывода, большинство механизмов прерывания имеют несколько уровней приоритета; в операционной системе UNIX используется от четырех до шести уровней. Эти приоритеты определяют порядок, в котором процессор должен обработать прерывания. Приоритеты есть как у внутренних исключений, так и у прерываний ввода-вывода; обычно у прерываний ввода-вывода более низкий приоритет, чем у внутренних исключений. Может быть несколько уровней приоритета прерываний ввода-вывода, при этом высокоскоростные устройства связываются с более высоким уровнем приоритета.

Для поддержки уровней приоритета прерываний MIPS предоставляет примитивы, позволяющие операционной системе реализовать свою политику. Это похоже на порядок обработки в MIPS промахов при обращении к TLB. Ключевые регистры показаны на рис. 6.4.

Регистр состояния (Status) определяет того, кто может прервать работу компьютера. Если бит разрешения прерывания установлен в 0, то прервать его работу не может никто. Более конкретизированная блокировка прерываний возможна при использовании поля маски прерываний. В маске имеются биты, соответствующие каждому биту в поле отложенных прерываний регистра причины (Cause). Для разрешения соответствующего прерывания в соответствующем бите поля маски должна быть выставлена единица. Когда произойдет прерывание, операционная система может определить его причину в поле кода исключения регистра состояния

(Status): 0 означает возникновение прерывания, а остальные значения относительно исключений рассмотрены в главе 5.

При обработке прерывания должны быть выполнены следующие шаги:

1. Проведение логической операции «И» над полем ожидающего обработки прерывания и полем маски прерываний, чтобы посмотреть, какие из возможных прерываний могут быть заблокированы. Копии этих двух регистров делаются с помощью инструкции `mfc0`.
2. Выбор прерывания, имеющего наивысший приоритет. По соглашению, касающемуся разработки программного обеспечения, наивысшим приоритетом обладает самое левое прерывание.
3. Сохранение поля маски прерываний в регистре состояния.
4. Изменение поля маски прерываний для запрещения всех прерываний, имеющих приоритет равного или более низкого уровня.
5. Сохранение состояния процессора, необходимое для обработки прерывания.
6. Установление бита разрешения в регистре причины (Cause) в единицу, чтобы разрешить прерывания более высокого уровня.
7. Вызов соответствующей программы обработки прерывания.
8. Установка перед восстановлением состояния бита разрешения в регистре причины в нуль. Это позволит восстановить поле маски прерываний.

Рис. 6.4. Регистры причины (Cause) и состояния (Status). Эта версия регистра Cause соответствует архитектуре MIPS-32. Ранняя архитектура MIPS I имела три вложенных набора битов режимов ядра/пользователя и битов разрешения прерываний для поддержки вложенных прерываний

Как *уровни приоритета прерываний* (interrupt priority levels, IPL) соотносятся с данным механизмом? Уровень IPL задается операционной системой и хранится в памяти процесса. IPL задается для каждого процесса. На самом низком IPL разрешены все прерывания. И наоборот, на самом высоком IPL все прерывания заблокированы. Повышение и понижение IPL требует изменений в поле маски прерываний регистра состояния.

Уточнение. Два самых младших разряда полей прерываний, ожидающих обработку, и маски прерываний предназначены для программных прерываний, имеющих самый низкий приоритет. Обычно они используются прерываниями с более высоким уровнем приоритета, чтобы, как только будет обработана неотложная причина прерывания, дать возможность обработки прерываний с более низким уровнем приоритета. После завершения обработки прерывания, имеющего более высокий уровень приоритета, задачи, имеющие более низкий уровень приоритета, будут оповещены и обработаны.

Передача данных между устройством и памятью

Мы рассмотрели два разных способа, позволяющих устройству связываться с процессором. Эти две технологии — опрос и прерывания ввода-вывода — создают основу для двух способов реализации передачи данных между устройством ввода-вывода и памятью. Обе эти технологии лучше всего работают с устройствами, имеющими низкую пропускную способность, где мы наиболее заинтересованы в снижении стоимости контроллера устройства и интерфейса, чем в предоставлении более высокой пропускной способности передачи данных. И опрос, и передача данных, основанная на использовании прерываний, возлагают весь груз ответственности за перемещение данных и за управление этим перемещением на процессор. После рассмотрения этих двух схем мы рассмотрим схему, которая более подходит для высокопроизводительных устройств или наборов устройств.

Процессор может использоваться для передачи данных между устройством и памятью, основанной на опросе. В приложениях реального времени процессор загружает данные из регистров устройства ввода-вывода и сохраняет их в памяти.

В качестве альтернативы используется механизм, делающий передачу данных управляемой с помощью прерываний. В этом случае операционная система попрежнему будет передавать данные из устройства и в него небольшими порциями байтов. Но, поскольку операции ввода-вывода управляются прерываниями, операционная система, пока данные считываются с устройства или записываются в него, просто работает над решением других задач. Когда операционная система заметит прерывание от устройства, она считывает состояние на наличие ошибок. Если ошибки отсутствуют, операционная система может предоставить новую порцию данных, например за счет последовательности записей с отображением на память. После передачи последнего байта согласно запросу ввода-вывода и завершения операции ввода-вывода операционная система может проинформировать об этом программу. Вся работа в данном процессе — обращение к устройству и к памяти для передачи каждой порции данных — проделывается процессором и операционной системой.

Ввод-вывод, управляемый прерываниями, освобождает процессор от необходимости ждать каждого события ввода-вывода, хотя, если этот метод используется для передачи данных с жесткого диска и на него, издержки могут все же стать неприемлемыми, поскольку при такой передаче данных может быть задействована слишком большая часть времени процессора. Для таких высокопроизводительных устройств, как жесткие диски, передача состоит в основном из относительно боль-

ших блоков данных (в сотни и тысячи байт). Поэтому разработчики компьютеров обрели механизм для разгрузки процессора и принуждения контроллера устройства к передаче данных непосредственно из памяти или в память без привлечения процессора. Это механизм называется **прямым доступом к памяти** (direct memory access, DMA). Механизм использования прерываний все еще используется устройствами для связи с процессором, но только при завершении передачи данных, связанной с операцией ввода-вывода или при возникновении ошибки.

DMA реализуется с помощью специализированного контроллера, который передает данные между устройством ввода-вывода и памятью независимо от процессора. DMA-контроллер становится **ведущим** и напрямую управляет операциями чтения или записи между ним самим и памятью. При DMA-передаче выполняются три шага:

1. Процессор настраивает DMA, предоставляя указание устройства, операцию, которую на нем нужно выполнить, адрес памяти, являющийся источником или приемником передаваемых данных, и количество передаваемых байтов.
2. Блок DMA запускает операцию на устройстве и принимает решение на связь. Когда данные доступны (с устройства или из памяти), он передает эти данные. DMA-устройство предоставляет адрес памяти для чтения или записи. Если запрос требует более одной передачи данных, блок DMA генерирует следующий адрес памяти и инициирует следующую передачу. Используя этот механизм, блок DMA может завершить всю передачу, которая может иметь длину в тысячи байт, не надоедая при этом процессору. Многие DMA-контроллеры имеют небольшой объем памяти, позволяющий им более гибко работать либо при задержках в передаче, либо при ожидании получения ведущей роли.
3. Как только DMA-передача завершится, контроллер прерывает работу процессора, который затем может установить путем опроса DMA-устройства или изучения памяти факт успешного завершения операции.

В компьютерной системе может быть несколько устройств DMA. Например, в компьютере с единой шиной «процессор—память» и несколькими шинами ввода-вывода каждый контроллер шины ввода-вывода зачастую содержит DMA-процессор, управляющий всеми передачами между устройством на шине ввода-вывода и памятью.

В отличие от опроса или ввода-вывода, управляемого прерываниями, DMA может использоваться для создания интерфейса с жестким диском, не забирающего все процессорные циклы на обслуживание одного устройства ввода-вывода. Разумеется, если процессор также вступает в соревнование за обращение к памяти, он может быть задержан, пока память будет занята DMA-передачей. Используя кэш-память, процессор в большинстве случаев может избежать необходимости ожидания получения доступа к памяти, оставляя основную

Прямой доступ к памяти (DMA)
Механизм, предоставляющий контроллер устройства с возможностью передачи данных непосредственно в память или из памяти без участия процессора.

Ведущее устройство
Устройство связи ввода-вывода, способное инициировать запросы на передачу данных.

часть пропускной способности памяти свободной для использования устройствами ввода-вывода.

Уточнение. Для дальнейшего сокращения необходимости прерывания процессора и задействования его в обработке запросов на ввод-вывод, которые могут потребовать сразу несколько текущих операций, контроллер ввода-вывода может быть сделан более разумным. Такие контроллеры часто называют *процессорами ввода-вывода* (а также *контроллерами ввода-вывода* или *контроллерами канала*). Эти специализированные процессоры в основном исполняют серии операций ввода-вывода, называемые *программами ввода-вывода*. Программа может храниться в процессоре ввода-вывода или же она может храниться в памяти и извлекаться из нее процессором ввода-вывода. При использовании процессора ввода-вывода операционная система обычно настраивает программу ввода-вывода, в которой прописываются исполняемые операции ввода-вывода, а также размер и адрес передачи для любых операций чтения или записи. Затем процессор ввода-вывода извлекает операции из программы ввода-вывода и прерывает работу основного процессора только по завершении всей программы. DMA-процессоры по сути представляют собой процессоры специального назначения (обычно непрограммируемые и исполняемые на одном кристалле), а процессоры ввода-вывода часто реализуются с помощью микропроцессора общего назначения, выполняющего специализированную программу ввода-вывода.

Прямой доступ к памяти и система памяти

Когда DMA встроен в систему ввода-вывода, отношения между системой памяти и процессором изменяются. Без DMA все обращения к системе памяти исходят из процессора и поэтому обрабатываются путем преобразования адреса и обращения к кэш-памяти, как будто процессор сгенерировал ссылки. При наличии DMA появляется еще один путь к системе памяти, который не проходит через механизм преобразования адреса или через иерархию кэш-памяти. Это отличие создает ряд проблем как в системах с виртуальной памятью, так и в системах с устройствами кэш-памяти. Эти проблемы обычно решаются с помощью сочетания аппаратных технологий и программной поддержки.

Сложности, создаваемые DMA в системе виртуальной памяти, возникают из-за того, что страницы имеют как физические, так и виртуальные адреса. DMA также создает проблемы для систем, использующих устройства кэш-памяти, потому что могут быть две копии элемента данных: один в кэш-памяти, а другой в оперативной памяти. Поскольку DMA-процессор выдает запросы к памяти напрямую, а не через кэш-память основного процессора, значения в том месте памяти, которое видит блок DMA и процессор, могут различаться. Рассмотрим чтение с диска, результаты которого блок DMA помещает непосредственно в память. Если некоторые из мест, в которые блок DMA ведет запись, находятся в кэш-памяти, то процессор при чтении получит старое значение. Точно так же, если кэш-память использует отложенную запись, блок DMA может читать значение непосредственно из памяти, в то время как более свежее значение находится в кэш-памяти, и это значение еще не возвращено в оперативную память. Это называется *проблемой выбрасывания устаревших данных*, или *проблемой поддержки целостности данных в кэш-памяти* (см. главу 5).

Мы рассмотрели три разных метода передачи данных между устройством ввода-вывода и памятью. При переходе от опроса к управлению прерываниями и к интерфейсу DMA мы перенесли ответственность за управление операциями ввода-вывода с процессора на более «умный» контроллер ввода-вывода. Преимущество этих методов в том, что они освобождают процессор. Их недостаток состоит в том, что они повышают стоимость системы ввода-вывода. Поэтому для отдельно взятой компьютерной системы должен быть сделан выбор, какой метод из предлагаемого спектра больше всего подойдет для подключенных к нему устройств ввода-вывода.

Перед тем как рассмотреть конструкцию систем ввода-вывода, давайте в следующем разделе взглянем на оценку их производительности.

Самопроверка

Какое из утверждений относительно выстраивания трех способов осуществления ввода-вывода соответствует истине?

1. Если нужно получить наименьшую латентность для операции ввода-вывода применительно к отдельному устройству ввода-вывода, способы должны располагаться в следующем порядке: опрос, использование DMA и использование управления с помощью прерываний.
2. С точки зрения наименьшего востребования процессора одним устройством ввода-вывода способы должны располагаться в следующем порядке: использование DMA, использование управления с помощью прерываний и опрос.

Интерфейс аппаратного и программного обеспечения

С чем в системах с виртуальной памятью должен работать DMA — с виртуальными или с физическими адресами? Очевидная проблема с виртуальными адресами состоит в том, что блоку DMA придется проводить преобразование виртуальных адресов в физические. Основной проблемой использования физических адресов в DMA-передаче является невозможность простого пересечения границ страницы. Если запрос на ввод-вывод пересекает границу страницы, то место в памяти, куда осуществлялась передача, может не сохранить целостность в виртуальной памяти. Следовательно, при использовании физических адресов нужно ограничивать все DMA-передачи в пределах одной страницы.

Метод, позволяющий системе инициировать DMA-передачи, пересекающие границы страниц, заключается в том, чтобы заставить DMA работать с виртуальными адресами. В такой системе блок DMA имеет небольшое количество записей отображений, предоставляющих отображение виртуальных адресов на физические для передачи. Операционная система предоставляет отображение при инициировании ввода-вывода. Благодаря этому отображению, блоку DMA не нужно беспокоиться о месте нахождения виртуальной страницы, вовлеченной в передачу.

Еще одна технология для операционной системы заключается в разбиении DMA-передачи на серии передач, каждая из которых помещается внутри отдельной

физической страницы. Затем передачи *выстраиваются в одну цепочку* и вручаются процессору ввода-вывода или «умному» блоку DMA, который выполнит всю последовательность передач, или же операционная система может выдать отдельные запросы на передачу.

Какой бы метод ни использовался, операционная система должна организовать взаимодействие таким образом, чтобы страницы не отображались заново, пока страница еще используется в DMA-передаче.

Интерфейс аппаратного и программного обеспечения

От проблем целостности данных ввода-вывода можно избавиться с помощью одной из трех главных технологий. Во-первых, можно провести все действия по вводу-выводу через кэш-память. Тогда при операциях чтения будет гарантировано получение последнего значения, а при операциях записи будут обновлены любые данные в кэш-памяти. Проводить весь ввод-вывод через кэш-память очень дорого, и потенциально это может весьма негативно сказаться на производительности процессора, потому что данные ввода-вывода редко нужны для немедленного использования и могут заменить собой полезные данные, необходимые выполняемой программе. Во-вторых, можно заставить операционную систему выборочно забраковать данные кэш-памяти при чтении ввода-вывода или заставить выполнить отложенные операции записи при записи ввода-вывода (что часто называют *сбросом* кэш-памяти — cache flushing). Такой подход требует небольшой аппаратной поддержки и, возможно, он будет более эффективен, если эта функция сможет быть легко и непринужденно выполнена программным обеспечением. Поскольку этот сброс больших частей кэш-памяти нужен только при обращениях блока DMA, он будет происходить сравнительно редко. В-третьих, можно предоставить аппаратный механизм для выборочного сброса (или установки недостоверности) элементов кэш-памяти. Аппаратная установка недостоверности с целью обеспечения целостности кэш-памяти часто применяется в микропроцессорных системах, и такая же технология может быть использована для ввода-вывода; более подробно эта тема раскрыта в главе 5.

6.7. Оценки производительности ввода-вывода: примеры, связанные с дисками и файловыми системами

Как можно сравнивать системы ввода-вывода? Это непростой вопрос, потому что производительность ввода-вывода зависит от многих аспектов системы, и разные приложения выводят на первый план разные аспекты системы ввода-вывода. Более того, в конструкции могут быть заложены сложные компромиссы между временем отклика и пропускной способностью, которые не дают оценить отдельно взятый аспект. Например, наискорейшая обработка запроса обычно минимизирует время отклика, в то время как хорошие показатели пропускной способности достигаются

в том случае, если мы стремимся обработать взаимосвязанные запросы вместе. Получается, что мы можем поднять пропускную способность при работе с диском путем группировки запросов, обращающихся к тем местам, которые расположены близко друг к другу. Но такой подход увеличит для некоторых запросов время отклика, что может привести к большому разбросу в данном показателе. Хотя пропускная способность повысится, некоторые контрольные задачи констатируют максимальное время отклика для любого запроса, выявляя потенциальную проблематичность подобных способов оптимизации.

В данном разделе будут приведены некоторые примеры оценок, предлагаемые для определения производительности систем хранения информации. Эти контрольные задачи испытывают влияние различных свойств системы, включая технологию дисков, способ их подключения, систему памяти, процессор и предоставляемую операционной системой файловую систему.

Перед рассмотрением этих контрольных задач нужно разобраться в терминологии и единицах измерения. Производительность систем ввода-вывода зависит от скорости, с которой система передает данные. Скорость передачи данных зависит от тактовой частоты, которая обычно задается в гигагерцах (ГГц) = 10^9 циклов в секунду. Скорость передачи обычно задается в гигабайтах в секунду (Гбайт/с). В системах ввода-вывода гигабайты (Гбайт) измеряются с использованием основания 10 (то есть 1 Гбайт = 10^9 = 1 000 000 000 байт), в отличие от оперативной памяти, где используется основание 2 (то есть 1 Гбайт = 2^{30} = 1 073 741 824 байт). Это различие требует конвертации систем счисления с основанием 10 (1 К = 1000) и 2 (1 К = 1024), поскольку многие обращения ввода-вывода касаются блоков данных, имеющих размер, кратный степеням числа 2. Чтобы не усложнять все наши примеры точной конвертацией одного из двух измерений, мы обозначили эту разницу здесь и констатировали тот факт, что трактовка двух измерений как имеющих одинаковые единицы будет небольшой ошибкой. Эта ошибка будет проиллюстрирована в разделе 6.11.

Контрольные задачи, связанные с обработкой транзакций

Приложения, связанные с **обработкой транзакций** (transaction processing, TP), включают в себя как требования к времени отклика, так и оценку производительности, основанную на пропускной способности. Кроме этого, большинство обращений ввода-вывода связаны с небольшими объемами данных. Поэтому TP-приложения в основном заинтересованы в **скорости ввода-вывода**, измеренной в количествах обращений в секунду, в отличие от **скорости передачи данных**, измеряемой в байтах данных в секунду. TP-приложения обычно связаны с из-

Обработка транзакций
Разновидность приложения, занимающегося выполнением небольших коротких операций (называемых транзакциями), которые обычно требуют как ввода-вывода данных, так и вычислений. К приложениям по обработке транзакций обычно предъявляются требования как по времени отклика, так и по пропускной способности транзакций.

Скорость ввода-вывода
Производительность, измеренная количеством операций ввода-вывода в единицу времени, например количеством чтений в секунду.

Скорость передачи данных
Производительность, измеренная в байтах в единицу времени, например в гигабайтах в секунду.

менениями больших баз данных, в системе, которая должна отвечать определенным требованиям по времени отклика, а также иметь возможность точной обработки отказов определенного типа. Эти приложения очень важны, а несоблюдение предъявляемых к ним требований может привести к существенным убыткам. Например, банки обычно используют системы обработки транзакций, поскольку в них уделяется внимание широкому спектру характеристик. Сюда включаются гарантии, исключающие утрату транзакций, быстрая обработка транзакций и минимизация стоимости обработки каждой транзакции. Хотя абсолютным требованием, предъявляемым к таким системам, является гарантия безопасности в случае отказов, для создания рентабельных систем важными факторами являются также и время отклика, и пропускная способность.

Существует несколько контрольных задач обработки транзакций. Известным набором контрольных задач является серия, разработанная советом по обработке транзакций — Transaction Processing Council (TPC).

Набор TPC-C, первоначально созданный в 1992 году, имитирует среду сложных запросов. Набор TPC-H моделирует поддержку произвольных решений — запросы не связаны друг с другом, и сведения о последних запросах не могут быть использованы для оптимизации будущих запросов; в результате время выполнения запросов может быть весьма продолжительным. Набор TPC-W является контрольной задачей транзакций, осуществляемых через Интернет, имитирующей работу веб-сервера по обработке коммерческих транзакций. В нем применяется система базы данных, а также базовое программное обеспечение веб-сервера. Набор TPC-App включает контрольные задачи для сервера приложения и веб-служб. Самый последний набор TPC-E имитирует рабочую нагрузку по обработке транзакций брокерской фирмы. Описание контрольных задач TPC дается на сайте www.tpc.org.

Все контрольные задачи TPC оценивают производительность в транзакциях в секунду. Кроме этого в них включены требования по времени отклика, поэтому производительность, связанная с пропускной способностью, оценивается только при выполнении ограничений по времени отклика. Чтобы смоделировать реальную систему, более высокие скорости транзакций также связаны с более крупными системами, как в понятиях количества пользователей, так и в понятиях размера базы данных, к которой применяются транзакции. Поэтому производительности должна соответствовать емкость запоминающего устройства. И наконец, чтобы позволить точное сравнение по соотношению цена—производительность, в систему контрольных задач должна быть также включена и стоимость системы.

Файловая система и контрольные задачи ввода-вывода через Интернет

Кроме контрольных задач по оценке работы процессора, SPEC предлагает также контрольные задачи по оценке производительности файлового сервера (SPECSFS) и контрольные задачи по оценке производительности веб-сервера (SPECWeb). SPECSFS является контрольной задачей для оценки производительности сетевой файловой системы — NFS (Network File System) с использованием сценария запро-

сов к файловому серверу; она тестирует производительность системы ввода-вывода, включая как дисковый, так и сетевой ввод-вывод, а также производительность процессора. Контрольная задача SPECSFS ориентирована на оценку пропускной способности, но с учетом важности требований ко времени отклика. Контрольная задача SPECWeb предназначена для оценки производительности веб-сервера и имитирует работу нескольких клиентов, запрашивающих с сервера как статические, так и динамические страницы, а также клиентов, отправляющих данные на сервер (см. главу 1).

Самые последние контрольные задачи SPEC нацелены на оценку потребляемой мощности. Контрольная задача SPECPower оценивает потребляемую мощность и характеристики производительности небольших серверов.

Компания Sun недавно представила *filebench*, среду контрольных задач по оценке работы файловой системы. Вместо стандартной рабочей нагрузки эта среда предоставляет язык, позволяющий дать описание рабочей нагрузки, требуемой для вашей файловой системы. Но в ней имеются и примеры рабочих нагрузок файловой системы, предназначенные для имитации распространенных вариантов использования файловой системы.

Самопроверка

Являются ли следующие утверждения истинными или ложными? В отличие от контрольных задач по оценке производительности процессоров, контрольные задачи ввода-вывода:

1) сконцентрированы на пропускной способности, а не на латентности;
2) могут потребовать масштабирования размера набора данных или количества пользователей для достижения контрольных точек производительности;
3) часто сообщают о стоимости достижения производительности.

6.8. Разработка систем ввода-вывода

Существует два основных типа технических требований, с которыми сталкиваются разработчики систем ввода-вывода: ограничения по латентности и ограничения по пропускной способности. В обоих случаях на конструкцию и оценки влияют знания о модели трафика.

Ограничения по латентности включают гарантии, что латентность завершения операции ввода-вывода ограничена определенным показателем. В простом случае система может быть не нагружена, и конструктор должен гарантировать выполнение некоторых границ латентности либо из-за их критичности для приложения, либо из-за того, что приложение должно получить определенное гарантированное обслуживание, чтобы предупредить появление ошибок. Кроме того, определение латентности на ненагруженной системе дается относительно легко, поскольку оно включает отслеживание путей операции ввода-вывода и суммирование отдельных показателей латентности.

Определение средней латентности (или распределения латентности) под нагрузкой дается намного сложнее. Такие задачи решаются либо с помощью теории массового обслуживания (когда поведение запросов рабочей нагрузки и показатели времени обслуживания ввода-вывода могут быть приблизительно оценены с помощью простых распределений), либо путем имитационного моделирования (при сложном стечении обстоятельств ввода-вывода). Обе эти темы не вписываются в рамки данного текста.

Еще одной типичной проблемой, с которой сталкиваются разработчики, является конструирование системы ввода-вывода, отвечающей набору ограничений пропускной способности при заданной рабочей нагрузке. Кроме этого разработчику может быть передана частично сконфигурированная система ввода-вывода и поставлена задача по балансированию системы с целью поддержания максимально достижимой полосы пропускания, продиктованной предварительно сконфигурированной частью системы. Эта последняя задача конструирования является упрощенной версией первой задачи.

При разработке таких систем используются следующие общие подходы:

1. Поиск в системе ввода-вывода самого слабого звена, представляющего собой компонент пути ввода-вывода, который накладывает ограничения на конструкцию. В зависимости от рабочей нагрузки, этот компонент может находиться в любом месте, включая процессоры, систему памяти, контроллеры ввода-вывода или устройства. Место самого слабого звена может быть продиктовано как рабочей нагрузкой, так и ограничениями конфигурации.
2. Настройка этого компонента для поддержки требуемой пропускной способности.
3. Определение требований для всей остальной системы и настройка ее на поддержку этой пропускной способности.

Проще всего освоить эту методику на примерах. Простой анализ системы ввода-вывода сервера Sun Fire x4150, показывающий работу этой методики, будет сделан в разделе 6.10.

6.9. Параллелизм и ввод-вывод: избыточные массивы недорогих дисков

Закон Амдала в главе 1 говорил, насколько опрометчиво пренебрегать вводом-выводом в сегодняшней революции параллельных вычислений. Это можно продемонстрировать на простом примере.

Упражнение

Влияние ввода-вывода на производительность системы

Предположим, у нас есть контрольная задача, выполняемая в целом за 100 секунд, из которых 90 секунд составляет время работы центрального процессора, а остальное время затрачивается на ввод-вывод. Предположим, что количество процессоров удваивается каждые

ма года, но скорость остается прежней, а время ввода-вывода не улучшается. Насколько быстрее будет выполняться наша программа через шесть лет?

Ответ

Известно, что

```
Общее время = время центрального процессора + время ввода-вывода
            100 = 90 + время ввода-вывода
Время ввода-вывода = 10 секунд
```

Новые показатели времени работы центрального процессора и получающееся при этом вычисленное общее время приведены в следующей таблице.

По прошествии *n* лет	Время центрального процессора, с	Время ввода-вывода, с	Общее время	Время ввода-вывода, %
0	90	10	100 с	10
2	90/2=45	10	55 с	18
4	45/2=23	10	33 с	31
6	23/2=11	10	21 с	47

Улучшение производительности центрального процессора после шести лет равно

```
90 / 11 = 8
```

А улучшение общего времени составляет всего лишь

```
100 / 21 = 4.7
```

И доля времени на ввод-вывод увеличилась с 10% до 47% общего времени.

Следовательно, революция параллельных вычислений должна прийти в системы ввода-вывода, так же, как она пришла в вычисления, или же усилия, затраченные на параллелизацию, могут уйти впустую, если программы выполняют необходимые им операции ввода-вывода.

Увеличение производительности ввода-вывода стало исходной мотивацией создания дисковых массивов. В конце 1980-х годов из высокопроизводительных устройств хранения информации можно было выбрать большие, дорогостоящие диски, такие как один из самых больших, показанных на рис. 6.2. Аргумент состоял в том, что, заменив несколько больших дисков множеством небольших, можно улучшить производительность, поскольку в этом случае будет больше считывающих головок. Это изменение вполне соответствует множеству процессоров, поскольку наличие множества головок чтения-записи означает, что система хранения может поддерживать намного больше независимых обращений, а также длинные передачи, данные для которых разбросаны по нескольким дискам. То есть вы можете получить как большее количество операций ввода-вывода в секунду, так и более высокую скорость передачи данных. Вдобавок к высокой производительности может быть получено преимущество в стоимости, потребляемой мощности и пространстве размещения, поскольку диски меньших размеров имеют, как правило, более высокую эффективность на гигабайт, чем более крупные диски.

В качестве отрицательного аргумента утверждалось, что дисковые массивы могут снизить надежность. Эти небольшие, недорогие накопители имеют более

низкие уровни MTTF, чем большие накопители, но, что более важно, за счет замены одного накопителя, скажем, пятьюдесятью небольшими накопителями интенсивность отказов возрастает как минимум в 50 раз!

Решением стало добавление избыточности, чтобы система могла справляться с отказами дисков без потери информации. При наличии множества небольших дисков расходы на дополнительную избыточность для повышения готовности невелики относительно расходов, которые требуются для решений, связанных с применением небольшого количества крупных дисков. Таким образом, безотказность была более доступной по цене при создании избыточного массива недорогих дисков. Это наблюдение легло в основу его названия: **избыточные массивы недорогих дисков** – redundant arrays of inexpensive disks, сокращенно **RAID**.

В ретроспективе, хотя их изобретение было мотивировано производительностью, главной причиной широкой популярности RAID-массивов стала безотказность. Параллельная революция вновь выдвинула на первый план исходный аргумент производительности RAID-массивов. В остальной части раздела дается краткий обзор возможных вариантов достижения безотказности и их влияния на стоимость и производительность.

Рис. 6.5. RAID-массив для примера с четырьмя дисками данных, в котором показываются дополнительные контрольные диски для каждого уровня RAID и компании, использующие каждый уровень. На рис. 6.6 и 6.7 объясняется разница между RAID 3, RAID 4 и RAID 5

Какой уровень безотказности вам нужен? Нужна ли вам дополнительная информация для обнаружения отказов? Играет ли какую-нибудь роль порядок организации данных и дополнительная проверочная информация на этих дисках? В статье, где предлагалось название, дан развернутый ответ на эти вопросы, который начинается с самого простого, но наиболее дорогостоящего решения.

На рис. 6.5 показано развитие RAID-массивов и их примерная стоимость, выраженная в количестве дополнительных контрольных дисков. Чтобы отслеживать развитие RAID-массивов, авторы пронумеровали стадии этого развития, и эти номера используются по сей день.

Без избыточности (RAID 0)

Простое распределение данных по нескольким дискам, называемое **чередованием** (striping), автоматически вынуждающее обращаться к нескольким дискам. Разделение по набору дисков заставляет всю коллекцию появляться для программного обеспечения в виде одного большого диска, что упрощает управление хранением данных. Оно также повышает производительность больших обращений, поскольку одновременно может работать сразу несколько дисков. К примеру, системы редактирования видео часто чередуют свои данные и не слишком заботятся об отказоустойчивости по сравнению с базами данных.

Название RAID 0 употребляется неправомерно, поскольку в этом массиве отсутствует избыточность. Но уровни RAID-массивов часто указываются оператору для настройки, и RAID 0 часто присутствует в качестве одного из вариантов. Этим и объясняется широкое распространение названия RAID 0.

Зеркальное копирование (RAID 1)

Эта традиционная схема, допускающая отказ диска, называется **зеркалированием** или *экранированием* и использует в два раза больше дисков, чем RAID 0. Как только данные записываются на один диск, эти же данные также записываются на избыточный диск, чтобы всегда было две копии информации. Если диск отказывает, система просто переходит на «зеркало» и считывает его содержимое для получения желаемой информации. Зеркалирование является наиболее затратным RAID-решением, поскольку для него требуется наибольшее количество дисков.

Избыточные массивы недорогих дисков (RAID)
Организация дисков, использующих массив небольших и недорогих дисков с целью повышения производительности и надежности.

Чередование
Распределение логически последовательных блоков по разным дискам для достижения более высокой производительности по сравнению с той, что предоставляется отдельным диском.

Зеркалирование
Запись одних и тех же данных на несколько дисков для повышения готовности данных.

Обнаружение ошибок и корректирующий код (RAID 2)

Массив RAID 2, перенимающий схему обнаружения и исправления ошибок, наиболее часто

используемую для устройств памяти. Поскольку в настоящее время RAID 2 уже не используется, его описание здесь не приводится.

Четность с чередованием битов (RAID 3)

Стоимость более высокой готовности может быть снижена на $1/n$, где n — это количество дисков в **группе защиты**. Вместо хранения полной копии исходных данных для каждого диска нужно добавить лишь строго необходимое количество избыточной информации для восстановления утраченных при отказе данных. В операциях чтения или записи участвуют все диски группы и еще один диск, хранящий контрольную информацию на случай отказа. Массив RAID 3 популярен в приложениях с большими наборами данных, например в мультимедийных и некоторых научных программах.

Четность является одной из таких схем. Читатели, не знакомые с четностью, могут представить себе избыточный диск как хранилище суммы всех данных, имеющихся на других дисках. При отказе диска все данные из исправных дисков вычитаются из данных диска четности; оставшаяся информация должна быть идентична утраченной. Четность — это просто сумма по модулю два.

В отличие от RAID 1, для определения утраченных данных должны быть прочитаны несколько дисков. В основе этой технологии лежит предположение о приемлемости компромисса между более продолжительным временем восстановления после отказа и меньшими затратами на хранение избыточных данных.

Четность с чередованием блоков (RAID 4)

Массив RAID 4 использует такое же соотношение дисков с данными и контрольных дисков, как и RAID 3, но доступ к данным у них осуществляется по-разному. Информация о четности хранится в виде блоков и связана с набором блоков данных.

В RAID 3 каждое обращение направляется на все диски. Но некоторые приложения предпочитают небольшие обращения, допуская параллельное осуществление независимого доступа. В этом и заключается цель создания массивов RAID уровней от 4 до 6. Поскольку для определения корректности данных при чтении каждого сектора проверяется информация обнаружения ошибок, такие «небольшие считывания» с каждого диска могут происходить независимо, потому что минимально доступен один сектор. В среде RAID-массива небольшое обращение относится только к одному диску из группы защиты, а большое обращение относится ко всем дискам этой группы.

При записи все обстоит по-другому. Может показаться, что каждая небольшая запись будет требовать доступа ко всем остальным дискам, чтобы прочитать всю информацию, необходимую для нового вычисления контроля четности, как в левой части рис. 6.6. «Небольшая запись» потребует чтения старых данных и старого значения контроля четности с добавлением новой информации.

Группа защиты
Группа дисков с данными или блоки, использующие общий контрольный диск или блок.

затем она потребует записи нового значения четности на избыточный диск и новых данных на диск с данными.

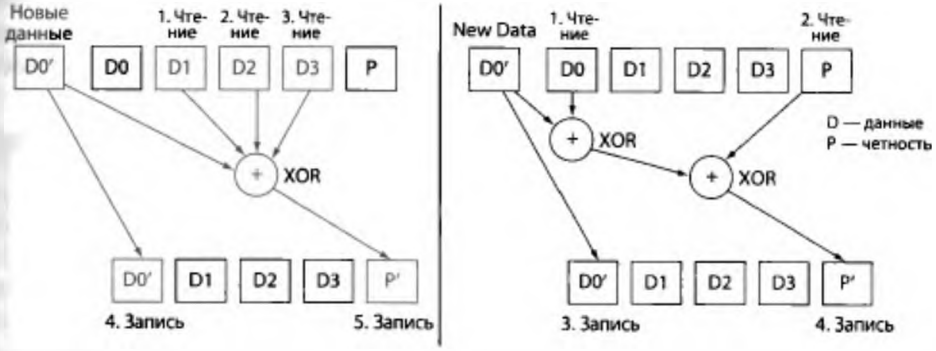


Рис. 6.6. Обновление в связи с небольшой записью в массиве RAID 3 по сравнению с таким же обновлением в массиве RAID 4. Эта оптимизация для небольших записей сокращает количество обращений к дискам, а также количество задействованных дисков. На данном рисунке предполагается, что используются четыре блока данных и один блок четности. Обычное вычисление четности в массиве RAID 3 в левой части рисунка требует чтения блоков D1, D2 и D3 перед добавлением блока D0' для вычисления нового значения четности P'. (Здесь нет ничего удивительного, новые данные D0' поступают прямо из центрального процессора, поэтому диски в их чтении не задействованы.) Сокращение операций в массиве RAID 4, изображенное в правой части рисунка, предусматривает чтение старого значения D0, которое сравнивается с новым значением D0', чтобы узнать, какие биты изменились. Затем считывается старое значение четности P, после чего изменяются соответствующие биты для получения значения P'. Логическая функция исключающее ИЛИ (XOR) подходит для этого как нельзя лучше. В данном примере три чтения дисков (D1, D2, D3) и две записи на диски (D0', P'), в которых были задействованы все диски, заменены двумя чтениями дисков (D0, P) и двумя записями на диски (D0', P'), в которых задействуются всего два диска. Увеличение размера группы четности влечет за собой экономию за счет сокращения числа операций. Такое же сокращение используется и в массиве RAID 5

Ключевой догадкой, позволившей сократить издержки, явилось то, что четность — это просто сумма данных; если отслеживать изменившиеся биты при записи новой информации, то потребуется изменить лишь соответствующие биты на диске четности. Сокращение количества операций показано в правой части рис. 6.6. Нужно прочитать старые данные с диска, на который будет вестись запись, сравнить их с новыми данными, чтобы узнать, какие биты изменились, прочитать старое значение четности, изменить соответствующие биты, а затем записать новые данные и новое значение четности. Таким образом, небольшая запись повлечет за собой четыре обращения к двум дискам вместо обращения ко всем дискам. Такая организация используется в массиве RAID 4.

Четность с распространенным чередованием блоков (RAID 5)

В массиве RAID 4 эффективно поддерживаются разные сочетания больших чтений, больших записей и небольших чтений, плюс к этому имеется возможность небольших записей. Небольшим недостатком этой системы является необходи-

мость обновления содержимого диска четности при каждой записи, поэтому диск четности становится узким местом для последовательных записей.

Чтобы устранить узкое место, связанное с записью значений четности, информация о четности может быть распространена по всем дискам, чтобы не было единого узкого места для записей. Организация распространенной информации о четности является особенностью массива RAID 5.

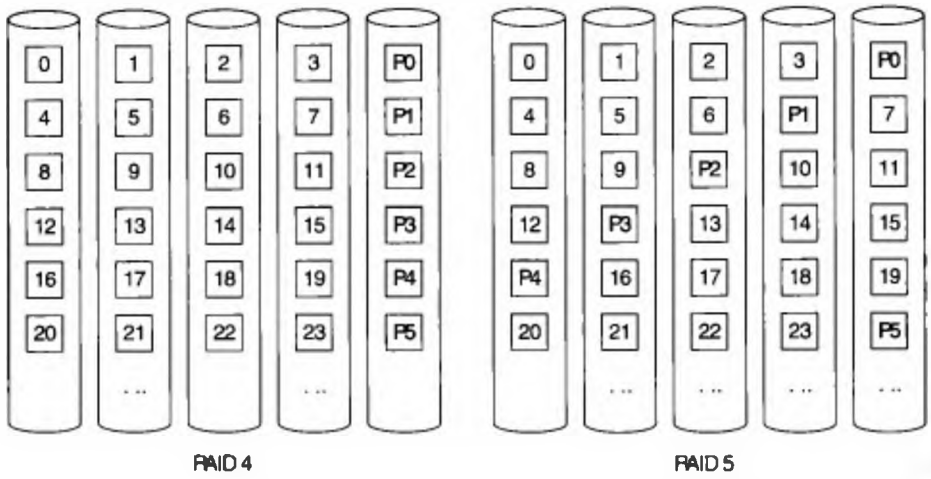


Рис. 6.7. Четность с чередованием блоков (RAID 4) в сравнении с четностью с распространенным чередованием блоков (RAID 5). За счет распространения блоков четности по всем дискам некоторые небольшие записи могут вестись параллельно

На рис. 6.7 показано, как данные распространяются в массиве RAID 4 по сравнению с массивом RAID 5. Как показано в организации справа, в массиве RAID 5 четность связана с каждой строкой блоков данных и больше не ограничивается размещением на одном диске. Такая организация позволяет нескольким записям вестись одновременно, поскольку блоки четности больше не размещаются на одном и том же диске. Например, запись в блок 8 в правой части рисунка должна также обращаться и к его блоку четности P2, задействуя таким образом первый и третий диски. Вторая запись в блок 5 в правой части рисунка предполагает обновление его блока четности P1, обращаясь ко второму и четвертому дискам, и поэтому она может вестись параллельно с записью в блок 8. Точно такие же записи при организации, показанной в левой части рисунка, приведут к изменениям в блоках P1 и P2, а оба этих блока расположены на пятом диске, что и составляет узкое место такой организации.

P + Q избыточность (RAID 6)

Схемы, основанные на вычислении четности, защищают от единичного самоопределяющегося отказа. Когда проведения одной ликвидации последствий отказа недостаточно, четность может быть распространена, чтобы можно было иметь возможность второго вычисления данных и еще один диск с контрольной информа-

шей. Второй контрольный блок позволяет восстановить информацию при втором отказе. Поэтому издержки хранения информации по сравнению с массивом RAID 5 удваиваются. Сокращение издержек небольшой записи, показанное на рис. 6.6, работает точно так же, за исключением того, что теперь происходит шесть обращений к дискам вместо четырех для обновления информации как в P, так и в Q блоке.

Краткие выводы по RAID-массивам

Массивы RAID 1 и RAID 5 широко используются на серверах: по одной из оценок 50% дисков на серверах имеют RAID-организацию.

Одним из слабых мест RAID-систем является ремонт. Во-первых, чтобы избежать недоступности данных во время ремонта, массив должен быть сконструирован так, чтобы отказавшие диски можно было заменять, не выключая системы. У RAID-массивов достаточно избыточности, чтобы позволить им работать в непрерывном режиме, но **горячая замена** дисков накладывает определенные требования на физическую и электрическую конструкцию массива и на интерфейсы, используемые при работе с дисками. Во-вторых, во время ремонта может произойти еще один отказ, поэтому время ремонта влияет на шансы потери данных: чем дольше ремонт, тем выше шансы еще одного отказа, который приведет к потере данных. Чтобы исключить вынужденное ожидание, пока оператор принесет исправный диск, в некоторые системы включаются **ожидающие подключения резервы**, чтобы данные могли быть тут же реконструированы, как только обнаружится отказ. Затем оператор без лишней спешки может заменить отказавшие диски. Обратите внимание, что решение, какие диски нужно заменить, принимает оператор. Как показано в табл. 6.2, в роли оператора выступают люди, поэтому если они случайно удалят вместо неисправного исправный диск, это приведет к невосстанавливаемому отказу диска.

Помимо вопросов конструирования пригодной к ремонту RAID-системы, возникают вопросы о том, как как с течением времени изменяется технология дисковых устройств. Хотя производители дисков заявляют об очень высоких показателях MTTF для своих продуктов, все эти показатели соответствуют вполне определенным условиям эксплуатации. Если отдельно взятый дисковый массив подвергся температурным перепадам, скажем, из-за поломки системы кондиционирования воздуха, или же он подвергся ударным нагрузкам из-за недостатков в конструкции или установки аппаратной стойки, интенсивность отказов может увеличиться в 3–6 раз (см. заблуждение в разделе 6.11). Вычисление готовности RAID-системы предполагает независимость отказов дисков, но эти отказы могут быть связаны друг с другом, поскольку повреждения из-за эксплуатационной среды вполне вероятно могут коснуться всех дисков массива. Другая проблема возникает в связи с тем, что пропускная способность дисков растет намного медленнее, чем их емкость, время на ремонт диска в RAID-системе возрастает, что в свою очередь вызывает рост шансов на повторный отказ. Например, последовательное чтение

Горячая замена
Замена компонента оборудования на работающей системе.

Резервы, ожидающие подключения
Резервные ресурсы оборудования, способные немедленно заменить вышедший из строя компонент.

SATA-диска емкостью 1000 Гбайт, если не будет никаких помех, может занять почти три часа. Учитывая то, что поврежденный RAID-массив, скорее всего, продолжает обслуживать данные, реконструкция может существенно затянуться. Кроме увеличения этого времени возникает также опасение, что чтение существенно большего объема данных в процессе реконструкции будет означать увеличение шансов на невосстанавливаемый отказ чтения носителя, который может повлечь за собой утрату данных. Еще одной проблемой, связанной с одновременно возникающими отказами, является рост числа дисков в массиве и использование SATA-дисков, работающих медленнее и имеющих более высокую емкость по сравнению с обычными серверными дисками.

Следовательно, потребителей привлекает защита сразу от нескольких отказов, и поэтому возрастают предложения массивов RAID 6 в качестве дополнений и в качестве основных устройств.

Самопроверка

Что из следующих утверждений справедливо для RAID-массивов уровней 1, 3, 4, 5 и 6?

1. Для достижения высокой готовности RAID-системы полагаются на избыточность информации.
2. RAID 1 (зеркалирование) имеет самые большие накладные расходы, связанные с контрольными дисками.
3. Для небольших записей RAID 3 (четность с чередованием битов) имеет наихудшую пропускную способность.
4. Для больших записей RAID 3, 4 и 5 имеют одинаковую пропускную способность.

Уточнение. Возникает вопрос, как зеркалирование взаимодействует с чередованием. Предположим, что у вас есть, скажем, четыре диска, на которых хранятся ценные данные и восемь физических дисков, которыми можно воспользоваться. Станете ли вы создавать четыре пары дисков, каждая из которых организована как RAID 1, а затем чередовать данные между этими четырьмя парами RAID 1? Или вы будете создавать два набора из четырех дисков, каждый из которых организован как RAID 0, а затем зеркалировать записи для обоих наборов RAID 0? В развитие терминологии RAID первое сочетание стало называться RAID 1 + 0, или RAID 10 («чередующееся зеркалирование»), а последнее стало называться RAID 0 + 1, или RAID 01 («зеркалированное чередование»).

6.10. Реальное оборудование: сервер Sun Fire x4150

Вдобавок к революции в конструкции микропроцессоров мы наблюдаем революцию в доставке программного обеспечения. Вместо традиционной продажи программ на компакт-дисках или их доставке по Интернету для установки на вашем компьютере появилась альтернатива — *программное обеспечение в виде оказываемой услуги (Software as a Service, SaaS)*. То есть вы по Интернету переходите на ком-

пьютер, предоставляющий услугу в виде уже запущенной, требующейся вам программы, с помощью которой и проделывается вся работа. Наиболее популярным примером будет, наверное, веб-поиск, но есть службы для редактирования и хранения фотографий, обработки документов, хранения баз данных, предоставления виртуальных миров и т. д. Если как следует поискать, можно найти сервисную версию практически любой программы, используемой на вашем настольном компьютере.

Это изменение привело к созданию больших центров обработки данных (дата-центров) с компьютерами и дисками для запуска служб, используемых миллионами внешних пользователей. На что должны быть похожи компьютеры, если они разрабатываются для работы в этих огромных дата-центрах? Конечно же, им не нужны дисплеи и клавиатуры. Понятно, что если в дата-центре 10 000 таких компьютеров, то к традиционным вопросам стоимости и производительности добавятся вопросы эффективного использования пространства и энергопотребления.

В связи с этим возникает вопрос, а как должны выглядеть хранилища данных в дата-центре? При всем богатстве вариантов одна из популярных версий состоит в создании строительного блока, состоящего из сборки дисков, процессора и памяти. Чтобы отбросить сомнения насчет надежности, сами приложения создают избыточные копии и отвечают за сохранение их целостности и восстановление после отказов.

IT-промышленность во многом пошла по пути соблюдения ряда стандартов, касающихся физической конструкции компьютеров для дата-центров, особенно это касается стоек, в которых должны находиться компьютеры таких центров. Наибольшую популярность приобрела стойка шириной 19 дюймов (486,2 мм). Компьютеры, разработанные для стойки, так и называются — *монтируемыми в стойке* (rack mount), но их также называют *секцией стойки*, или просто *полкой*. Поскольку традиционное посадочное место, в которое монтируются полки, имеет высоту 1,75 дюйма (44,45 мм), это расстояние обычно называется *единицей высоты устройства, монтируемого в стойку* (rack unit), или просто *единицей* (unit, U). Наиболее популярная 19-дюймовая стойка имеет высоту 42 U, что составляет 42 × 1,75, или 73,5 дюйма. Глубина полок может быть разной.

Следовательно, наименьший компьютер, предназначенный для монтажа в стойку, имеет ширину 19 дюймов и высоту 1,75 дюйма, его часто называют 1U компьютером или 1U сервером. Из-за размеров их прозвали *коробками с пиццей* («pizza box»). На рис. 6.8 показан пример стандартной стойки, в которой находятся 42 1U сервера.

На рис. 6.9 в качестве примера 1U сервера показан Sun Fire x4150. В максимальной конфигурации этот 1U сервер содержит:

- 82,66 ГГц процессоров, распределенных по двум гнездам (2 Intel Xeon 5345);
- 64 Гбайт DRAM-памяти DDR2-667, распределенной по 164 Гбайт FBDIMM-блокам;
- 873 Гбайт 2,5-дюймовых дисковых SAS-накопителей со скоростью вращения дисков 15 000 об/мин.;
- 1 RAID-контроллер (поддерживающий RAID 0, RAID 1, RAID 5 и RAID 6);
- 410/100/1000 Ethernet-порта;
- 3 PCI Express x8 порта;
- 4 внешних и 1 внутренний порт USB 2.0.

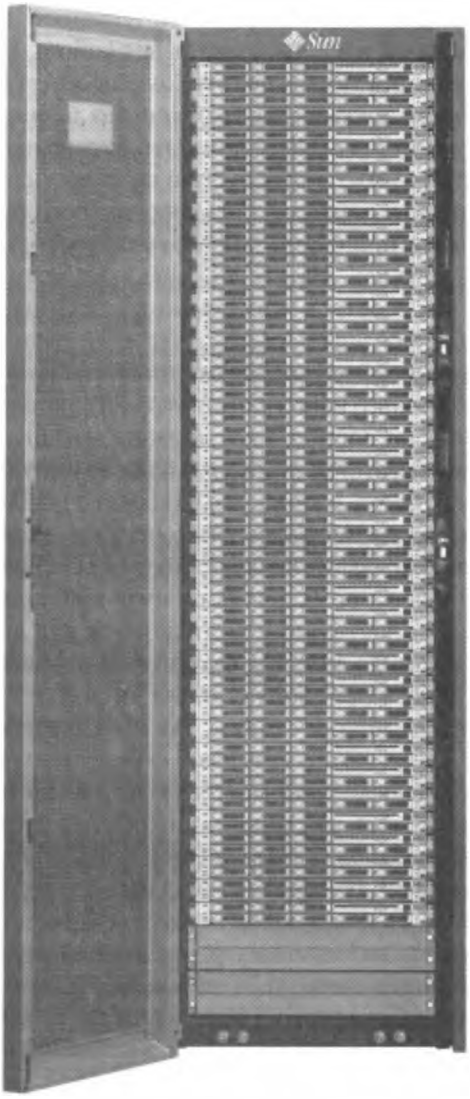


Рис. 6.8. Стандартная 19-дюймовая стойка, заполненная 42 серверами высотой 1U. В этой стойке находятся 42 1U сервера типа «коробки с пиццей». Источник: http://gchelpdesk.ualberta.ca/news/07mar06/cbhd_news_07mar06.php

На рис. 6.10 показана схема соединений и пропускные способности микросхем, расположеных на материнской плате. На рис. 6.3 и в табл. 6.7 дается описание чипсета ввода-вывода для Intel 5345, а в табл. 6.3 дается описание SAS-дисков Sun Fire x4150.

Чтобы пояснить суть совета по разработке систем ввода-вывода из раздела 6.8, давайте произведем вычисления производительности, чтобы узнать, где могут быть узкие места гипотетического приложения.

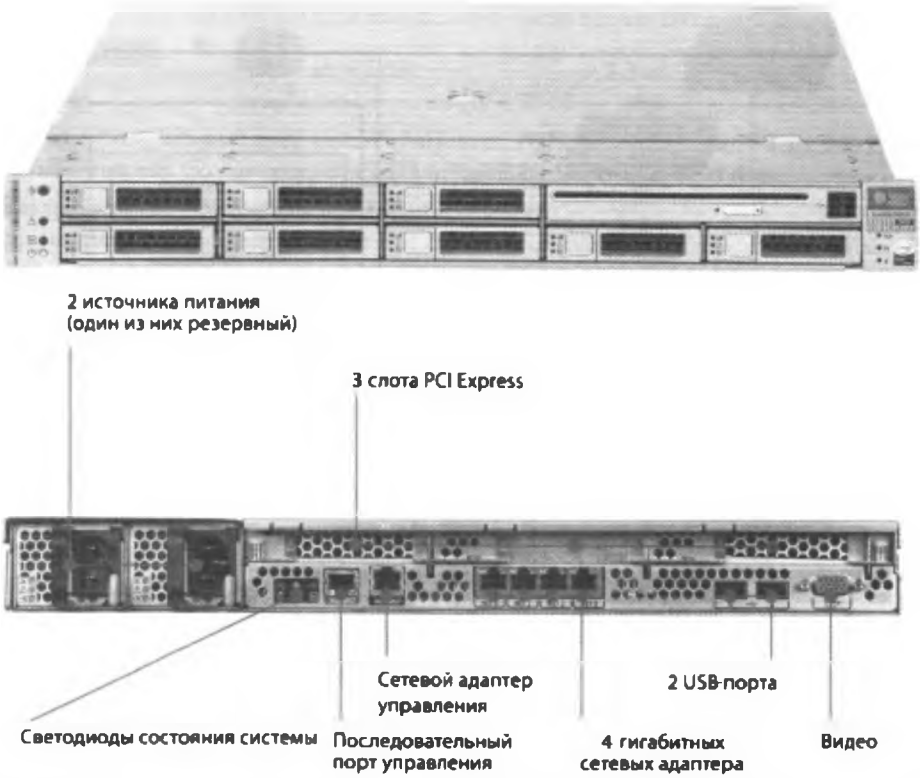


Рис. 6.9. Вид спереди и вид сзади сервера Sun Fire x4150 1U. Его размеры 1,75 дюйма в высоту и 19 дюймов в ширину. С лицевой части может быть произведена замена восьми 2,5-дюймовых дисков. В верхней правой части находятся DVD-привод и два USB-порта. На задней стороне имеются резервные источник питания и вентиляторы, предназначенные для обеспечения долговременной работы сервера, допускающей отказ одного из этих компонентов

Упражнение

Конструирование системы ввода-вывода

Примем ряд предположений относительно Sun Fire x4150:

- Пользовательская программа использует 200 000 инструкций на каждую операцию ввода вывода.
- Операционная система в среднем выполняет 100 000 инструкций на каждую операцию ввода-вывода.
- Рабочая нагрузка состоит из чтений по 64 Кбайт данных.
- Каждый процессор выполняет 1 млрд инструкций в секунду.

Определите максимальную устойчивую скорость ввода-вывода для полностью загруженного сервера Sun Fire x4150 для чтений в произвольном и в последовательном порядке. Предположим, что операции чтения всегда могут выполняться на бездействующем диске, если таковой имеется (то есть конфликты дисков игнорируются), и что RAID-контроллер не является узким местом системы.

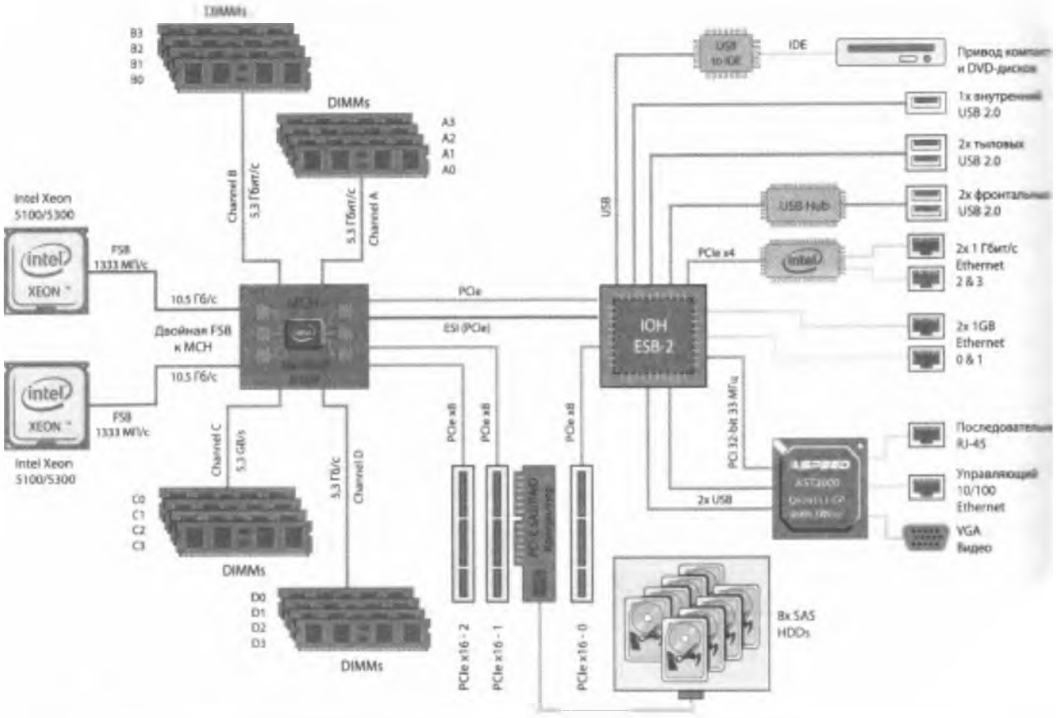


Рис. 6.10. Логические соединения и показатели пропускной способности компонентов Sun Fire x4150. Три слота PCIe допускают установку x16-карт, но обеспечивают лишь восьмиканальную пропускную способность при обмене с контроллером-концентратором памяти (MCH). Источник: рис. 5 «SUN FIRE™ X4150 AND X4450. SERVER ARCHITECTURE» (см. www.sun.com/servers/x64/x4150/)

Ответ

Давайте сначала определим скорость ввода-вывода отдельного процессора. На каждый ввод-вывод приходится по 200 000 пользовательских инструкций и по 100 000 инструкций операционной системы, следовательно, максимальная скорость ввода-вывода одного процессора =

$$4{,}0 \text{ мс} = \frac{0{,}5 \text{ оборота}}{15\,000 \text{ об/мин}} + \frac{0{,}5 \text{ Кбайт}}{100 \text{ Мбайт/с}} + 0{,}2 \text{ мс} = 4{,}0 + 2{,}0 + 0{,}005 + 0{,}2 = 6{,}2 \text{ мс}.$$

Поскольку в одном гнезде Intel 5345 работают четыре процессора, они могут выполнять 13 333 вводов-выводов в секунду. Восемь процессоров в двух гнездах могут выполнять 26 667 вводов-выводов в секунду.

Давайте определим количество вводов-выводов в секунду на один процессор для произвольных и последовательных операций чтения с 2,5-дюймового SAS-диска, показанного в табл. 6.3. Вместо использования среднего времени поиска, заявленного производителем, давайте предположим, что оно составляет только четверть заявленного времени, как это часто и бывает на самом деле (см. раздел 6.3). Время произвольного чтения с одного диска:

Время на один ввод-вывод с диска = поиск + время прокрутки + время передачи =

$$= \frac{2,9}{4} \text{мс} + 2,0 \text{мс} + \frac{64 \text{ Кбайт}}{112 \text{ Кбайт/с}} = 3,3 \text{ мс}.$$

Получается, что каждый диск может совершить 1000 мс/3,3 мс или 303 вводов-выводов в секунду, а восемь дисков совершат 2424 произвольных чтения в секунду.

Для последовательного чтения нужно просто размер передаваемых данных разделить на пропускную способность диска:

$$\frac{112 \text{ Кбайт/с}}{64 \text{ Кбайт}} = 1750 \text{ Вводов-выводов в секунду}.$$

Восемь дисков могут выполнить 14 000 последовательных чтений по 64 Кбайт.

Нужно посмотреть, не являются ли узкими местами пути от дисков к памяти и процессорам. Давайте начнем с соединения по шине PCI Express между RAID-картой и микросхемой северного моста. Каждый канал в PCIe передает данные со скоростью 250 Мбайт/с, поэтому восемь каналов могут передавать данные со скоростью 2 Гбайт/с.

$$\text{Максимальная скорость ввода-вывода } PCIex8 = \frac{\text{Пропускная способность шины}}{\text{Байт в вводе-выводе}} =$$

$$= \frac{5,3 \times 10^9}{64 \times 10^3} = 81\,540 \text{ Вводов-выводов в секунду}.$$

Даже для последовательной передачи данных с восьми дисков используется меньше половины возможностей канала PCIe x8. Поскольку данные получает блок управления памятью, они должны быть записаны в DRAM. Пропускная способность DDR2 667 МГц FBDIMM составляет 5336 Мбайт/с. Один DIMM-блок может выполнить

$$\frac{5356 \text{ Мбайт/с}}{64 \text{ Кбайт}} = 83\,375 \text{ Вводов-выводов в секунду}.$$

Память не станет узким местом даже при использовании одного DIMM-модуля, а у нас в полной конфигурации Sun Fire x4150 их 16.

Последним звеном в цепи является фронтальная шина — Front Side Bus, которая связывает концентратор северного моста с гнездом Intel 5345. Ее пиковая пропускная способность равна 10,6 Гбайт/с, но в разделе 7.10 предлагается не брать в расчет более половины пиковой возможности. Тогда имея 64 Кбайт при каждой передаче ввода-вывода

$$\text{Максимальная скорость ввода-вывода } FSB = \frac{\text{Пропускная способность шины}}{\text{Байт в вводе-выводе}} =$$

$$= \frac{5,3 \times 10^9}{64 \times 10^3} = 81\,540 \text{ Вводов-выводов в секунду}.$$

На каждое гнездо приходится одна фронтальная шина, поэтому пиковая возможность двойной FSB равна 150 000 вводов-выводов в секунду, поэтому и FSB не является узким местом.

Следовательно Sun Fire x4150 в полной конфигурации может поддерживать пиковую пропускную способность восьми дисков, составляющую 2424 произвольных чтения в секунду или 14 000 последовательных чтений в секунду.

Учтите, что для выполнения этого упражнения понадобилось большое количество упрощающих предположений. На практике многие из этих упрощений могут не соблюдаться для приложений с особенно интенсивным вводом-выводом. По этой причине зачастую единственным приемлемым способом оценки производительности ввода-вывода является работа с реальной рабочей нагрузкой или запуск подходящей контрольной задачи.

Как уже упоминалось в начале данного раздела, для этих новых дата-центров решаются не только вопросы стоимости и производительности, но и вопросы эффективного использования пространства и энергопотребления. В табл. 6.8 показана мощность, требуемая для Sun Fire x4150 в режимах ожидания и пиковой нагрузки, с распределением по каждому компоненту. Давайте посмотрим на альтернативные конфигурации Sun Fire x4150, позволяющие сберечь энергию.

Таблица 6.8. Потребляемая мощность Sun Fire x4150 полной конфигурации в режиме ожидания и пиковой нагрузки. Эти экспериментальные данные были получены при запуске контрольной задачи SPECJBB на 29 различных конфигурациях с целью получения разной пиковой потребляемой мощности при запуске разных приложений (источник: www.sun.com/servers/x64/x4150/calc)

	Компоненты			Система			
Элемент	Ожидание	Пик	К-во	Ожидание		Пик	
Одно гнездо с Intel 2,66 ГГц E5345, Intel 5000 MCB/IOH чипсетом, Ethernet-контроллерами, блоками питания, вентиляторами, ...	154 Вт	215 Вт	1	154 Вт	37%	215 Вт	39%
Дополнительное гнездо с Intel 2,66 ГГц E5345	22 Вт	79 Вт	1	22 Вт	5%	79 Вт	14%
4 Гбайт DDR2-667 5300 FBDIMM	10 Вт	11 Вт	16	160 Вт	39%	176 Вт	32%
73 Гбайт SAS 15K дисковые накопители	8 Вт	8 Вт	8	64 Вт	15%	64 Вт	12%
PCIe x8 дисковый RAID-контроллер	15 Вт	15 Вт	1	15 Вт	4%	15 Вт	3%
Всего	—	—	—	415 Вт	100%	549 Вт	100%

Упражнение

Оценка энергопотребления системы ввода-вывода

Переконфигурируйте Sun Fire x4150 для сведения к минимуму потребляемой энергии, при условии, что единственной работой для 1U сервера является рабочая нагрузка из предыдущего упражнения.

Ответ

Для получения 2424 произвольных чтений 64 Кбайт в секунду из предыдущего упражнения нам понадобятся все восемь дисков и PCI RAID-контроллер. Предыдущие вычисления

сказали, что один DIMM-модуль может поддерживать свыше 80 000 вводов-выводов в секунду, поэтому можно сэкономить энергию на памяти. Минимальная конфигурация памяти для Sun Fire x4150 составляет два DIMM-модуля, поэтому мы можем сэкономить энергию (и средства) на четырнадцати 4 Гбайт DIMM-модулях. Процессоры одного гнезда могут поддерживать 13 333 вводов-выводов в секунду, поэтому мы также можем сократить и одно количество используемых гнезд для Intel E5345.

Используя числа из табл. 6.8, получаем, что теперь общая потребляемая мощность равна

$$\text{Мощность ожидания}_{\text{произвольные чтения}} = 154 + 2 \times 10 + 8 \times 8 + 15 = 253 \text{ Вт}$$

$$\text{Пиковая мощность}_{\text{произвольные чтения}} = 215 + 2 \times 11 + 8 \times 8 + 15 = 316 \text{ Вт}$$

следовательно, потребляемая мощность сократилась в 1,6–1,7 раза.

Исходная система может производить 14 000 последовательных чтений 64 Кбайт в секунду. Нам по-прежнему нужны все диски и контроллер дисков, и эта более высокая нагрузка может обслуживаться таким же количеством DIMM-модулей. Рабочая нагрузка превышает вычислительную мощность процессоров, имеющихся в одном гнезде Intel E5345, поэтому нам нужно установить процессоры и во второе гнездо.

$$\text{Мощность ожидания}_{\text{последовательные чтения}} = 154 + 22 + 2 \times 10 + 8 \times 8 + 15 = 275 \text{ Вт}$$

$$\text{Пиковая мощность}_{\text{последовательные чтения}} = 215 + 79 + 2 \times 11 + 8 \times 8 + 15 = 395 \text{ Вт}$$

следовательно, потребляемая мощность сократилась в 1,4–1,5 раза.

6.11. Заблуждения и недоразумения

Заблуждение. *Рассчитанное время наработки на отказ у дисков составляет 1 200 000 часов, или почти 140 лет, поэтому диски практически никогда не выходят из строя.*

Приемы маркетинга производителей дисков дезориентируют пользователей. Как вычисляется такой показатель MTTF? Сначала производители помещают тысячи дисков в комнату, гоняют их несколько месяцев и подсчитывают количество отказавших. Показатель MTTF рассчитывается как общее количество часов суммарной доступности дисков, деленное на количество отказавших дисков.

Проблема в том, что полученное число существенно превышает срок службы диска, который обычно предполагается равным пяти годам, или 43 800 часам. Чтобы этот большой показатель MTTF имел какой-нибудь смысл, производители дисков утверждают, что расчеты соответствуют тому, что пользователь покупает диск, а затем меняет его на новый каждые пять лет, что соответствует запланированному сроку службы диска. Смысл утверждения состоит в том, что если многие пользователи (и их дети, внуки и правнуки) именно так и будут поступать не только в этом, но и в следующем столетии, то в среднем они заменят диск 27 раз, пока не столкнутся с его отказом, что составит около140 лет.

Более полезным показателем будет процент дисков, отказавших в течение года, который называется частотой отказов за год (annual failure rate, AFR). Предположим, 1000 дисков с показателем MTTF, равным 1 200 000 часов, используются 24 часа в сутки. Если заменять отказавшие диски новыми с теми же характеристиками надежности, то количество отказавших дисков за год (8 760 часов) будет равно

$$\text{Отказавшие диски} = \frac{1000 \text{ дисков} \times 8760 \text{ часов на диск}}{1\ 200\,000 \text{ часов на отказ}} = 7{,}3.$$

Иначе говоря, показатель AFR равен 0,73%. Чтобы предоставить пользователям более реальную картину того, что им следует ожидать от продукции, производители начали приводить показатели AFR, а также показатели MTTB.

Заблуждение. *На практике показатели отказов дисков соответствуют заявленным.*

В двух недавних исследованиях была произведена оценка большого количества дисков, чтобы проверить, как практические результаты соотносятся с заявленными параметрами. В одном исследовании использовалось почти 100 000 ATA и SCSI дисков с заявленными показателями MTTF от 1 000 000 до 1 500 000 часов, или с AFR от 0,6% до 0,8%. Оказалось, что чаще всего показатели AFR были равны от 2% до 4%, зачастую в 3–5 раз выше заявленных (Schroeder and Gibson, 2007).

Во втором исследовании, где использовалось более 100 000 ATA дисков с заявленными показателями AFR около 1,5%, оказалось, что показатель отказов, равный 1,7% для дисковых накопителей в первый год эксплуатации, поднялся до 8,6% в их третий год эксплуатации, или стал в 5–6 раз выше заявленного (Pinheiro, Weber and Barroso, 2007).

Заблуждение. *По каналу передачи данных, имеющему пропускную способность Гбайт/с, можно передать 1 Гбайт данных за одну секунду.*

Во-первых, использование на 100% любого ресурса компьютера в принципе невозможно. Что касается шины, в лучшем случае это удастся сделать на 70–80% от пиковой пропускной способности. Среди причин такого расхождения можно назвать время на отправку адреса, время на подтверждение сигналов и время ожидания освобождения шины.

Во-вторых, определения гигабайта устройства хранения и гигабайта в секунду при оценке пропускной способности отличаются друг от друга. Как уже ранее упоминалось, оценка полосы пропускания ввода-вывода дается по основанию 10 (то есть 1 Гбайт/с = 10^9 байт/с), а 1 Гбайт данных указывается по основанию 2 (то есть 1 Гбайт = 2^{30} байт). Насколько существенно данное различие? Если есть возможность использования шины на все 100% для передачи данных, то время передачи 1 Гбайт данных по каналу с пропускной способностью 1 Гбайт/с фактически будет равно

$$\frac{2^{30}}{10^9} = \frac{1\ 073\ 741\ 824}{1\ 000\ 000\ 000} = 1{,}073741824 \approx 1{,}07 \text{ секунды}.$$

Заблуждение. *Попытка предоставления определенных свойств только внутри сети, а не на конечных точках.*

Проблема заключается в предоставлении на низком уровне тех свойств, которые могут быть доведены до совершенства лишь на более высоком уровне, что лишь частично удовлетворяет потребностям передачи данных. Зальцер, Рид и Кларк в 1984 году изложили свой *аргумент в пользу конечных точек* следующим образом (Saltzer, Reed and Clark, 1984):

Рассматриваемое действие может быть полноценно и правильно определено только при осведомленности и помощи приложения, установленного в конечных точках коммуникационной системы. Поэтому предоставлять самой коммуникационной системе в виде ее свойства это подвергаемое сомнению действие нельзя.

Примером их заблуждения была сеть в MIT, использовавшая несколько шлюзов, каждый из которых добавлял контрольную сумму при передаче данных от одного шлюза к другому. Программисты приложений предполагали, что контрольная сумма гарантирует точность, ошибочно считая, что сообщение было защищено при хранении в памяти каждого шлюза. Но временами один из шлюзов отказывал, меняя местами одну пару на миллион переданных байтов. Спустя некоторое время исходный код одной операционной системы был несколько раз передан через шлюз, в результате чего он был поврежден. Одним из решений было исправление поврежденных исходных файлов путем сравнения с распечатками и правки кода вручную! Если бы контрольные суммы вычислялись и проверялись приложением, запущенным на конечных системах, безопасность была бы обеспечена.

Промежуточные проверки тоже полезны, но при условии, что доступна проверка и на конечных точках. Проверка на конечных точках может показать, что *что-то* было повреждено между двумя узлами, но она не указывает, где именно возникла проблема. Промежуточные проверки могут определить, *какой из компонентов* поврежден. Для ремонта нужны обе проверки.

Заблуждение. *Ожидание, что передача функций от центрального процессора процессору ввода-вывода улучшит производительность и без проведения тщательного анализа.*

Есть немало примеров, свидетельствующих о попадании людей в эту ловушку, хотя при правильном использовании процессоры ввода-вывода действительно могут поднять производительность. Довольно часто это заблуждение проявляется в использовании «умных» интерфейсов ввода-вывода, у которых из-за более высоких издержек при выдаче запроса на ввод-вывод могут оказаться худшие показатели латентности, чем у ввода-вывода, управляемого обычным процессором (хотя, если процессор полностью освобожден, пропускная способность системы может все-таки возрасти). Зачастую производительность падает, когда процессор ввода-вывода обладает намного меньшей производительностью, чем основной процессор. Вследствие этого небольшое количество времени основного процессора заменяется более существенным временем процессора ввода-вывода. С обоими феноменами разработчикам рабочих станций приходилось сталкиваться много раз.

Майер и Сазерленд в 1968 году написали классическую статью о компромиссах сложности и производительности в контроллерах ввода-вывода. Позаимствовав религиозную концепцию о «круговороте реинкарнаций», они в конечном итоге заметили, что попали в цикл постоянного роста мощности процессора ввода-вывода, который закончился тем, что для него понадобился свой собственный простой сопроцессор (Myer and Sutherland, 1968).

Мы подходили к решению задачи, начиная с простой схемы с последующим добавлением команд и свойств, которые, как нам казалось, должны были повысить мощность машины. Постепенно [дисплейный] процессор становился все сложнее <...>.

В конечном итоге дисплейный процессор стал напоминать полноценный компьютер, обладающий рядом специальных графических возможностей. И тогда произошло нечто необычное. Мы почувствовали необходимость добавить к процессору второй, вспомогательный процессор, который уже сам по себе стал все более усложняться. Именно тогда мы пришли к неутешительному выводу. Разработка дисплейного процессора может превратиться в бесконечный циклический процесс. Фактически мы поняли, что процесс нас настолько расстроил, что мы стали называть его «круговоротом реинкарнаций».

Недоразумение. *Использование накопителей на магнитной ленте для резервного копирования дисков.*

Это не только недоразумение, но и заблуждение. Накопители на магнитных лентах были частью компьютерных систем наряду с дисками, поскольку использовали ту же технологию, что и диски, и поэтому со временем подвергались тем же самым улучшениям плотности записи. Исторически сложившаяся разница между дисками и лентами в соотношении «стоимость—производительность» основана на том, что находящийся в герметично закрытом корпусе вращающийся диск имеет меньшее время доступа по сравнению с лентой с последовательным доступом; но съемные катушки с магнитной лентой дают возможность использования на одном устройстве чтения большого количества лент, которые могут иметь очень большую длину и, соответственно, большую емкость. Следовательно, когда в прошлом на одной магнитной ленте могло храниться содержимое множества дисков, и так как гигабайт был в 10–100 раз дешевле, чем на дисках, ленты были вполне приемлемым носителем для резервных копий.

Была установка по отслеживанию развития дисков, чтобы нововведения в дисках помогли прогрессу ленточных накопителей. Эта установка обусловливалась тем, что у лент был весьма небольшой рынок и не было возможности проводить отдельные большие научные исследования и разработки. Одна из причин узости рынка состоит в том, что владельцы настольных компьютеров не делают резервных копий на лентах. Получается, что настольные компьютеры являются самым большим рынком для дисков и слишком небольшим рынком для накопителей на магнитных лентах.

Увы, более широкий рынок привел к тому, что диски совершенствовались намного быстрее, чем накопители на магнитных лентах. Начиная с 2000–2002 годов наиболее популярные диски были больше по объему самого популярного накопителя на магнитной ленте. В тот же период времени стоимость гигабайта ATA дисков упала ниже его стоимости в ленточных накопителях. Сторонники ленточных накопителей утверждали, что у этих накопителей есть такие требования по совместимости, которые не предъявляются к дискам; ленточные накопители могут читать или вести запись на текущее или предыдущее поколение магнитных лент, и должны читать данные с последних четырех поколений. Поскольку диски являются закрытыми системами, головкам требуется читать данные только с тех пластин, которые находятся внутри этих дисков, и это преимущество объясняет, почему диски совершенствуются намного быстрее.

Сегодня некоторые организации вообще отказались от магнитных лент, используя сети и удаленные диски для копирования данных в других географических местах. Действительно, многие компании, предлагающие программное обеспечение в виде служб, используют недорогие компоненты, но дублируют данные на уровне приложений сразу в нескольких местах. Места подобраны таким образом, чтобы аварийные ситуации не могли вывести из строя сразу оба места, допуская мгновенное восстановление. (Большое время восстановления после аварийных ситуаций является еще одним крупным недостатком последовательной природы доступа к информации на магнитной ленте.) Такое решение, чтобы быть экономически оправданным, зависит от прогресса в повышении емкости дисков и пропускной способности сетей, но в развитие этих двух направлений вкладывается намного больше инвестиций, и, следовательно, у них намного более впечатляющие результаты, чем у накопителей на магнитной ленте.

Заблуждение. *Операционная система является наилучшим местом диспетчеризации обращений к диску.*

Как уже упоминалось в разделе 6.3, такие высокоуровневые интерфейсы, как ATA и SCSI, предлагают основной операционной системе адреса логических блоков. Располагая этой высокоуровневой абстракцией, лучшее, что может сделать операционная система, пытаясь содействовать производительности, — это отсортировать адреса логических блоков по возрастающей. Но, поскольку диск знает о настоящем отображении логических адресов на физическую геометрию секторов, дорожек и поверхностей, он может сократить латентность подкрутки и поиска за счет перепланировки обращений.

Предположим, например, что рабочая нагрузка состоит из четырех чтений (Anderson, 2003):

Операция	Начальный Логический адрес блока (LBA)	Длина
Чтение	724	8
Чтение	100	16
Чтение	9987	1
Чтение	26	128

Основная операционная система может изменить порядок четырех чтений, выстроив их по порядку следования логических блоков:

Операция	Начальный LBA	Длина
Чтение	26	128
Чтение	100	16
Чтение	724	8
Чтение	9987	1

В зависимости от относительного размещения данных на диске перестановка, как показано на рис. 6.11, может только ухудшить ситуацию. Чтение, порядок которого определяется самим дисковым устройством, может завершиться за три четверти оборота диска, но при определении этого порядка операционной системой чтение займет три оборота.

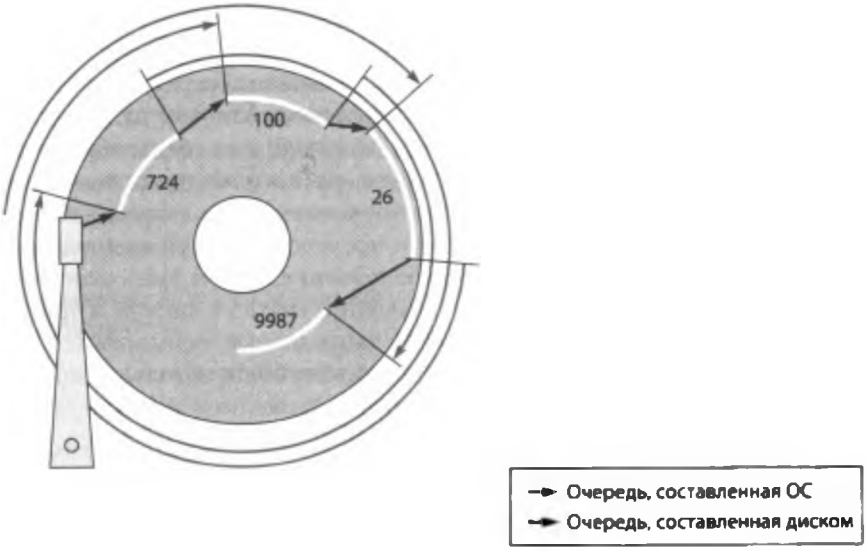


Рис. 6.11. Пример, показывающий план обращений, составленный операционной системой, в сравнении с планом, составленным дисковым устройством. Реализация первого плана занимает три оборота, а второго — только три четверти оборота (Anderson, 2003)

Заблуждение. *Использование пиковой скорости передачи, присущей какой-нибудь части системы ввода-вывода для прогнозирования или сравнения производительности.*

Многие компоненты системы ввода-вывода, от устройств до контроллеров и шин, характеризуются своей пиковой пропускной способностью. На практике оценки этой пиковой пропускной способности зачастую базируются на предположениях относительно системы, весьма далеких от реальности, или же они являются просто недосягаемыми из-за присущих системе ограничений. Например, при оценке производительности шины пиковая скорость передачи иногда определяется с использованием такой системы памяти, которую невозможно создать. А для сетевых систем игнорируются издержки программного обеспечения на установку связи.

32-разрядная шина PCI, работающая на частоте 33 МГц, имеет пиковую скорость передачи данных около 133 Мбайт/с. На практике даже для длинных передач для реальных систем памяти довольно трудно достичь скорости хотя бы в 80 Мбайт/с.

Закон Амдала также напоминает нам о том, что пропускная способность системы ввода-вывода будет ограничиваться компонентом на маршруте ввода-вывода, имеющим наименьшую производительность.

6.12. Заключительные комментарии

Системы ввода-вывода оцениваются по нескольким разным характеристикам: по безотказности, по разнообразию поддерживаемых устройств ввода-вывода, по максимальному количеству устройств ввода-вывода, по стоимости и производительности, измеряемой как по латентности, так и по пропускной способности. Эти оценки приводят к широкому разнообразию схем создания интерфейсов систем ввода-вывода. В самых дешевых и средних по стоимости системах доминирующим механизмом передачи является буферированный DMA. В дорогостоящих системах может быть важна как латентность, так и пропускная способность, а стоимость может иметь второстепенный характер. Дорогостоящие системы часто характеризуются множественностью маршрутов к устройствам ввода-вывода и ограниченной буферизацией. По мере разрастания системы более важной характеристикой обычно становится возможность доступа к данным на устройстве ввода-вывода в любой нужный момент (высокая степень готовности). В результате по мере расширения системы все более и более общепринятыми становятся избыточность и коррекция ошибок.

Требования к хранилищам данных и передаче данных по сети растут с невероятной скоростью, в частности из-за возрастания требований ко всей информации, которая должна быть в постоянном распоряжении. По одной из оценок, объем информации, созданной в 2002 году, был равен 5 эксабайтам, что эквивалентно 500 000 копиям текста в библиотеке Конгресса США, и этот общий объем информации в мире удваивается каждые три года (Lyman and Varian, 2003).

Будущее направление ввода-вывода включает расширение досягаемости проводных и беспроводных сетей, чтобы практически у каждого устройства потенциально был свой IP-адрес, расширение роли флэш-памяти в системах хранения информации.

Интерфейс аппаратного и программного обеспечения

Производительность систем ввода-вывода, независимо от того, в чем она измеряется, в показателях пропускной способности или латентности, зависит от всех элементов маршрута между устройством и памятью, включая операционную систему, генерирующую команды ввода-вывода. Пропускная способность каналов связи, памяти и устройства определяет максимальную скорость передачи данных от устройства или к устройству. Точно так же, латентность зависит от латентности устройства в совокупности с латентностью, налагаемой системой памяти или шинами. Высокие показатели пропускной способности и времени отклика также зависят от других запросов ввода-вывода, которые могут вызвать соревнование за обладание каким-нибудь ресурсом по маршруту. И наконец, узким местом является операционная система. В некоторых случаях операционная система очень долго доставляет запрос на ввод-вывод от пользовательской программы к устройству ввода-вывода, что приводит к высокой латентности. Бывают также случаи, когда операционная система существенно сужает пропускную способность ввода-вывода

в силу ограничений на количество параллельных операций ввода-вывода, которое она в состоянии поддерживать.

Имейте в виду: несмотря на то, что высокие показатели производительности могут способствовать увеличению объемов продаж системы ввода-вывода, пользователям необходимо получить от своих систем ввода-вывода большую емкость и безотказность.

6.13. Упражнения

Предоставлены Перри Александером (Perry Alexander) из Канзасского университета.

Упражнение 6.1

В табл. 6.1 приводится описание поведения, партнеров и скорости передачи данных устройств ввода-вывода. Но эта классификация не всегда дает полную картину потока данных внутри системы. Исследуйте классификацию следующих устройств.

а	Видеоигра
б	Портативный GPS-навигатор

6.1.1 [5] <6.1> Определите интерфейсы ввода-вывода и укажите классификацию по партнерам и поведению перечисленных в таблице устройств.

6.1.2 [5] <6.1> Вычислите скорость передачи данных интерфейсов, определенных в предыдущей задаче.

6.1.3 [5] <6.1> Определите, что будет лучше характеризовать производительность интерфейсов, определенных в предыдущей задаче, — скорость передачи данных или скорость, выраженная в количестве операций в секунду.

Упражнение 6.2

Среднее время безотказной работы (MTBF), среднее время на восстановление (MTTR) и наработка на отказ (MTTF) являются полезными показателями для оценки надежности и готовности ресурса хранения информации. Исследуйте эти понятия, отвечая на вопросы относительно устройств со следующими показателями.

	MTTF	MTTR
а	5 лет	1 неделя
б	10 лет	5 дней

6.2.1 [5] <6.1, 6.2> Вычислите MTBF для каждого устройства, представленного в таблице.

6.2.2 [5] <6.1, 6.2> Вычислите готовность для каждого устройства, представленного в таблице.

6.2.3 [5] <6.1, 6.2> Что происходит с готовностью при приближении показателя MTTR к нулю? Насколько реалистична подобная ситуация?

6.2.4 [5] <61, 6.2> Что происходит с готовностью, если показатель MTTR будет становиться очень высоким, то есть устройство будет трудно восстановить? Подразумевает ли это, что у устройства низкая степень готовности?

Упражнение 6.3

Для сравнения устройств хранения часто измеряется среднее и минимальное время чтения и записи данных. Используя технологии, рассмотренные в данной главе, вычислите значения, имеющие отношение к времени чтения и записи для дисков со следующими характеристиками.

	Среднее время поиска, мс	Об/мин	Скорость передачи данных, достигаемая диском, Мбайт/с	Скорость передачи данных, достигаемая контроллером, Мбайт/с
а	11	7200	34	480
б	9	7200	30	500

6.3.1 [10] <6.2, 6.3> Вычислите среднее время чтения или записи 1024-байтового сектора для каждого диска, перечисленного в таблице.

6.3.2 [10] <6.2, 6.3> Вычислите минимальное время чтения или записи 2048-байтового сектора для каждого диска, перечисленного в таблице.

6.3.3 [10] <6.2, 6.3> Определите для каждого перечисленного в таблице диска доминирующий фактор его производительности. Точнее говоря, в какой из показателей диска вы бы внесли изменения, располагая возможностями улучшения любого из них? Если доминирующие факторы отсутствуют, объясните почему.

Упражнение 6.4

В конечном счете, проектирование систем хранения данных требует не только рассмотрения параметров дисков, но и сценариев их использования. Разные ситуации требуют разных исходных параметров. Давайте попробуем дать оценку дисковым системам. Исследуйте разницу в том, как системы хранения информации должны быть оценены, путем ответа на вопросы относительно следующих приложений.

а	Интерактивная база спутников NASA
б	Система видеоигр

6.4.1 [5] <6.2, 6.3> Повысит ли производительность системы применительно к каждому приложению уменьшение размера сектора в процессе чтения или записи? Обоснуйте ответ.

6.4.2 [5] <6.2, 6.3> Повысит ли производительность системы применительно к каждому приложению увеличение скорости вращения диска? Обоснуйте ответ.

6.4.3 [5] <6.2, 6.3> Повысит ли применительно к каждому приложению увеличение скорости вращения диска производительность системы при условии уменьшения показателя MTTF? Обоснуйте ответ.

Упражнение 6.5

Флэш-память — один из главных реальных конкурентов традиционных дисковых накопителей. Исследуйте значение флэш-памяти, ответив на вопросы, касающиеся следующих приложений.

а	Интерактивная база спутников NASA
б	Система видеоигр

6.5.1 [5] <6.2, 6.3, 6.4> Что изменится в показателях времени чтения диска при переходе на твердотельные накопители, созданные из флэш-памяти, при условии, что скорость передачи данных останется прежней?

6.5.2 [10] <6.2, 6.3, 6.4> Получит ли каждое из приложений преимущества от твердотельного флэш-накопителя, учитывая, что стоимость является расчетным фактором?

6.5.3 [10] <6.2, 6.3, 6.4> Будет ли каждое из приложений считаться неподходящим для твердотельных флэш-накопителей, учитывая, что стоимость не является расчетным фактором?

Упражнение 6.6

Исследуйте особенности флэш-памяти, ответив на вопросы, касающиеся производительности устройств флэш-памяти со следующими характеристиками.

	Скорость передачи данных, Мбайт/с	Скорость передачи данных контроллером, Мбайт/с
а	34	480
б	30	500

6.6.1 [10] <6.2, 6.3, 6.4> Вычислите среднее время чтения или записи 1024-байтового сектора для каждого устройства флэш-памяти, перечисленного в таблице.

6.6.2 [10] <6.2, 6.3, 6.4> Вычислите минимальное время чтения или записи 512-байтового сектора для каждого устройства флэш-памяти, перечисленного в таблице.

6.6.3 [5] <6.2, 6.3, 6.4> В табл. 6.4 показано, что время обращения к флэш-памяти при чтении и записи увеличивается по мере того, как увеличивается объем флэш-памяти. Является ли это неожиданностью? Какие факторы являются причиной этого?

Упражнение 6.7

Ввод-вывод может осуществляться либо в синхронном, либо в асинхронном режиме. Исследуйте разницу между ними, отвечая на вопросы о производительности следующих периферийных устройств.

а	Мышь
б	Контроллер памяти

6.7.1 [5] <6.5> Какой тип шины лучше всего подойдет (синхронный или асинхронный) для обработки обмена данными между центральным процессором и перечисленными в таблице периферийными устройствами?

6.7.2 [5] <6.5> Какие проблемы создадут длинные, синхронные шины для связи центрального процессора с перечисленными в таблице периферийными устройствами?

6.7.3 [5] <6.5> Какие проблемы создадут асинхронные шины для связи центрального процессора с перечисленными в таблице периферийными устройствами?

Упражнение 6.8

К наиболее часто используемым сегодня типам шин относятся FireWire (IEEE 1394), USB, PCI и SATA. Хотя все четыре шины являются асинхронными, они реализованы по-разному, поэтому у них разные характеристики. Исследуйте структуры разных шин, отвечая на вопросы о шинах и о следующих периферийных устройствах.

а	Мышь
б	Контроллер памяти

6.8.1 [5] <6.5> Подберите шину (FireWire, USB, PCI или SATA), подходящую для перечисленных в таблице периферийных устройств. (Основные характеристики каждой шины приведены в табл. 6.6.)

6.8.2 [20] <6.5> Воспользуйтесь интернет-ресурсами или библиотечными материалами и составьте краткую сводку структуры обмена данными для шины каждого типа. Определите, что делает контроллер шины и где физически осуществляется управление.

6.8.3 [15] <6.5> Опишите ограничения шины каждого типа. Объясните, почему эти ограничения стоит учитывать при использовании шины.

Упражнение 6.9

Связь с устройствами ввода-вывода достигается с использованием комбинаций из опроса, обработки прерываний, отображения памяти и специальных инструкций ввода-вывода. Ответьте на вопросы о связи подсистем ввода-вывода со следующими приложениями с использованием комбинаций этих технологий.

а	Контроллер видеоигры
б	Компьютерный монитор

6.9.1 [5] <6.6> Опишите последовательный опрос устройства. Подойдет ли каждое из перечисленных в таблице приложений для связи с использованием технологии опроса? Обоснуйте.

6.9.2 [5] <6.6> Опишите обмен данными, управляемый с помощью прерываний. Для каждого из перечисленных в таблице приложений, если к нему не подходит опрос, предложите другую технологию.

6.9.3 [10] <6.6> Для приложений, перечисленных в таблице, опишите конструкцию обмена данными, использующую отображение на память. Определите зарезервированные места памяти и опишите их содержимое.

6.9.4 [10] <6.6> Дайте краткое описание конструкции, реализующей обмен данными, управляемый командами. Опишите команды и их взаимодействие с устройствами, перечисленными в таблице.

6.9.5 [5] <6.6> Есть ли смысл в определении подсистем ввода-вывода, которые используют комбинацию отображения в памяти и обмен данными, управляемый командами? Обоснуйте ответ.

Упражнение 6.10

В разделе 6.6 было дано определение процесса обработки прерываний, состоящего из восьми шагов. Информация о причине прерывания и о состоянии системы обработки прерываний предоставлялась в регистрах причины (Cause) и состояния (Status). Исследуйте обработку прерываний, ответив на вопросы о следующей комбинации прерываний.

а	Выключение питания	Перегрев	Контроллер данных Ethernet
б	Перегрев	Перезагрузка	Контроллер мыши

6.10.1 [5] <6.6> При обнаружении прерывания происходит сохранение значения регистра состояния и все прерывания, кроме тех, что имеют более высокий приоритет, запрещаются. Почему запрещаются прерывания с более низким приоритетом? Почему значение регистра состояния сохраняется до запрещения прерываний?

6.10.2 [10] <6.6> Расставьте по уровням приоритетности прерывания от устройств, перечисленных в каждой строке таблицы.

6.10.3 [10] <6.6> Кратко опишите, как будет обработано прерывание от каждого устройства, перечисленного в таблице.

6.10.4 [5] <6.6> Что произойдет, если бит разрешения прерывания в регистре причины на момент обработки этого прерывания не установлен? Какое значение может принять бит маски прерываний, чтобы получился тот же результат?

6.10.5 [5] <6.6> Большинство систем обработки прерываний реализуется в операционной системе. Какая аппаратная поддержка может быть добавлена для

повышения эффективности обработки прерываний? Сравните свое решение с потенциальной аппаратной поддержкой для вызовов функций.

6.10.6 [5] <6.6> В некоторых реализациях обработки прерываний прерывание становится причиной немедленного перехода на вектор прерывания. Вместо регистра причины, где каждым прерыванием устанавливается бит, у каждого прерывания есть свой собственный вектор прерывания. Может ли такая же система приоритетов прерываний быть реализована с использованием этого подхода?

Упражнение 6.11

Прямой доступ к памяти (DMA) позволяет устройствам обращаться к памяти напрямую, а не работать с ней через центральный процессор. Это может существенно повысить скорость работы периферийных устройств, но усложняет реализацию систем памяти. Исследуйте последствия применения DMA, ответив на вопросы о следующих периферийных устройствах.

а	Графическая карта
б	Звуковая карта

6.11.1 [5] <6.6> Отказывается ли центральный процессор от управления памятью, когда DMA проявляет активность? Например, может ли периферийное устройство напрямую вести обмен данными с памятью в обход центрального процессора?

6.11.2 [10] <6.6> Какое из периферийных устройств, перечисленных в таблице, будет иметь преимущества от использования DMA? Каким критерием определяется приемлемость DMA?

6.11.3 [10] <6.6> Какое из периферийных устройств, перечисленных в таблице, может стать причиной нарушения целостности содержимого кэш-памяти? Каким критерием определяется необходимость постановки вопроса о нарушении целостности?

6.11.4 [5] <6.6> Опишите проблемы, которые могут появиться при объединении DMA и виртуальной памяти. Какое из перечисленных в таблице периферийных устройств может вызвать такие проблемы? Как можно избежать их появления?

Упражнение 6.12

Оценочные показатели производительности устройств ввода-вывода могут очень сильно варьироваться от приложения к приложению. В одних ситуациях доминирующее влияние на производительность оказывает количество обрабатываемых транзакций, а в других ситуациях такое доминирующее влияние оказывает пропускная способность. Исследуйте оценку производительности ввода-вывода, отвечая на вопросы, касающиеся следующих приложений.

а	Просмотр интернет-ресурсов
б	Редактирование звука

6.12.1 [10] <6.7> Будет ли для каждого из перечисленных в таблице приложений доминирующим показателем общей производительности системы производительность ввода-вывода?

6.12.2 [10] <6.7> Будет ли простой показатель пропускной способности наилучшей оценкой производительности ввода-вывода для каждого из перечисленных в таблице приложений?

6.12.3 [5] <6.7> Будет ли количество обработанных транзакций наилучшей оценкой производительности ввода-вывода для каждого из перечисленных в таблице приложений?

6.12.4 [5] <6.7> Существует ли взаимосвязь между оценочными показателями производительности из предыдущих двух задач и выбором между использованием опросов или прерываний при организации управления обменом данными? А как насчет выбора ввода-вывода с отображением на память или с управлением с помощью команд?

Упражнение 6.13

Тесты играют важную роль в оценке и подборе периферийных устройств. Хорошие тесты должны создавать ситуации, имитирующие реальные условия работы. Изучите контрольные задачи и подбор устройств, ответив на вопросы, касающиеся следующих приложений.

а	Просмотр интернет-ресурсов
б	Редактирование звука

6.13.1 [5] <6.7> Для каждого из перечисленных в таблице приложений определите характеристики, которые необходимо протестировать при оценке подсистем ввода-вывода.

6.13.2 [15] <6.7> Используя интернет-ресурсы или библиотечные материалы определите набор стандартных контрольных задач для приложений, перечисленных в таблице. Чем обусловливается помощь стандартных контрольных задач?

6.13.3 [5] <6.7> Имеет ли смысл оценивать подсистему ввода-вывода отдельно от более крупной системы, частью которой она является? А как насчет оценки центрального процессора?

Упражнение 6.14

RAID-массив принадлежит к наиболее популярным подходам развития параллелизма и избыточности в системах хранения информации. Само название — Redundant Arrays of Inexpensive Disks, то есть избыточные массивы недорогих дисков, подразумевает сразу несколько понятий, относящихся к RAID-массивам, которые будут исследованы следующих задачах.

а	Предоставление интерактивных служб баз данных
б	Редактирование звука

6.14.1 [10] <6.9> Массив RAID 0 использует чередование для инициирования параллельного доступа сразу к нескольким дискам. Почему чередование увеличивает производительность работы с дисками? Будет ли чередование наилучшим образом соответствовать каждому действию, перечисленному в таблице?

6.14.2 [5] <6.9> Массив RAID 1 зеркально отображает данные на нескольких дисках. Учитывая то, что у недорогих дисков более низкий показатель MTBF, чем у дорогих, как может избыточное использование недорогих дисков отражаться на системе с более низким показателем MTBF? Для обоснования ответа воспользуйтесь математическим определением MTBF. Будет ли применение RAID 1 наилучшим образом соответствовать каждому действию, перечисленному в таблице?

6.14.3 [5] <6.9> Как и RAID 1, RAID 3 предоставляет более высокий показатель готовности данных. Объясните, согласуются друг с другом массивы RAID 1 и RAID 3. Получит ли каждое из приложений, перечисленных в таблице, преимущества от использования RAID 3 по сравнению с использованием RAID 1?

Упражнение 6.15

Массивы RAID 3, RAID 4 и RAID 5 для защиты блоков данных используют систему четности. Точнее говоря, блок четности связан с коллекцией блоков данных. Каждая строка в следующей таблице содержит значения блоков данных и четности, показанных на рис. 6.6.

	Новое значение D0	D0	D1	D2	D3	P
а	FEFE	00FF	A387	F345	FF00	4582
б	AB9C	F457	0098	00FF	2FFF	A387

6.15.1 [10] <6.9> Вычислите новое значение блока четности P' для массива RAID 3.

6.15.2 [10] <6.9> Вычислите новое значение блока четности P' для массива RAID 4.

6.15.3 [5] <6.9> Можно ли сказать, что RAID 3 или RAID 4 работает эффективнее? Есть ли причины, по которым RAID 3 будет предпочтительнее RAID 4?

6.15.4 [5] <6.9> RAID 4 и RAID 5 используют примерно один и тот же механизм для вычисления и сохранения четности для блоков данных. Чем RAID 5 отличается от RAID 4 и для каких приложений RAID 5 будет работать более эффективно?

6.15.5 [5] <6.9> Скорость работы RAID 4 и RAID 5 растет по сравнению RAID 3 по мере роста размеров защищенных блоков. Почему так происходит? Может ли сложиться ситуация, при которой RAID 4 и RAID 5 не будут работать более эффективно, чем RAID 3?

Упражнение 6.16

Появление веб-серверов для электронной торговли, интернет-магазинов и обмена информацией сделало дисковые серверы весьма ответственным устройством. Готовность и скорость являются широко известными показателями работы дисковых

серверов, но постоянно повышается и важность энергопотребления. Ответьте на вопросы, касающиеся конфигурации и оценки дисковых серверов, имеющих следующие параметры.

	Количество инструкций программы, приходящееся на одну операцию ввода-вывода	Количество инструкций операционной системы, приходящееся на одну операцию ввода-вывода	Рабочая нагрузка, Кбайт чтения	Скорость процессора, млрд инструкций/с
а	250 000	50 000	128	4
б	100 000	50 000	64	4

6.16.1 [10] <6.8, 6.10> Определите максимальную устойчивую скорость ввода-вывода для произвольного чтения и записи. Проигнорируйте конфликты дисков и представьте, что RAID-контроллер не является узким местом. Действуйте так, как было описано в разделе 6.10, делая такие же предположения там, где это необходимо.

6.16.2 [10] <6.8, 6.10> Предположим, мы занимаемся конфигурированием сервера Sun Fire x4150, как описано в разделе 6.10. Определите, будет ли узким местом ввода-вывода конфигурация из восьми дисков. Определите то же самое для конфигураций из 16, 4 и 2 дисков.

6.16.3 [10] <6.8, 6.10> Определите, представляют ли из себя узкое место ввода-вывода шина PCI, DIMM-модули или фронтальная шина. Используйте такие же параметры и предположения, что и в разделе 6.10.

6.16.4 [5] <6.8, 6.10> Объясните, почему для оценки текущей производительности в реальных системах стараются применять контрольные задачи или реальные приложения.

Упражнение 6.17

Определение производительности отдельного сервера с относительно полными данными не представляется трудной задачей. Но при сравнении серверов от разных производителей, предоставляющих разные данные, сделать выбор бывает нелегко. Исследуйте процесс подбора и оценки серверов, отвечая на вопросы, касающиеся следующего приложения.

Веб-сервер

6.17.1 [15] <6.8, 6.10> Определите рабочие характеристики для действующей системы применительно к показанному выше приложению. Выберите характеристики, которые будут поддерживать оценку, подобную той, которая давалась для упражнения 6.16.

6.17.2 [15] <6.8, 6.10> Подберите сервер, доступный на рынке, который, по вашему мнению, подошел бы для запуска показанного выше приложения. Перед оценкой сервера назовите причины, по которым он был отобран.

6.17.3 [20] <6.8, 6.10> Используя оценочные показатели, подобные тем, что использовались в главе 6 и в упражнении 6.16, оцените сервер, выбранный вами

в упражнении 6.17.2 в сравнении с сервером Sun Fire x4150, который оценивался в упражнении 6.16. Какой бы из них вы выбрали? Удивитесь ли вы результатам своего анализа? Изменился ли бы ваш выбор с учетом полученных результатов?

6.17.4 [15] <6.8, 6.10> Определите набор стандартных контрольных задач, который пригодился бы для сравнения подобранного вами в упражнении 6.17.2 сервера с Sun Fire x4150.

Упражнение 6.18

Чтобы получить предварительную оценку поведения накопителей, нужно весьма осторожно относиться к параметрам и статистике, предоставляемыми их поставщиками. В следующей таблице представлены данные для разных дисковых накопителей

	Количество накопителей	Количество часов на накопитель	Количество часов на отказ
а	1000	8 760	1 000 000
б	1000	10 512	1 500 000

6.18.1 [10] <6.12> Вычислите частоту отказов за год (AFR) для показанных в таблице дисков.

6.18.2 [10] <6.12> Предположим, что показатель частоты отказов за год варьируется в течение срока службы дисков, показанных в предыдущей таблице. Предположим, в частности, что AFR возрастает за первый месяц эксплуатации в три раза и удваивается каждый год начиная с пятого года. Сколько дисков будет подлежать замене после 7 лет эксплуатации? А сколько их будет заменено после 10 лет эксплуатации?

6.18.3 [10] <6.12> Предположим, что диски с более низкими показателями отказов стоят дороже. В частности, по более высокой цене можно приобрести диски, у которых частота отказов начнет удваиваться на восьмом, а не на пятом году эксплуатации. Насколько больше вы заплатите за диски, если захотите сохранить их в течение 7 лет? В течение 10 лет?

Упражнение 6.19

Предположим, что для дисков из таблицы в упражнении 6.18 поставщик предложил использовать конфигурацию RAID 0, которая должна повысить пропускную способность запоминающей системы на 70%, и конфигурацию RAID 1, которая снизит показатель AFR дисковых пар вдвое. Предположим, что стоимость каждого последующего решения в 1,6 раза больше, чем стоимость первоначального решения.

6.19.1 [5] <6.9, 6.12> Учитывая только лишь исходные параметры задачи, порекомендовали бы вы провести обновление либо до RAID 0, либо до RAID 1, при условии, что индивидуальные параметры диска останутся такими же, что и в предыдущей таблице?

6.19.2 [5] <6.9, 6.12> Учитывая то, что ваша компания работает с глобальной поисковой машиной, имеющей большой парк дисковых устройств, будет ли иметь экономический смысл обновление до RAID 0 или RAID 1, при условии, что ваша модель дохода основана на количестве поданных рекламных объявлений?

6.19.3 [5] <6.9, 6.12> Выполните упражнение 6.19.2 еще раз для крупного дискового парка, задействованного компанией, предоставляющей услуги резервного копирования по Интернету. Будет ли обновление либо до RAID 0, либо до RAID 1 иметь экономический смысл при условии, что ваша модель дохода основана на готовности ваших серверов?

Упражнение 6.20

Ежедневная оценка и обслуживание действующих компьютерных систем требуют применения многих понятий, рассмотренных в главе 6. Исследуйте сложности в оценке систем, изучив следующие вопросы.

6.20.1 [20] <6.10, 6.12> Сконфигурируйте Sun Fire x4150 для предоставления 10 Тбайт запоминающего устройства массиву из 1000 процессоров, занятых имитационным моделированием в области биоинформатики. Ваша конфигурация должна минимизировать потребляемую мощность при решении вопросов пропускной способности и готовности для дискового массива. При создании конфигурации не забудьте про свойства крупномасштабных задач имитационного моделирования.

6.20.2 [20] <6.10, 6.12> Выберите систему архивирования данных и хранения резервных копий применительно к дисковому массиву из упражнения 6.20.1. Дайте сравнительную характеристику диска, ленты и возможностей сохранения резервных копий в интернет-хранилище. Воспользуйтесь ресурсами Интернета и библиотечными материалами для определения потенциальных серверов. Оцените стоимость и приемлемость системы для приложения, используя параметры, рассмотренные в главе 6. Выберите параметры для сравнения, используя свойства приложения, а также указанные требования.

6.20.3 [15] <6.10, 6.12> Конкурирующие между собой поставщики систем, выбранных в упражнении 6.20.2, предлагают оценить их системы на месте. Укажите контрольные задачи, которые вы будете использовать для определения самой лучшей системы. Определите, сколько времени займет сбор достаточных данных для принятия решения.

Ответы на вопросы для самопроверки

Раздел 6.2: Справедливы утверждения 2 и 3.

Раздел 6.3: Справедливы утверждения 3 и 4.

Раздел 6.4: Все утверждения справедливы (при условии, что 40 Мбайт/с не сильно отличаются от 100 Мбайт/с).

Раздел 6.5: Достоверным является сведение 1.

Раздел 6.6: Истине соответствуют утверждения 1 и 2.

Раздел 6.7: Истинны утверждения 1 и 2. Утверждение 3 является ложным.

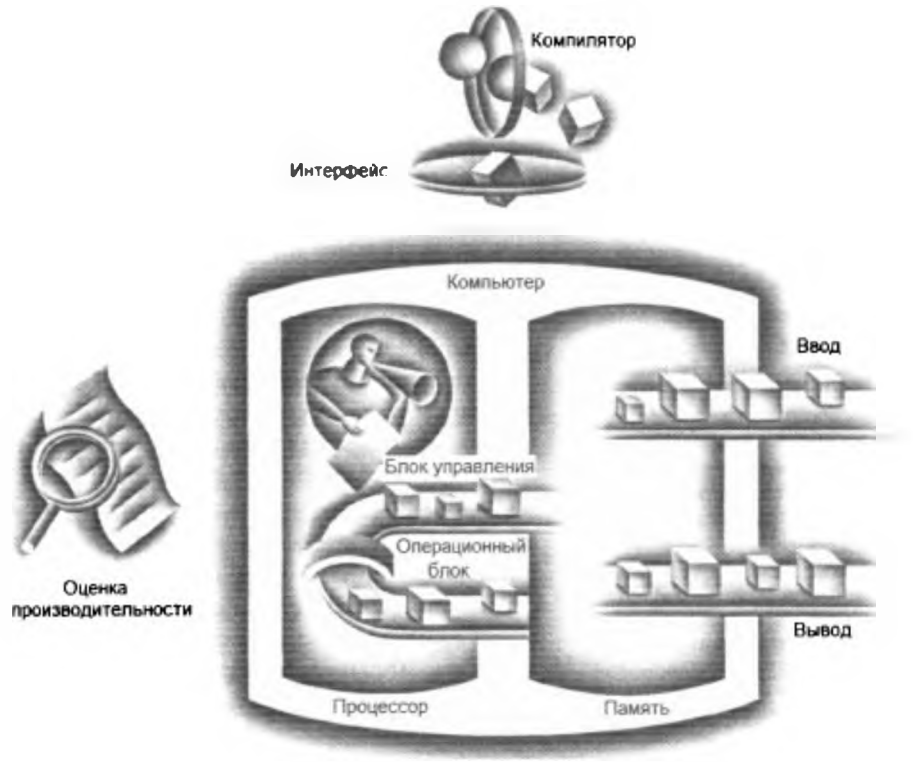
Раздел 6.9: Все утверждения являются справедливыми.

Глава 7

Многоядерность, мультипроцессорные системы и кластеры

Самая красивая рыба еще не поймана.
Ирландская поговорка

Мультипроцессор или кластерная организация



7.1. Введение

И ответила Тень:
«Где рождается день,
Лунных Гор где чуть зрима громада
Через ад, через рай,
Все вперед поезжай,
Если хочешь найти Эльдорадо!»[1]

Эдгар Аллан По,
«Эльдорадо», строфа 4, 1849 г.

Разработчики компьютеров долго искали Эльдорадо: создание мощных компьютеров путем простого соединения множества менее мощных. Это представление о компьютерном «золоте» привело к созданию мультипроцессорных систем. В идеале клиенты заказывают столько процессоров, сколько могут себе позволить, получая при этом соответствующий уровень производительности. Таким образом, программное обеспечение мультипроцессорных систем должно быть разработано с учетом его запуска на разном количестве процессоров. Как упоминалось в главе 1, как для центров обработки данных, так и для микропроцессоров важнейшей проблемой стала потребляемая мощность. Замена больших, неэффективных процессоров множеством небольших, эффективных может предоставить лучшую производительность на ватт или на джоуль потребляемой мощности, как в больших, так и в небольших системах, если программное обеспечение сможет эффективно их использовать. Таким образом, к повышению эффективности энергопотребления присоединяется масштабируемая производительность в случае использования мультипроцессоров.

Поскольку программное обеспечения мультипроцессорных систем может масштабироваться, некоторые конструкции поддерживают операции в случае выхода оборудования из строя; то есть, если в мультипроцессорной системе, состоящей из n процессоров, выйдет из строя один процессор, эта система продолжит предоставление услуг, имея в своем распоряжении $n - 1$ процессор. Следовательно, мультипроцессоры могут также повысить готовность (см. главу 6).

Мультипроцессор
Компьютерная система, имеющая минимум два процессора, в отличие от однопроцессорной системы, у которой имеется всего один процессор.

Параллелизм на уровне заданий или параллелизим на уровне процессов
Использование нескольких процессоров путем одновременного запуска независимых программ.

Программа, выполняемая в параллельном режиме
Отдельная программа, одновременно запущенная сразу на нескольких процессорах.

Высокая производительность может означать высокую пропускную способность для независимых друг от друга заданий, которая называется **параллелизмом на уровне заданий или параллелизмом на уровне процессов**. Эти параллельные задания являются независимыми приложениями, важным и весьма популярным способом использования параллельных компьютеров. Такой подход сильно отличается от запуска на мультипроцессорной системе одного задания. Термин **«программа, выполняемая**

[1] Перевод К. Бальмонта.

в параллельном **режиме**» будет использоваться для обозначения одной программы, которая одновременно запускается на нескольких процессорах.

Существуют такие научные задачи, для решения которых нужны намного более быстродействующие компьютеры, и этот класс задач используется для настройки многих новых параллельных компьютеров на протяжении последних десятилетий. Часть из них будет рассмотрена в данной главе. Некоторые из этих задач могут решаться довольно просто, с использованием **кластера**, составленного из микропроцессоров, находящихся во многих независимых серверах или в персональных компьютерах. Кроме того, кластеры могут обслуживать и столь же требовательные приложения, не относящиеся к научной сфере, например поисковые машины, веб-серверы, серверы электронной почты и базы данных.

Как уже говорилось в главе 1, мультипроцессоры привелкли наше внимание в первую очередь, поскольку задача повышения производительности может решаться только за счет увеличения количества процессоров на одном кристалле, а не за счет более высоких тактовых частот и улучшения показателя CPI. Они назывались **многоядерными микропроцессорами**, а не мультипроцессорными микропроцессорами, предположительно для того, чтобы избавиться от тавтологии в названии. Следовательно, процессоры в многоядерных кристаллах часто называют ядрами. Ожидается, что количество ядер будет удваиваться каждые два года. Стало быть, программисты, которых интересует производительность, должны становиться специалистами по параллельному программированию, так как последовательные программы означает медленные программы.

Самой серьезной задачей, с которой столкнулась промышленность, стало создание аппаратного и программного обеспечения, облегчающего написание работоспособных параллельно выполняемых программ, имеющих эффективные показатели производительности и энергопотребления по мере геометрического роста количества ядер на одном кристалле.

Этот неожиданный поворот в конструкции микропроцессоров многих застал врасплох, создав путаницу в терминологии. В табл. 7.1 предпринимается попытка дать определения таким понятиям, как «последовательный» и «параллельный», применительно к аппаратному и программному обеспечению. В столбцах этой таблицы представлено программное обеспечение, которое бывает либо последовательным (sequential), либо параллельным (concurrent). В строках таблицы представлено аппаратное обеспечение, которое также предназначено для последовательной (serial) или для параллельной (parrallel) работы. Например, разработчики компиляторов считают последовательными программы, осуществляющие следующие действия: этапы лексического анализа, синтаксический анализ, генерацию кода, оптимизацию и т. д. В отличие от этого, разработчики операционных систем обычно считают параллельными программами взаимодействующие процессы обработки событий ввода-вывода, поскольку они относятся к разным заданиям, запущенным на компьютере.

Кластер
Набор компьютеров, соединенных по локальной вычислительной сети (local area network, LAN), функционирующий как единая мультипроцессорная система.

Многоядерный микропроцессор
Микропроцессор, содержащий несколько процессоров («ядер») на одной интегральной микросхеме.

Таблица 7.1. Классификация аппаратного и программного обеспечения и «параллельность» с точки зрения разработчиков приложений и с точки зрения разработчиков оборудования

		Программное обеспечение	
		Последовательное (sequential)	**Параллельное (concurrent)**
Аппаратное обеспечение	**Последовательное (serial)**	Перемножение матриц, написанное на MatLab и запущенное на Intel Pentium 4	Операционная система Windows Vista, запущенная на Intel Pentium 4
	Параллельное (parallel)	Перемножение матриц, написанное на MatLab и запущенное на Intel Xeon e5345 (Clovertown)	Операционная система Windows Vista, запущенная на Intel Xeon e5345 (Clovertown)

Суть этих двух точек зрения, представленных в табл. 7.1, состоит в том, что параллельное программное обеспечение может запускаться на последовательном оборудовании, например операционные системы на одноядерном процессоре Intel Pentium 4, или на параллельном оборудовании, представленном более современным процессором Intel Xeon e5345 (Clovertown). То же самое справедливо и для последовательного программного обеспечения. Например, программисты, работающие на MatLab, пишут перемножение матриц, представляя эту работу в последовательном режиме, но это перемножение может быть запущено последовательно на оборудовании Pentium 4 или параллельно на оборудовании Xeon e5345. Нетрудно догадаться, что единственной сложностью параллельной революции является определение методов достижения высокой производительности выполнения изначально последовательного программного обеспечения на мультипроцессорных системах по мере роста количества процессоров. После того как мы во всем этом разобрались, для обозначения последовательного или параллельного программного обеспечения, запущенного на параллельном оборудовании, во всем остальном материале главы будет использоваться понятие *параллельно выполняемой программы*, или *параллельного программного обеспечения*.

В следующем разделе описываются причины сложностей создания эффективных параллельно выполняемых программ. В разделах 7.3 и 7.4 описываются две альтернативные характеристики фундаментального параллельного оборудования, основанные на зависимости или независимости всех процессоров от единого физического адреса. Две популярные версии этих альтернатив называются *мультипроцессорами с общей памятью* и *кластерами*. Затем в разделе 7.5 дается описание *многопоточности*, понятия, которое часто путают с мультипроцессорностью, отчасти потому, что многопоточность полагается на такую же параллельность в программах. В разделе 7.6 описывается классификационная схема, отображающая более ранние представления по сравнению с теми, которые приведены в табл. 7.1. Кроме этого в нем дается описание двух типов архитектур набора инструкций, которые поддерживают запуск последовательных приложений на параллельном оборудовании, а именно *SIMD* и *vector*. В разделе 7.7 описывается относительно

новый тип компьютера от сообщества разработчиков графического оборудования, который называется *блоком обработки графики* (graphics processing unit, *GPU*), или *графическим процессором*. Затем в разделе 7.9 рассматриваются сложности подбора контрольных задач для параллельного выполнения. За этим разделом следует описание новой, простой и все же довольно глубокой модели, помогающей в разработке приложений и архитектур. Эта модель используется в разделе 7.11 для оценки четырех последних многоядерных процессоров на двух основных прикладных задачах. В завершение рассматриваются заблуждения и недоразумения и даются краткие выводы по параллельным системам.

Перед тем как углубиться в тематику параллельных вычислений, не забудьте о наших сделанных в предыдущих главах вторжениях в эту область:

- глава 2, раздел 2.11: Параллелизм и инструкции: синхронизация;
- глава 3, раздел 3.6: Параллелизм и компьютерная арифметика: ассоциативность;
- глава 4, раздел 4.10: Параллелизм и расширенный параллелизм на уровне инструкций;
- глава 5, раздел 5.8: Параллелизм и иерархии памяти: целостность данных в кэш-памяти;
- глава 6, раздел 6.9: Параллелизм и ввод-вывод: избыточные массивы недорогих дисков.

Самопроверка

Определите истинность или ложность следующего утверждения: чтобы получить преимущество от использования мультипроцессора, приложение должно быть рассчитано на параллельное выполнение.

7.2. Сложности создания программ, выполняемых в параллельном режиме

Сложности параллелизму создает не оборудование, на самом деле для ускорения выполнения задач на мультипроцессорах были переписаны лишь немногие известные прикладные программы. Сложно создать такое программное обеспечение, которое использует несколько процессоров для ускоренного завершения одной задачи, и с ростом количества процессоров эта проблема только усугубляется.

В чем причина этой сложности? Почему разработка параллельно выполняемых программ дается намного сложнее, чем разработка последовательных программ?

Первая причина состоит в том, что вы *должны* получить более высокую производительность и эффективность от параллельного выполнения программ на мультипроцессорной системе, иначе получится, что вы будете просто использовать последовательную программу на одном процессоре, поскольку такое программи-

рование проще. На самом деле в таких технологиях конструирования однопроцессорных систем как суперскаляры и процессоры с измененной последовательностью выполнения инструкций, используется параллелизм на уровне инструкций (см. главу 4), и для этого не требуется вмешательство программиста. Подобные нововведения сокращают потребности в переписывании программ для мультипроцессоров, поскольку программистам здесь нечего делать, и даже их последовательные программы будут работать на новых компьютерах быстрее.

Почему так трудно написать параллельно выполняемые программы, которые работают быстрее, особенно при увеличении количества процессоров? В главе 1 была использована аналогия с восемью репортерами, пытающимися написать одну статью в надежде на то, что это удастся сделать в восемь раз быстрее. Чтобы добиться успеха, задача должна быть разбита на восемь частей одинакового размера, поскольку в противном случае кто-то из репортеров будет бездельничать в ожидании тех коллег, которым досталась более весомая часть работы. Еще одна загвоздка производительности заключается в том, что репортерам придется потратить слишком много времени на общение друг с другом вместо написания своей части статьи. Как для данной аналогии, так и для параллельного программирования трудности заключаются в диспетчеризации, балансе загруженности, времени на синхронизацию и издержках на обмен данными между участниками. Ситуация усложняется с ростом количества репортеров, пишущих статью в газету и с ростом количества процессоров для параллельного программирования.

При рассмотрении материала главы 1 было обнаружено еще одно препятствие, а именно закон Амдала. Он напоминает нам о том, что в программе, разрабатываемой с прицелом на рациональное использование множества ядер, необходимо распараллелить даже небольшие части.

Упражнение

Сложности ускорения

Предположим, что нужно достичь ускорения в 90 раз, имея 100 процессоров. Какой процент исходных вычислений может проводиться в последовательном режиме?

Ответ

Закон Амдала (глава 1) гласит:

$$\text{Время выполнения после улучшения} = \frac{\text{Время выполнения, на которое влияет улучшение}}{\text{Степень улучшения}} + \text{Время выполнения, на которое не влияет улучшение.}$$

Этот закон можно переформулировать в понятиях ускорения по отношению к исходному времени выполнения:

$$\text{Ускорение} = \frac{\text{Прежнее время выполнения}}{(\text{Прежнее время выполнения} - \text{Время выполнения, на которое влияет улучшение}) + \frac{\text{Время выполнения, на которое влияет улучшение}}{100}}.$$

Эта формула обычно переписывается на основе предположения, что прежнее время выполнения можно принять за некую единицу измерения времени, равную 1, время выполнения, на которое влияет улучшение, считать долей прежнего времени выполнения:

$$\text{Ускорение} = \frac{1}{(1 - \text{Доля времени, на которое влияет улучшение}) + \frac{\text{Доля времени, на которое влияет улучшение}}{100}}$$

Подставляем в показанную выше формулу требуемое ускорение в 90 раз:

$$90 = \frac{1}{(1 - \text{Доля времени, на которое влияет улучшение}) + \frac{\text{Доля времени, на которое влияет улучшение}}{100}}$$

Затем упрощаем формулу и вычисляем долю времени, на которое влияет улучшение:

$$90 \times (1 - 0{,}99 \times \text{Доля времени, на которое влияет улучшение}) = 1;$$

$$90 - (90 \times 0{,}99 \times \text{Доля времени, на которое влияет улучшение}) = 1;$$

$$90 - 1 = 90 \times 0{,}99 \times \text{Доля времени, на которое влияет улучшение};$$

$$\text{Доля времени, на которое влияет улучшение} = 89/89{,}1 = 0{,}999.$$

Получается, что для достижения ускорения в 90 раз при использовании 100 процессоров процент последовательно выполняемого кода должен составлять только 0,1%.

Кроме того, существуют приложения с последовательным параллелизмом.

Упражнение

Более сложная задача

Предположим, что нужно получить две суммы: сумму 10 скалярных переменных и матрицу суммы двух двумерных массивов размером 10 на 10. Какое ускорение будет получено при использовании 10 процессоров по сравнению с использованием 100? После этого вычислите ускорение при условии, что матрица увеличилась в размерах до 100 на 100 элементов.

Ответ

Если предположить, что производительность — это функция времени для суммирования, t, то мы получим 10 суммирований, которые не получают преимуществ от параллельных процессоров, и 100 суммирований, которые такие преимущества получают. Если время для одного процессора равно 110 t, то время выполнения для 10 процессоров равно

$$\text{Время выполнения после улучшения} = \frac{\text{Время выполнения, на которое влияет улучшение}}{\text{Степень улучшения}} + \text{Время выполнения, на которое не влияет улучшение.}$$

Получается, что ускорение при использовании 10 процессоров равно $100\,t/20\,t = 5{,}5$. Время выполнения при использовании 100 процессоров равно

$$\text{Время выполнения после улучшения} = \frac{100\,t}{10} + 10\,t = 20\,t.$$

Получается, что ускорение при использовании 100 процессоров равно $110\,t/11\,t = 10$.

Таким образом, при данных условиях задачи мы получаем 55% потенциального ускорения при использовании 10 процессоров и только 10% при использовании 100 процессоров. Посмотрим, что получится при увеличении размеров матрицы. Последовательной программе теперь понадобится $100\,t + 10000\,t = 10\,010\,t$. Время выполнения при использовании 10 процессоров будет равно

$$\text{Время выполнения после улучшения} = \frac{100\,t}{10} + 10\,t = 11\,t.$$

получается, что ускорение при 10 процессорах будет равно $10\,010\,t/1010\,t = 9{,}9$. Время выполнения при использовании 100 процессоров будет равно

$$\text{Время выполнения после улучшения} = \frac{10\,000\,t}{10} + 10\,t = 1010\,t.$$

Получается, что ускорение при 100 процессорах будет равно $10\,010\,t/110\,t = 91$. Таким образом, для этой более объемной задачи мы получаем около 99% потенциального ускорения при 10 процессорах и более чем 90% при 100 процессорах.

Из этих примеров следует, что получить хорошее ускорение на мультипроцессоре при фиксированном объеме задачи труднее, чем получить его при увеличении объема задачи. Это дает возможность представить еще два понятия, описывающих способы увеличения масштаба. **Строгое масштабирование** означает оценку ускорения при фиксированном объеме задачи. **Нестрогое масштабирование** означает, что объем задачи растет пропорционально увеличению количества процессоров. Предположим, что объем задачи, M, равен рабочему набору, хранящемуся в оперативной памяти, и у нас имеется P процессоров. Тогда объем памяти, приходящийся на каждый процессор для строгого масштабирования, будет примерно равен M/P, а для нестрогого масштабирования он будет примерно равен M.

В зависимости от приложения можно привести доводы в пользу любого подхода к масштабированию. Например, дебетно-кредитная контрольная задача TPC-C (глава 6) требует увеличения количества клиентских счетов для достижения большего количества транзакций в минуту. Возражение в том, что наивно полагать, что имеющаяся масса клиентов неожиданно начнет использовать банкоматы по сто раз на день только потому, что у банка появился более скоростной компьютер. Вместо этого, если вы собираетесь продемонстрировать систему, которая может выполнять в 100 раз больше транзакций в минуту, нужно провести эксперимент с увеличенным в 100 раз количеством клиентов.

Строгое масштабирование
Ускорение, достигнутое на мультипроцессорной системе БЕЗ увеличения объема задачи.

Нестрогое масштабирование
Ускорение, достигнутое на мультипроцессорной системе при увеличении объема задачи, пропорциональном увеличению количества процессоров.

В последнем упражнении показывается важность получения сбалансированной нагрузки.

Упражнение

Получение сбалансированной нагрузки
Чтобы получить 91-разовое ускорение в предыдущей более сложной задаче с использованием 100 процессоров, мы предполагали, что нагрузка была

идеально сбалансирована. То есть у каждого из 100 процессоров был 1% выполняемой работы. А теперь вместо этого покажем влияние на ускорение более высокой загруженности одного процессора по сравнению со всеми остальными. Проведите вычисления исходя из нагрузки 2 и 5 процентов.

Ответ

Если процессор имеет 2% параллельной нагрузки, то он должен произвести 2% × 1000 или 200 сложений, а остальные 99 процессоров поделят между собой оставшиеся 9800. Поскольку они работают одновременно, время выполнения можно вычислить как максимум.

$$\text{Время выполнения после улучшения} = \frac{10\,000\,t}{10} + 10\,t = 110\,t.$$

Ускорение упадет до $10\,010\,t/210\,t = 48$. Если у одного из процессоров будет 5% нагрузки, он должен будет выполнить 500 сложений.

$$\text{Время выполнения после улучшения} = \text{Max}\left(\frac{9500\,t}{99}, \frac{500\,t}{1}\right) + 10\,t = 510\,t.$$

Ускорение упадет еще больше, до $10\,010\,t/510\,t = 20$. Этот пример показывает важность сбалансированной нагрузки. Удвоенная по сравнению со всеми остальными нагрузка одного процессора срезает ускорение почти наполовину, а его пятикратная по отношению к другим процессорам нагрузка снижает ускорение почти в пять раз.

Самопроверка

Определите истинность или ложность следующего утверждения: строгое масштабирование не ограничивается законом Амдала.

7.3. Мультипроцессоры с общей памятью

Учитывая трудности переписывания старых программ для их эффективной работы на параллельном оборудовании, вполне уместно задать вопрос: что могут сделать разработчики компьютеров, чтобы упростить эту задачу? Один из ответов состоит в предоставлении единого физического адресного пространства в общее пользование всем процессорам, чтобы программы не утруждали себя проблемами места их запуска и учитывали только то, что они могут выполняться в параллельном режиме. При таком подходе все переменные программы могут быть в любое время доступны любому процессору. Альтернативный вариант заключается в создании отдельного адресного пространства для каждого процессора, что потребует явного обозначения общей памяти; этот вариант будет рассмотрен в следующем разделе. При использовании общего физического адресного пространства, что является вполне обычным явлением для многоядерных микросхем, оборудование, как правило, обеспечивает целостность кэш-памяти, чтобы предоставить единую картину общей памяти (см. раздел 5.8 главы 5).

Мультипроцессор с общей памятью (shared memory multiprocessor, **SMP**) относится к мультипроцессорам, предоставляющим программистам *единое физическое*

адресное пространство для всех процессоров, хотя, если выражаться точнее, его нужно было бы назвать мультипроцессором с общими *адресами*. Следует заметить, что такие системы могут все же запускать независимые друг от друга задания в их собственных виртуальных адресных пространствах, даже если все они используют общее физическое адресное пространство. Взаимодействие процессоров осуществляется через общие переменные в памяти, и все процессоры имеют возможность обращаться к любому месту памяти, загружая или записывая данные. Классическая организация SMP показана на рис. 7.1.

Рис. 7.1. Классическая организация мультипроцессора с общей памятью

Микропроцессоры с единым адресным пространством бывают двух типов. Первый из них характеризуется одинаковым временем обращения к оперативной памяти, независимо от того, какой из процессоров запрашивает это обращение и какое именно слово запрашивается. Такие машины называются мультипроцессорами с **однородным доступом к памяти** (uniform memory access, **UMA**). У второго типа часть адресов памяти имеет намного более высокое быстродействие, в зависимости от того, какой процессор какое слово запросил. Такие машины называются мультипроцессорами с **неоднородным доступом к памяти** (nonuniform memory access, **NUMA**). Как вы, наверное, догадались, программирование для NUMA-микропроцессоров дается труднее, чем программирование для UMA-процессоров, но NUMA-машины могут масштабироваться до весьма больших размеров, и NUMA-технологии могут обеспечивать меньшую латентность для близлежащей памяти.

Мультипроцессор с общей памятью (SMP)
Параллельный процессор с единым адресным пространством, допускающий передачу данных посредством загрузок и сохранений.

Однородный доступ к памяти (UMA)
Мультипроцессор, в котором доступ к оперативной памяти занимает примерно одинаковое время, независимо от того, какой процессор запросил доступ и какое слово он запросил.

Неоднородный доступ к памяти (NUMA)
Тип мультипроцессора с единым адресным пространством, в котором некоторые обращения к памяти осуществляются намного быстрее других в зависимости от того, какой процессор какое слово запросил.

По мере того как работающие в параллельном режиме процессоры будут использовать общие данные, им понадобится координация работы с общими данными, в противном случае один процессор может приступить к работе с данными еще до того, как другой завершит работу с ними. Такая координация называется **синхронизацией**. В то время как общий доступ поддерживается единым адресным простран-

ством, для синхронизации необходим отдельный механизм. В одном из подходов для общих переменных используется **блокировка**. Только один процессор в конкретный момент времени может воспользоваться блокировкой, а другие процессоры, заинтересованные в доступе к общим данным, должны ждать, пока первый процессор не разблокирует переменную. В разделе 2.11 главы 2 дается описание инструкций, которые используются для блокировки в MIPS.

Упражнение

Простая программа, выполняющаяся в параллельном режиме, предназначенная для общего адресного пространства

Предположим, что нужно получить сумму 100 000 чисел на компьютере, имеющем мультипроцессор с общей памятью с однородным доступом к памяти. Будем считать, что у нас имеется 100 процессоров.

Ответ

Сначала опять нужно будет разбить набор чисел на поднаборы одинакового размера. Распределять поднаборы по разным пространствам памяти не придется, поскольку на данной машине используется единое адресное пространство; мы просто дадим каждому процессору разные начальные адреса. Pn — это номер, идентифицирующий каждый процессор в диапазоне от 0 до 99. Все процессоры начинают выполнение программы с запуска цикла, суммирующего их поднабор чисел:

```
sum[Pn] = 0;
for (i = 1000*Pn; i < 1000*(Pn+1); i = i + 1)
    sum[Pn] = sum[Pn] + A[i]; /* сумма заданных областей */
```

Следующим шагом будет суммирование этого множества частичных сумм. Этот шаг называется **редукцией**. Мы разделяем, чтобы властвовать. Половина процессоров складывает пары частичных сумм, а затем четверть процессоров складывает пары новых частичных сумм, и так далее, пока не будет получена единственная, окончательная сумма. На рис. 6.2 показана иерархическая природа этой редукции.

В данном примере два процессора должны быть синхронизированы еще до того, как «потребляющий» процессор попытается прочитать результат из того места в памяти, куда ведет запись «производящий» процессор, в противном случае потребитель может прочитать старое значение данных. Нужно, чтобы каждый процессор имел свою собственную версию переменной счетчика цикла i, поэтому мы должны обозначить ее как «закрытую» переменную. Программный код выглядит следующим образом (переменная half также является закрытой):

```
half = 100; /* 100 процессоров в мультипроцессоре */
repeat
    synch();    /* ожидание вычисления частичной
сумма */
    if (half%2 != 0 && Pn == 0)
        sum[0] = sum[0] + sum[half-1];
        /* Условная сумма нужна в том случае, когда
        половина имеет нечетное
        значение, недостающий элемент получает
        Процессор0 */
        half = half/2; /* черта, разделяющая
        суммируемые значения */
        if (Pn < half) sum[Pn] = sum[Pn] +
        sum[Pn+half];
            until (half == 1); /* выход
            с окончательной суммой в Sum[0] */
```

Синхронизация
Процесс координации поведения двух и более процессоров, который может быть запущен на других процессорах.

Блокировка
Устройство синхронизации, позволяющее в отдельно взятый момент времени иметь доступ к данным только одному процессору.

Редукция
Функция, которая обрабатывает структуру данных и возвращает одно значение.

Самопроверка

Определите истинность или ложность следующего утверждения: невозможно добиться преимуществ от параллелизма на уровне заданий на мультипроцессорах с общей памятью.

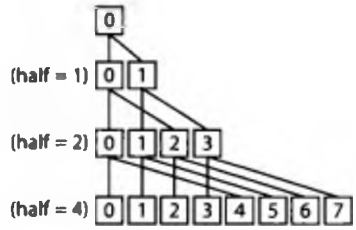


Рис. 7.2. Последние четыре уровня редукции, суммирующей результаты, полученные от каждого процессора снизу вверх. Для всех процессоров, чей номер i меньше значения half, к их сумме прибавляется сумма, полученная на процессоре с номером (i + half)

Уточнение. Альтернативой общему физическому адресному пространству могут быть отдельные физические адресные пространства при общем виртуальном адресном пространстве и возложение на операционную систему работы по обмену данными. Такой подход был испытан, но у него оказался очень высокий уровень издержек по предоставлению программистам работоспособной абстракции общей памяти.

7.4. Кластеры и другие мультипроцессоры с передачей сообщений

Другим подходом к общему адресному пространству является использование процессоров, у каждого из которых имеется свое собственное физическое адресное пространство. На рис. 6.3 показана классическая организация мультипроцессора с множеством отдельных адресных пространств. Этот альтернативный мультипроцессор должен вести обмен данными посредством явной **передачи сообщений**, которая традиционно является названием такого типа компьютеров. При условии, что система имеет подпрограммы для **отправки** и **получения сообщений**, координация выстраивается на передаче сообщений, поскольку один процессор знает, когда сообщение отправлено, а второй, получающий процессор, знает, когда это сообщение прибыло. Если отправителю необходимо получить подтверждение о прибытии сообщения, получивший его процессор может затем отправить ответное уведомительное сообщение в адрес отправителя.

Передача сообщения
Связь между несколькими процессорами путем явной отправки и получения информации.

Подпрограмма отправки сообщения
Подпрограмма, используемая процессором в машинах с отдельной памятью для передачи сообщения другому процессору

Подпрограмма получения сообщения
Подпрограмма, используемая процессором на машинах с отдельной памятью для получения сообщения от другого процессора.

Некоторые параллельные приложения хорошо работают на параллельном оборудовании, независимо от того, предоставляются ли там

общие адреса или передача сообщений. В частности, для параллелизма на уровне задач и нормальной работы приложений с небольшой взаимосвязанностью, например для систем интернет-поиска, почтовых серверов и файловых серверов, не требуется общая адресация.

Было предпринято несколько попыток создания высокопроизводительных компьютеров на основе скоростных сетей передачи сообщений, и они показали лучшую абсолютную производительность обмена данными, чем кластеры, созданные на основе локальных вычислительных сетей. Проблема была в том, что они обходились значительно дороже. Оправдать существенно более высокую стоимость могли только лишь некоторые приложения. Следовательно, именно **кластеры** стали в наши дни наиболее распространенным примером параллельных компьютеров, основанных на передаче сообщений. Кластеры в общем виде представляют собой набор массово производимых компьютеров, объединенных системами ввода-вывода, и стандартных сетевых коммутаторов и кабелей. Каждый из компьютеров работает под управлением отдельной копии операционной системы. Практически каждый интернет-сервис основан на кластерах, собранных из массовых серверов и коммутаторов.

Рис. 7.3. Классическая организация мультипроцессора с множеством отдельных адресных пространств, традиционно называемого мультипроцессором с передачей сообщений. Обратите внимание, что здесь, в отличие от SMP на рис. 7.1, объединяющая сеть находится не между кэш-памятью и оперативной памятью, а между узлами процессор-память

Один из недостатков кластеров заключается в том, что администрирование кластера из n машин стоит примерно столько же сколько администрирование n независимых машин, а стоимость администрирования мультипроцессора с общей памятью, имеющего n процессоров, примерно равна стоимости администрирования одной машины.

Это слабое место является одной из причин популярности виртуальных машин (глава 5), поскольку они упрощают администрирование кластеров. Например, виртуальные машины позволяют останавливать или запускать программы в атомарном режиме, что упрощает обновление программного обеспечения. Виртуальные машины могут даже

Кластеры
Наборы компьютеров, ведущих обмен данными через устройства ввода-вывода и стандартные сетевые коммутаторы с целью формирования мультипроцессоров, основанных на передаче сообщений

переместить программу с одного компьютера в кластере на другой без ее остановки, позволяя программе переместиться с отказавшего оборудования.

Еще один недостаток кластеров заключается в том, что процессоры в кластере обычно объединяются через устройства ввода-вывода каждого компьютера, тогда как ядра в мультипроцессоре обычно объединяются через внутренние схемы памяти компьютера. У внутренних схем памяти высокая пропускная способность и низкий показатель латентности, допускающие намного более высокую производительность при обмене данными.

И последний недостаток — это издержки от деления памяти: кластер из n машин имеет n независимых устройств памяти и n копий операционной системы, а мультипроцессоры с общей памятью позволяют отдельной программе использовать почти всю память компьютера, и для них нужна только одно копия операционной системы.

Упражнения

Эффективное использование памяти
Предположим, что у процессора с единой общей памятью имеется 20 Гбайт оперативной памяти, а у каждого из пяти объединенных в кластер компьютеров имеется 4 Гбайт и операционная система, занимающая 1 Гбайт. Насколько больше пространства достается пользователям при использовании общей памяти?

Ответ

Соотношение памяти, доступной пользовательским программам на компьютере с общей памятью и на кластере, будет равно

$$\frac{20-1}{5\times(4-1)}=\frac{19}{15}=1,25.$$

следовательно, у компьютера с общей памятью примерно на 25% больше пространства памяти.

Давайте еще раз выполним упражнение по суммированию из предыдущего раздела, чтобы изучить влияние нескольких пространств памяти и явно заданный обмен данными.

Упражнения

Простая программа, выполняющаяся в параллельном режиме, предназначенная для мультипроцессора с передачей сообщений
Предположим, что нужно сложить 100 000 чисел на мультипроцессоре с передачей сообщений, имеющем 100 процессоров, у каждого из которых свое собственное устройство памяти.

Ответ

Поскольку у этого компьютера несколько адресных пространств, прежде всего нужно распространить 100 поднаборов среди всех локальных устройств памяти. Процессор, содержащий 100 000 чисел, отправляет поднаборы каждому из 100 узлов «процессор—память».

Следующим шагом будет получение суммы каждого поднабора. Этот шаг заключается в простом цикле, выполняемом на каждом процессоре: чтение слова из локальной памяти и прибавление его к локальной переменной:

```
sum = 0;
for (i = 0; i<1000; i = i + 1) /* перебор элементов каждого массива */
    sum = sum + AN[i];  /* суммирование значений элементов локальных
                           массивов */
```

Последним шагом станет редукция, складывающая эти 100 частичных сумм. Сложность в том, что каждая частичная сумма расположена на другом процессоре. Следовательно, нужно воспользоваться объединяющей сетью для отправки частичных сумм с целью аккумулирования итоговой суммы. Вместо отправки всех частичных сумм на отдельный процессор, мы займемся последовательным сложением частичных сумм.

Сначала половина процессоров посылает свои частичные суммы другой половине процессоров, где эти частичные суммы складываются вместе. Затем одна четверть процессоров (половина от половины) посылает эти новые частичные суммы другой четверти процессоров (оставшейся половине от половины) для следующего раунда суммирования. Это деление пополам, отправка и получение продолжаются до тех пор, пока не будет получена единая сумма всех чисел. Пусть Pn представляет номер процессора, `send(x,y)` является подпрограммой, которая посылает по объединяющей сети процессору `x` значение `y`, а `receive()` является функцией, получающей значение из сети для этого процессора. Тогда код будет иметь следующий вид:

```
limit = 100; half = 100; /* 100 процессоров */
repeat
  half = (half+1)/2; /* граница между отправителями и получателями */
  if (Pn >= half && Pn < limit) send(Pn - half, sum);
  if (Pn < (limit/2)) sum = sum + receive();
  limit = half; /* верхняя граница отправителей */
until (half == 1); /* выход с итоговой суммой */
```

Этот код делит все процессоры на отправителей и получателей, и каждый получающий процессор получает только одно сообщение, поэтому мы можем допускать, что получающий процессор будет простаивать до приема сообщения. Таким образом, отправка и получение могут использоваться в качестве примитивов синхронизации, а также для обмена данными, поскольку процессоры знают о передаче данных.

Если количество узлов нечетное, средний узел не участвует в отправке-получении. Тогда граница (`limit`) устанавливается таким образом, чтобы этот узел стал наивысшим узлом в следующей итерации.

Уточнение. В данном примере подразумевается, что передача сообщения осуществляется практически так же быстро, как и сложение. На самом деле отправка и прием сообщений осуществляются намного медленнее. Для лучшей сбалансированности вычислений и передачи данных можно провести оптимизацию, чтобы множество сумм от других процессоров получало меньшее количество узлов.

Интерфейс аппаратного и программного обеспечения

Разработчикам намного проще проектировать компьютеры, зависящие от передачи сообщений, чем компьютеры, имеющие общую память и обеспечивающие целостность кэш-памяти (см. раздел 5.8 главы 5). Для программистов преимуще-

ство заключается в том, что обмен данными осуществляется в явном виде, значит, будет меньше неожиданностей при выполнении, чем при неявном обмене данными в компьютерах с общей памятью и поддержанием целостности кэш-памяти. А недостатком для программистов является усложненный перенос последовательной программы на компьютер, основанный на передаче сообщений, поскольку каждый обмен данными должен быть идентифицирован заранее, иначе программа не будет работать. Общая память с поддержанием целостности данных кэш-памяти позволяет оборудованию определить, какие данные должны быть переданы, что облегчает перенос программы. Относительно наикратчайшего пути к высокой производительности существуют разные мнения, учитывающие все за и против неявного обмена данными.

Недостатки использования отдельных устройств памяти превращаются в достоинства относительно готовности системы. Поскольку кластер состоит из независимых компьютеров, связанных по локальной сети, то, по сравнению с SMP, заменить машину без остановки системы намного проще. Вообще-то, использование общих адресов означает, что без героических усилий операционной системы изолировать и заменить процессор очень трудно. Поскольку программное обеспечение кластера находится на уровне, работающем поверх локальных операционных систем, сломавшуюся машину отключить и заменить намного проще.

При условии, что кластеры создаются из целых компьютеров и независимых, масштабируемых сетей, эта изолированность также упрощает расширение системы без остановки приложения, запущенного на верхнем уровне кластера.

Более низкая стоимость, высокая готовность, улучшенные параметры энергопотребления и возможность быстрого дополнительного расширения сделали кластеры привлекательными для компаний, предоставляющих сервисы во Всемирной паутине.

Эта технология применяется в поисковых машинах, которыми пользуются каждый день миллионы людей. У таких компаний, как eBay, Google, Microsoft, Yahoo и многих других, имеется множество дата-центров, в каждом из которых работают десятки тысяч процессоров. Вполне очевидно, что использование множества процессоров в компаниях, предоставляющих интернет-сервис, оказалось чрезвычайно успешным.

Уточнение. Еще одной формой крупномасштабных вычислений являются grid-вычисления, где компьютеры разбросаны по большим площадям, из-за чего запущенная в такой системе программа должна осуществлять обмен данными по сетям большой протяженности. Наиболее популярная и уникальная форма grid-вычислений была впервые представлена в рамках проекта SETI@home. Было замечено, что в отдельно взятый момент времени миллионы персональных компьютеров простаивают, не выполняя никакой полезной работы, и они могут быть позаимствованы и с пользой задействованы, если кто-нибудь разработает программное обеспечение, которое может быть запущено на этих компьютерах, после чего даст каждому персональному компьютеру независимую часть задачи, над которой тот будет работать. Первым примером была система поиска внеземного разума Search for ExtraTerrestrial Intelligence (SETI). На SETI@home зарегистрировалось более 5 млн компьютеров в более чем

200 странах, и они внесли суммарный вклад, равный более чем 19 млрд часов вычислительного времени. На конец 2006 года grid-система SETI@home работала со скоростью 257 Терафлопс.

Самопроверка

1. Определите истинность или ложность следующего утверждения: как и SMP-системы, компьютеры с передачей сообщений основаны на блокировках, осуществляемых с целью синхронизации.
2. Определите истинность или ложность следующего утверждения: в отличие от SMP-систем, компьютеры с передачей сообщений нуждаются во множестве копий параллельно выполняемой программы и операционной системы.

7.5. Аппаратная многопоточность

Аппаратная **многопоточность** позволяет нескольким потокам совместно использовать функциональные блоки одного и того же процессора. Чтобы разрешить подобное совместное использование, процессор должен копировать независимое состояние каждого потока. Например, у каждого потока должна быть отдельная копия файла регистров и счетчика команд. Память как таковая должна совместно использоваться посредством механизмов виртуальной памяти, которые уже имеют мультипрограммную поддержку. Кроме этого оборудование должно поддерживать возможность достаточно быстрого переключения между разными потоками. В частности, переключение потока должно быть намного более эффективным, чем переключение процессов, для которого обычно требуются от сотен до тысяч процессорных циклов, а переключение потоков должно быть мгновенным.

Существуют два основных подхода к аппаратной многопоточности. **Мелкомодульная многопоточность** (Fine-grained multithreading) производит переключение между потоками на каждой инструкции, что приводит к чередующемуся выполнению нескольких потоков. Это чередование часто осуществляется по кругу с пропуском любых приостановленных потоков. Чтобы мелкомодульная многопоточность стала практичной, процессор должен быть в состоянии переключать потоки с каждым тактовым циклом. Одно из основных преимуществ мелкомодульной многопоточности заключается в том, что она может скрадывать потери пропускной способности, возникающие как из-за коротких, так и из-за длительных приостановок, поскольку пока один из потоков приостановлен, будут выполняться инструкции других потоков. Основной недостаток мелкомодульной многопоточности заключается в том, что она замедляет выполнение отдельно взятых потоков, поскольку поток, готовый к выполнению без остановок, будет замедлен инструкциями других потоков.

Аппаратная многопоточность
Увеличение времени использования процессора за счет переключения на другой поток, когда работа текущего потока приостанавливается.

Мелкомодульная многопоточность
Версия аппаратной многопоточности, предлагающая переключаться между потоками после каждой инструкции.

В качестве альтернативы мелкомодульной многопоточности была придумана **крупномодульная многопоточность** (coarse-grained multithreading). При использовании крупномодульной многопоточности потоки переключаются только при значительных приостановках, например при промахах при обращении к кэш-памяти второго уровня. Это изменение освобождает от необходимости не иметь никаких затрат на переключение потоков и существенно снижает вероятность замедления выполнения отдельного потока, поскольку инструкции из других потоков могут быть запущены, только когда попадется весьма затратная приостановка. Но крупномодульная многопоточность страдает от другого существенного недостатка: она ограничена в возможностях преодоления потерь пропускной способности, особенно от коротких приостановок. Это ограничение является результатом стоимости конвейерного запуска крупномодульной многопоточности. Поскольку процессор с крупномодульной многопоточностью выдает инструкции из одного потока, то в случае приостановки конвейер должен быть очищен или заморожен. Новый поток, который начинает выполняться после приостановки, должен заполнить конвейер до того, как инструкции получат возможность завершаться. Из-за этих пусковых издержек крупномодульная многопоточность больше всего подходит для сокращения издержек дорогостоящих приостановок, когда перезаполнением конвейера на фоне времени приостановки можно пренебречь.

Параллельная многопоточность (simultaneous multithreading, SMT) является вариантом аппаратной многопоточности, которая использует ресурсы процессора с параллельным запуском инструкций и динамической диспетчеризацией с тем, чтобы воспользоваться параллелизмом на уровне потоков и в то же время воспользоваться параллелизмом на уровне инструкций. Основной замысел, ставший поводом для разработки SMT, заключается в том, что процессоры с параллельным запуском инструкций зачастую обладают более высокой степенью параллелизма функциональных блоков, чем та, которую может эффективно использовать один поток. Более того, используя переименование регистров и динамическую диспетчеризацию, несколько инструкций из разных потоков могут быть запущены без учета их взаимозависимости; разрешение зависимостей может быть произведено за счет возможностей динамической диспетчеризации.

В силу своей зависимости от существующих динамических механизмов, SMT не производит переключения ресурсов при каждом цикле. Вместо этого SMT всегда выполняет инструкции из нескольких потоков, поручая оборудованию связывать слоты инструкций и переименовывать регистры в соответствии с потоками, которым они принадлежат.

Крупномодульная многопоточность
Версия аппаратной многопоточности, предлагающая переключение между потоками только после существенных событий, таких как промахи при обращении к кэш-памяти.

Параллельная многопоточность
Версия многопоточности, снижающая ее стоимость за счет использования ресурсов, необходимых для параллельного запуска нескольких инструкций и микроархитектуры динамической диспетчеризации.

На рис. 6.4 концептуально проиллюстрированы различия в возможностях процессоров использовать суперскалярные ресурсы для следующих процессорных конфигураций. В верхней части рисунка показано, как четыре потока будут выполняться независимо на суперскаляр-

ном процессоре, не поддерживающем многопоточность. В нижней части рисунка показано, как четыре потока могут быть объединены с максимальной эффективностью с использованием следующих схем:

- суперскаляр с крупномодульной многопоточностью;
- суперскаляр с мелкомодульной многопоточностью;
- суперскаляр с параллельной многопоточностью.

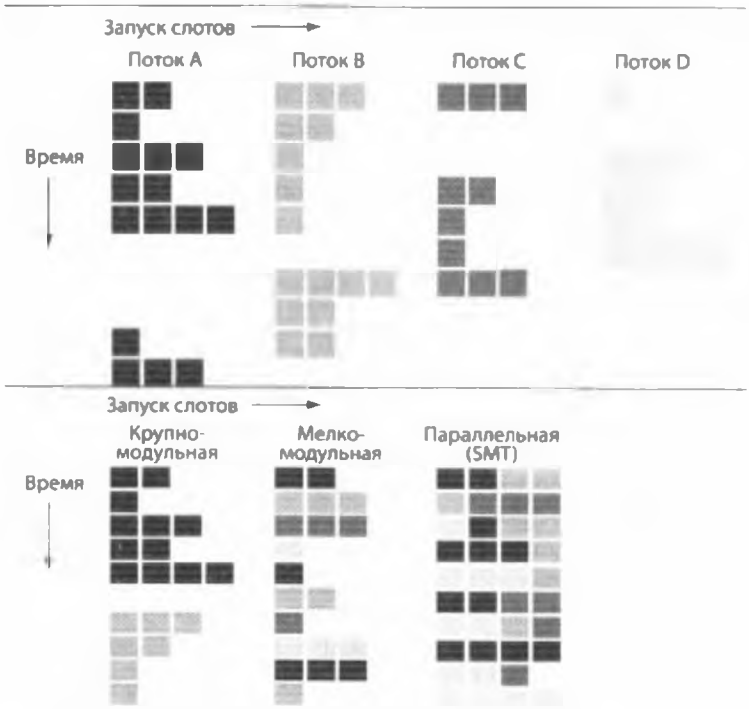


Рис. 7.4. Как четыре потока используют запуск слотов суперскалярного процессора при различных подходах. В верхней части рисунка для четырех потоков показано, как каждый из них будет исполняться, будучи запущенным в одиночестве на стандартном суперскалярном процессоре без поддержки многопоточности. В трех примерах в нижней части рисунка показано, как они будут выполняться при совместном запуске в трех многопоточных вариантах. По горизонтали отображается команда в каждый тактовый цикл. По вертикали представлена последовательность тактовых циклов. Пустые (белые) квадраты показывают, что в данном тактовом цикле соответствующий запущенный слот не задействован. Градации серого цвета в многопоточном процессоре соответствуют четырем разным потокам. Дополнительный пусковой эффект конвейера для крупномодульной многопоточности, не показанный на данном рисунке, приведет при ее использовании к еще большим потерям пропускной способности

В суперскаляре без аппаратной поддержки многопоточности использование слотов запуска ограничено недостатком параллелизма на уровне инструкций. Кроме того, существенные приостановки, возникающие, например, из-за промахов при обращении к кэш-памяти, приведут к простаиванию процессора.

В суперскаляре с крупномодульной многопоточностью длительные приостановки частично скрадываются за счет переключения на другой поток, который использует ресурсы процессора. Хотя это и сокращает количество тактовых циклов, приходящихся на полный простой, издержки запуска конвейера все же приводят к пустым циклам, а ограничения параллелизма на уровне инструкций означают, что все запущенные слоты использовать не удастся. В случае мелкомодульной многопоточности чередование потоков существенно сокращают количество полностью пустых слотов. Но, поскольку инструкции в отдельно взятом тактовом цикле запускает только один поток, ограничения в параллелизме на уровне команд все еще приводят к пустым слотам внутри некоторых тактовых циклов.

В случае использования SMT используются как параллелизм на уровне потоков, так и параллелизм на уровне инструкций, при этом запуск слотов в одном и том же тактовом цикле осуществляется сразу несколькими потоками. В идеале использование запущенного слота ограничено диспропорциями потребностей в ресурсах и возможностью их использования множеством потоков. На практике количество используемых слотов может быть ограничено и другими факторами. Хотя рис. 7.4 демонстрирует упрощенный взгляд на реальную работу этих процессоров, на нем хорошо видны потенциальные преимущества производительности многопоточности в целом и SMT в частности. Например, один из последних многоядерных процессоров Intel Nehalem для повышения степени использования ядра поддерживает SMT с двумя потоками.

Давайте подведем итог. Во-первых, из главы 1 мы узнали, что энергетический «барьер» заставил конструкторов двигаться в сторону упрощения и увеличения количества экономичных процессоров на одном кристалле. Может так удачно сложиться, что объем недовостребованных ресурсов процессоров с изменением последовательности выполнения инструкций может быть сокращен, и будут использованы упрощенные формы многопоточности. Например, микропроцессор Sun UltraSPARC T2 (Niagara 2), рассматриваемый в разделе 7.11, является свидетельством возврата к более простой микроархитектуре и, следовательно, использования мелкомодульной многопоточности.

Во-вторых, основной проблемой производительности является допустимость латентности, возникающей из-за промахов при обращении к кэш-памяти. Компьютеры с мелкомодульной многопоточностью, подобные UltraSPARC T2, при промахе переключаются на другой поток, что, возможно, более эффективно скрадывает латентность памяти, чем попытка заполнения неиспользуемых слотов, как в SMT.

Третье наблюдение состоит в том, что целью аппаратной многопоточности является более эффективное использование оборудования за счет распределения компонентов между разными задачами. Ресурсы распределяются и в многоядерных конструкциях. Например, два процессора могут совместно использовать блок вычислений с плавающей точкой или кэш-память уровня L3. Такое совместное использование уменьшает некоторые преимущества многопоточности по сравнению с предоставлением множества ядер, не обладающих многопоточностью.

Самопроверка

1. Определите истинность или ложность следующего утверждения: как многопоточность, так и многоядерность основаны на параллелизме с целью получения более эффективной отдачи от микропроцессора.
3. Определите истинность или ложность следующего утверждения: в параллельной многопоточности потоки используются для улучшения использования ресурсов процессоров с динамической диспетчеризацией и изменением порядка выполнения инструкций.

7.6. SISD, MIMD, SIMD, SPMD и использование векторов

Другая классификация параллельного оборудования была предложена в 1960-е годы и используется и в наши дни. Она была основана на количестве потоков инструкций и количестве потоков данных. Категории согласно этой классификации показаны в табл. 7.2. Соответственно, обычная однопроцессорная система имеет один поток инструкций и один поток данных, а обычный мультипроцессор имеет несколько потоков инструкций и несколько потоков данных. Для этих двух категорий используются, соответственно, аббревиатуры **SISD** и **MIMD**.

Таблица 7.2. Классификация оборудования и примеры, основанные на количестве потоков инструкций и потоков данных: SISD, SIMD, MISD и MIMD

		Потоки данных	
		Один	**Несколько**
Потоки инструкций	**Один**	SISD: Intel Pentium 4	SIMD: SSE инструкции x86
	Несколько	MISD: Пока примеров нет	MIMD: Intel Xeon e5345 (Clovertown)

Можно, конечно, написать отдельные программы, работающие на разных процессорах на MIMD-компьютере и все же работающие совместно над достижением более важной, согласованной цели, но программисты обычно пишут одну программу, запускаемую на всех процессорах MIMD-компьютера, полагаясь на инструкции условных переходов, когда разные процессоры должны выполнить разные фрагменты кода. Этот стиль называется **одна программа, несколько потоков данных** (Single Program Multiple Data, **SPMD**), но он является всего лишь вполне обычным способом программирования MIMD-компьютера.

SISD
Единственный поток инструкций, единственный поток данных (Single Instruction stream, Single Data stream). Одноядерный процессор.

MIMD
Несколько потоков инструкций, несколько потоков данных (Multiple Instruction streams, Multiple Data streams). Мультипроцессор.

SPMD
Одна программа, несколько потоков данных (Single Program Multiple Data). Обычная модель MIMD-программирования, где одна программа запускается сразу на всех процессорах.

Привести примеры компьютеров, подпадающих под категорию «несколько потоков инструкций, один поток данных» (MISD), довольно трудно, в инвертированном «варианте» смысла намного больше. **SIMD**-компьютеры работают с векторами данных. Например, одна SIMD-инструкция может сложить 64 числа, отправив 64 потока данных по адресу 64 АЛУ для формирования 64 сумм за один тактовый цикл.

Достоинства SIMD состоят в том, что все параллельные исполнительные устройства синхронизированы, и все они реагируют на одну и ту же инструкцию, которая выдается при работе одного и того же счетчика команд (PC). С точки зрения программиста это весьма близко к уже знакомому SISD. Хотя каждое исполнительное устройство будет выполнять одну и ту же инструкцию, у каждого из них есть свои собственные регистры адресов, и поэтому у каждого устройства разные адреса данных. Таким образом, в терминах табл. 7.1, последовательное приложение могло бы быть откомпилировано для работы на последовательном оборудовании с организацией SISD или на параллельном оборудовании, имеющем организацию SIMD.

Первоначальной мотивацией создания SIMD была амортизация стоимости блока управления нескольких десятков исполнительных устройств. Еще одно преимущество состоит в сокращении размера памяти программы, поскольку для SIMD нужна только одна копия одновременно выполняемого кода, в то время как для MIMD-систем с передачей сообщений может понадобиться копия на каждом процессоре, а для MIMD-систем с общей памятью понадобится несколько устройств кэш-памяти инструкций.

SIMD-система лучше всего работает с массивами в циклах `for`. Следовательно, чтобы параллелизм работал в SIMD, должно быть большое количество одинаковых структур данных, что называется **параллелизмом на уровне данных**. Самым слабым местом SIMD-системы являются инструкции `case` или `switch`, где каждое исполнительное устройство должно выполнять над своими данными разные операции, в зависимости от тех данных, которыми она располагает. Исполнительные устройства с неверными данными блокируются, чтобы устройства с приемлемыми данными могли продолжить работу. В таких ситуациях, по сути, достигается производительность, равная $1/n$, где n представляет собой количество случаев неверных данных.

Так называемые матричные процессоры, способствовавшие появлению категории SIMD, постепенно канули в Лету, но две текущие интерпретации SIMD остаются активными и по сей день.

SIMD
Один поток инструкций, несколько потоков данных (Single Instruction stream, Multiple Data streams). Мультипроцессор. Одна и та же инструкция применяется к нескольким потокам данных, как в векторном или матричном процессоре.

Параллелизм на уровне данных
Параллелизм, достигаемый за счет работы с независимыми данными.

SIMD в x86: мультимедийные расширения

Наиболее широко используемый вариант SIMD-системы сегодня имеется практически в каждом микропроцессоре и составляет основу

сотен MMX- и SSE-инструкций микропроцессора x86 (см. главу 2). Эти инструкции были добавлены для повышения производительности мультимедийных программ, и дают возможность оборудованию располагать множеством одновременно работающих АЛУ, или, что одно и то же, поделить одно широкое АЛУ на множество параллельных, меньших по размеру АЛУ, работающих одновременно. Например, один и тот же компонент оборудования можно рассматривать как одно 64-разрядное АЛУ, или как два 32-разрядных АЛУ, или как четыре 16-разрядных АЛУ, или как восемь 8-разрядных АЛУ. Загрузки и сохранения имеют просто такую же ширину, как и самое широкое АЛУ, поэтому программист может относиться к одним и тем же инструкциям передачи данных как к передаче 64-разрядных элементов данных, либо двух 32-разрядных элементов данных, либо четырех 16-разрядных элементов данных, либо восьми 8-разрядных элементов данных.

Этот очень дешевый параллелизм для ограниченных целочисленных данных послужил исходным толчком для создания инструкций MMX семейства процессоров x86. Под влиянием закона Мура к этим мультимедийным расширениям было добавлено еще больше оборудования, и теперь инструкции SSE2 поддерживают одновременное выполнение действий сразу над двумя 64-разрядными числами с плавающей точкой.

Ширина операции и регистров закодирована в поле кода операции (opcode) этих мультимедийных инструкций. По мере расширения данных регистров и операций число кодов операций для мультимедийных инструкций стремительно растет, и теперь уже насчитываются сотни SSE-инструкций, позволяющих выполнять весьма полезные комбинации действий (см. главу 2).

Вектор

Более старая и элегантная интерпретация SIMD называется векторной архитектурой, она тесно связана с продукцией компании Cray Computers. Здесь опять наблюдается довольно сильная связь с проблемами широкого применения параллелизма на уровне данных. Вместо использования 64 АЛУ, выполняющих одновременно 64 сложения, подобно старым матричным процессорам, векторная архитектура конвейеризирует АЛУ, чтобы получить высокую производительность при меньших затратах. Основная философия векторной архитектуры заключается в сборе элементов данных из памяти, помещении их в определенном порядке в большой набор регистров, проведении их последовательной обработки в регистрах, а затем в записи результатов обратно в память. Основным свойством векторных архитектур является набор векторных регистров. Поэтому векторная архитектура может иметь 32 векторных регистра, в каждом из которых 64 64-разрядных элемента.

Упражнения

Сравнение векторов с обычным кодом

Предположим, что мы расширяем архитектуру набора инструкций MIPS векторными инструкциями и векторными регистрами. Векторные операции используют те же названия, что и MIPS-операции, но с добавлением буквы «V». Например, `addv.d` складывает два вектора двойной точности. Векторные инструкции получают свои входные данные либо

из пары векторных регистров (addv.d), либо из векторного регистра и скалярного регистра (addvs.d). В последнем случае значение в скалярном регистре используется в качестве ввода для всех операций — операция addvs.d будет прибавлять содержимое скалярного регистра к каждому элементу в векторном регистре. Названия lv и sv обозначают векторную загрузку и векторное сохранение, и эти инструкции загружают или сохраняют целый вектор данных с двойной точностью. Один операнд определяет векторный регистр, подвергаемый загрузке или сохранению, а второй операнд, являющийся MIPS-регистром общего назначения, содержит стартовый адрес вектора в памяти. Исходя из этого короткого описания покажите обычный MIPS-код в сравнении с векторным MIPS-кодом для

$$Y = a \times X + Y,$$

где X и Y являются векторами из 64 чисел с плавающей точкой двойной точности, которые изначально находятся в памяти, а a является скалярной переменной двойной точности. (Этот пример является так называемым циклом DAXPY, который формирует внутренний цикл контрольной задачи Linpack. DAXPY означает double precision $a \times X$ plus Y.) Предположим, что начальные адреса X и Y находятся соответственно в $s0 и $s1.

Ответ

Вот как выглядит обычный MIPS-код для DAXPY:

```
        l.d     $f0,a($sp)      ;загрузка скаляра a
        addiu   r4,$s0,#512     ;верхняя граница того, что загружать
loop:   l.d     $f2,0($s0)      ;загрузка x(i)
        mul.d   $f2,$f2,$f0     ;a × x(i)
        l.d     $f4,0($s1)      ;загрузка y(i)
        add.d   $f4,$f4,$f2     ;a × x(i) + y(i)
        s.d     $f4,0($s1)      ;сохранение в y(i)
        addiu   $s0,$s0,#8      ;увеличение индекса к x
        addiu   $s1,$s1,#8      ;увеличение индекса к y
        subu    $t0,r4,$s0      ;вычисление границы compute bound
        bne     $t0,$zero,loop  ;проверка на готовность
```

А вот как выглядит векторный MIPS-код для DAXPY:

```
        l.d     $f0,a($sp)      ;загрузка скаляра a
        lv      $v1,0($s0)      ;загрузка вектора x
        mulvs.d $v2,$v1,$f0     ;перемножение вектора и скаляра
        lv      $v3,0($s1)      ;загрузка вектора y
        addv.d  $v4,$v2,$v3     ;добавление y к произведению
        sv      $v4,0($s1)      ;сохранение результата
```

В этом примере есть ряд интересных сравнений двух сегментов кода. Наиболее ярким является то, что векторный процессор существенно сокращает диапазон инструкций, выполняя только шесть инструкций против почти 600 для MIPS. Это сокращение получается из-за того, что векторные операции работают с 64 элементами, а также из-за того, что верхние команды, которые составляют на MIPS почти половину цикла, в векторном коде отсутствуют. Не удивительно, что это уменьшает количество извлекаемых инструкций и время их выполнения, экономя энергию.

Еще одним важным различием является частота конфликтов конвейера (см. главу 4). В простом MIPS-коде каждая инструкция add.d должна ждать выполнения инструкции mul.d, и каждая инструкция s.d должна ждать выполнения инструкции add.d. В векторном процессоре каждая векторная инструкция будет приостанав-

ливаться только для первого элемента каждого вектора, а затем последующие элементы будут ровно следовать вниз по конвейеру. Таким образом, задержки конвейера требуются только один раз для каждой векторной операции, а не один раз для каждого векторного элемента. В этом примере частота задержек конвейера в MIPS будет примерно в 64 раза выше, чем в VMIPS. Задержки конвейера могут быть сокращены в MIPS путем использования развертывания цикла (см. главу 4). Но большую разницу в диапазоне инструкций сократить не удастся.

Уточнение. В предыдущем примере цикл точно совпадает с длиной вектора. Когда циклы короче, векторная архитектура использует регистр для ограничения длины векторной операции. Когда циклы длиннее, мы добавляем отслеживающий код, чтобы итерационно обработать всю векторную графику. Этот последний процесс называют *разрывом итерации* (strip mining).

Сравнение векторов со скалярами

Векторные инструкции обладают рядом важных свойств по сравнению с обычной архитектурой набора инструкций, называемой в данном контексте *скалярной архитектурой*:

- Одной векторной инструкцией определяется большой объем работы, она эквивалентна выполнению целого цикла. Необходимый диапазон извлекаемых и декодируемых инструкций существенно сужается.
- За счет использования векторных инструкций компилятор или программист показывают, что вычисление каждого результата в векторе независимо от вычисления других результатов в этом же векторе, поэтому оборудованию не нужно вести проверку на наличие конфликтов данных внутри векторной инструкции.
- Векторные архитектуры и компиляторы прославились тем, что по сравнению с многопроцессорными MIMD-системами они позволяют существенно облегчить создание эффективных приложений при наличии параллелизма на уровне данных.
- Проверку на наличие конфликтов данных оборудованию нужно вести только лишь между двумя векторными инструкциями, и делать это для каждого векторного операнда, а не для каждого элемента внутри векторов. Сокращение проверок также ведет к экономии энергии.
- Векторные инструкции, обращающиеся к памяти, имеют известную схему обращения. Если все элементы векторов примыкают друг к другу, то извлечение вектора из набора банков памяти с развитым чередованием адресов работает очень эффективно. Поэтому потери на латентность при обращении к оперативной памяти возникают только один раз на вектор, а не на каждое слово в векторе.
- Поскольку весь цикл заменяется векторной инструкцией с предопределенным поведением, конфликты управления, которые, как правило, будут возникать из-за условных переходов в цикле, отсутствуют.
- Экономия на диапазоне инструкций и на проверке наличия конфликтов плюс эффективное использование диапазона памяти дают векторной архитектуре

преимущества по сравнению со скалярной архитектурой в мощности и энергопотреблении.

По этим причинам векторные операции могут проводиться быстрее, чем последовательность скалярных операций с одинаковым количеством элементов данных, и разработчики имеют вполне серьезные основания для включения векторных блоков.

Сравнение векторов и мультимедийных расширений

Как и мультимедийные расширения, имеющиеся в инструкциях x86 SSE, векторные инструкции определяют сразу несколько операций. Но мультимедийные расширения обычно определяют лишь небольшое их количество, в то время как векторные определяют десятки операций. В отличие от мультимедийных расширений, количество элементов в векторной операции определяется не в коде операции, а в отдельном регистре. Это означает, что разные версии векторной архитектуры могут быть реализованы с разным количеством элементов путем простого изменения содержимого этого регистра и, следовательно, сохранить двоичную совместимость. В отличие от этого, новый более крупный набор кодов операций всякий раз добавляет в мультимедийное расширение архитектуры x86 изменения длины вектора.

Также, в отличие от мультимедийных расширений, передача данных не нуждается в их смежности. Векторы поддерживают как пошаговые доступы, при которых оборудование загружает в память каждый *n*-ный элемент данных, так и индексированные доступы, при которых оборудование находит адреса загружаемых элементов в векторном регистре.

Как и мультимедийные расширения, вектор легко приспосабливается к гибкости в длине данных, поэтому заставить операции работать с 32 64-разрядными элементами данных, или с 64 32-разрядными элементами данных, или со 128 16-разрядными элементами данных, или с 256 8-разрядными элементами данных довольно просто.

В общем, векторные архитектуры являются довольно эффективным средством выполнения программ, занимающихся параллельной обработкой данных, они лучше соответствуют технологиям компилирования, нежели мультимедийные расширения, и им проще развиваться с течением времени, чем мультимедийным расширениям для архитектуры x86.

Самопроверка

Определите истинность или ложность следующего утверждения: как поясняется в x86, мультимедийные расширения можно представить в виде векторной архитектуры с короткими векторами, поддерживающей передачу только примыкающих друг к другу векторных данных.

Уточнение. Почему же имеющие такие преимущества вектора не стали более популярными за пределами высокопроизводительных вычислений? Высказывались серьезные беспокойства насчет того, что векторные регистры увеличивают время переключения контекста, а также насчет сложности обработки ошибок отсутствия страниц при загрузках и сохранениях векторов, и утверждалось, что SIMD-инструкции достигли некоторых преимуществ векторных инструкций. Но в последних новостях компании Intel были намеки на то, что роль векторов будет расти. Технология INTEL под названием «усовершенствованные векторные инструкции» Advanced Vector Instructions (AVI), появление которой намечалось на 2010 год, сможет мгновенно расширять SSE-регистры со 128 до 256 разрядов и допускать их расширение вплоть до 1024 разрядов. Последнее расширение эквивалентно 16 числам с плавающей точкой, имеющим двойную точность. Пока непонятно, будут ли при этом представлены инструкции загрузки и сохранения векторов. Кроме того, считается, что в процессоре Larrabee, проникновение Intel на рынок дискретных графических процессоров имеются векторные инструкции.

Уточнение. Еще одно преимущество векторов и мультимедийных расширений состоит в относительной простоте дополнения архитектуры набора скалярных инструкций инструкциями, использующими эти технологии для улучшения производительности операций параллельной обработки данных.

7.7. Введение в графические процессоры

Главным обоснованием добавления SIMD-инструкций к существующим архитектурам послужило то обстоятельство, что многие микропроцессоры в персональных компьютерах и рабочих станциях были подключены к дисплеям, поэтому на графику тратилось все больше вычислительного времени. Поэтому, когда согласно закону Мура количество доступных микропроцессорам транзисторов увеличилось, появился смысл улучшить обработку графики.

Как только закон Мура позволил центральному процессору улучшить обработку графики, он также дал возможность добавить в микросхемы видеографических контроллеров функции ускорения 2D- и 3D-графики. Более того, в числе самых передовых устройств появились графические карты, как правило, компании Silicon Graphics, которые можно было добавлять к рабочим станциям, что позволяло создавать изображения фотографического качества. Эти самые передовые графические карты были популярны при создании сгенерированных компьютером изображений, что позже было перенесено в телевизионную рекламу, а затем в кинопроизводство. Таким образом, у наращивания рабочих ресурсов видеографических контроллеров была вполне конкретная цель, что во многом было похоже на то, как суперкомпьютеры проложили дорогу микропроцессорам.

Большим стимулом для улучшения обработки графики стала индустрия компьютерных игр, развивавшаяся как на базе персональных компьютеров, так и на специальных игровых консолях, таких как Sony PlayStation. Быстро разраставшийся рынок компьютерных игр подтолкнул многие компании на инвестирование все более крупных средств в развитие быстродействующего графического оборудования, таким образом, темпы развития графической обработки стали более высо-

кими по сравнению с темпами развития вычислений общего назначения в серийно выпускаемых микропроцессорах.

Сообщество разработчиков графических устройств и компьютерных игр имело несколько иные цели, чем сообщество разработчиков микропроцессоров, и выработало свой собственный стиль обработки данных и свою собственную терминологию. Как только графические процессоры стали мощнее, их начали называть *графическими процессорными устройствами* (Graphics Processing Unit, *GPU*), чтобы отличать их от центральных процессорных устройств, CPU. Рассмотрим некоторые ключевые характеристики, благодаря которым графические процессоры — GPU отличаются от центральных процессоров — CPU:

- GPU являются ускорителями, дополняющими CPU, поэтому им не нужно уметь выполнять все задачи, присущие CPU. Эта роль позволяет им посвятить все свои ресурсы графике. Для GPU вполне нормально выполнять некоторые задачи недостаточно хорошо или вообще не выполнять, при условии, что в системе, где есть и CPU, и GPU, CPU может при необходимости эти задачи выполнить. Таким образом, комбинация CPU-GPU является одним из примеров *гетерогенной мультипроцессорной обработки*, где не все процессоры являются идентичными. (Еще одним примером может послужить архитектура IBM Cell, рассматриваемая в разделе 7.11, которая также была разработана для ускорения 2D- и 3D-графики.)
- Интерфейсами программирования GPU являются высокоуровневые интерфейсы прикладного программирования (application programming interface, API), такие как OpenGL и разработанный компанией Microsoft DirectX, в совокупности с высокоуровневыми языками графического затенения, или шейдинга (shading), такими как разработанный компанией NVIDIA язык C для графики (Cg) и разработанный компанией Microsoft язык высокого уровня для программирования шейдеров — High Level Shader Language (HLSL). Компиляторы языков предназначены не для машинных инструкций, а для промежуточных языков, отвечающих промышленным стандартам. Программный драйвер GPU генерирует оптимизированные машинные инструкции, подходящие к конкретному GPU. Хотя эти API и языки развиваются довольно быстро, охватывая все новые GPU-ресурсы, допускаемые законом Мура, отсутствие необходимости соблюдения обратной совместимости по двоичным инструкциям позволяет GPU-разработчикам исследовать новые архитектуры, не испытывая опасения за постоянные неудачи с реализацией. Такая среда способствует более быстрому появлению нововведений в GPU, чем в CPU.
- Обработка графики предполагает рисование точек (вершин) трехмерных геометрических примитивов, таких как линии и треугольники, и *затенение* (shading) или прорисовку, или рендеринг (rendering), пиксельных фрагментов геометрических примитивов. В видеоиграх, к примеру, рисуется в 20–30 раз больше пикселов, чем вершин.
- Каждая точка (вершина) может быть нарисована независимо, и прорисовка каждого пиксельного фрагмента также может быть произведена независимо. Чтобы быстро прорисовать миллионы пикселов на кадр, создается GPU, способный

параллельно выполнять множество потоков от программ, рисующих вершины и пиксельные затенения.

К типам графических данных относятся точки (вершины), состоящие из координат (x, y, z, w), и пикселы, состоящие из цветов (красный — red, зеленый — green, синий — blue, альфа — alpha). Графические процессоры представляют каждый компонент точки в виде 32-разрядного числа с плавающей точкой. Каждый из четырех компонентов пикселов сначала был 8-разрядным беззнаковым целым числом, но новые графические процессоры теперь представляют каждый компонент в виде числа с плавающей точкой одинарной точности в диапазоне между 0,0 и 1,0.

Рабочий набор может иметь объем в сотни мегабайт, и он не отличается такой же локальностью, связанной со временем, которая присуща данным в обычных приложениях. Более того, графические задачи отличаются большим уровнем параллелизма данных.

Эти различия приводят к различным типам архитектуры:

Возможно, самое большое различие состоит в том, что графические процессоры, в отличие от центральных, не полагаются на использование многоуровневой кэш-памяти для преодоления большой латентности при обращении к памяти. Вместо этого GPU полагаются на наличие достаточно большого количества потоков, позволяющее скрадывать латентность обращения к памяти. То есть между временем выдачи запроса к памяти и временем поступления данных GPU выполняет сотни или тысячи потоков, которые не зависят от этого запроса.

Для достижения высокой производительности GPU полагаются на повсеместный параллелизм, обеспечиваемый множеством параллельных процессоров и множеством параллельных потоков.

Оперативная память GPU в силу этих причин ориентирована на пропускную способность, а не на снижение времени латентности. Для GPU используются отдельные однотипные DRAM-модули с более высокими показателями разрядности и пропускной способности, чем у DRAM-модулей для центральных процессоров. Кроме того, оперативная память графических процессоров традиционно имела меньший объем, чем оперативная память обычных микропроцессоров. В 2008 году у графических процессоров обычно имелось не более 1 Гбайт оперативной памяти, в то время как у центральных процессоров имелось от 2 до 32 Гбайт. И наконец, следует иметь в виду, что для вычислений общего назначения нужно учесть еще время на передачу данных между памятью CPU и памятью GPU, поскольку последний является сопроцессором.

Учитывая, что графические процессоры для достижения высокой полосы пропускания при обращениях к памяти зависят от использования большого количества потоков, в них может быть столько же параллельных процессоров, сколько и потоков. Следовательно, каждый графический процессор является в высокой степени многопоточным устройством.

В прошлом для достижения производительности, необходимой для графических приложений, графические процессоры были основаны на использовании раз-

нотипных специализированных процессоров. Современные графические процессоры держат курс на применение одинаковых универсальных процессоров для предоставления большей гибкости в программировании, что делает их более похожими на многоядерные конструкции, наблюдаемые в вычислительной технике общего назначения.

- Учитывая четырехэлементную природу типов графических данных, графические процессоры исторически имели SIMD-инструкции, подобные инструкциям центральных процессоров. Но современные GPU, чтобы облегчить программирование и повысить эффективность, больше сориентированы на скалярные инструкции.
- В отличие от центральных процессоров в них не было поддержки арифметики чисел с плавающей точкой двойной точности, поскольку для графических приложений она была не нужна. В 2008 году были представлены первые GPU с поддержкой чисел двойной точности. Тем не менее даже на этих новых графических процессорах операции с одинарной точностью по-прежнему будут осуществляться в 8–10 раз быстрее операций с двойной точностью, в то время как разница в производительности для центральных процессоров сводится к положительным сторонам передач в системе памяти меньшего количества байтов в силу использования более узких данных.

Хотя GPU были разработаны для более узкого круга приложений, некоторые программисты задавались вопросом, могли бы они придать своим приложениям такую форму, которая позволила бы им раскрыть высокую потенциальную производительность графических процессоров. Чтобы выделить этот стиль использования графических процессоров, некоторые специалисты назвали его универсальным использованием графических процессоров (General Purpose GPUs, или GPGPUs). После изнурительных попыток решить свои задачи с использованием графических API и языков программирования графических шейдеров, они под влиянием языка C разработали языки, позволившие им создавать программы непосредственно для графических процессоров. В качестве примера можно привести Brook, потоковый язык для графических процессоров. Следующим шагом как в повышении возможностей программирования оборудования, так и в языках программирования стала разработанная компанией NVIDIA архитектура CUDA (Computer Unified Divice Architecture), позволяющая программистам создавать программы на языке C, которые, хотя и с некоторыми ограничениями, но все же могли выполняться на графических процессорах. С повышением программируемости стала расти и степень использования графических процессоров в параллельных вычислениях.

Введение в архитектуру графического процессора NVIDIA

Поскольку графические процессоры развивались в своей собственной среде, у них не только другая архитектура, как было сказано выше, но и другой набор терминов. По мере изучения терминологии графических процессоров вы увидите сходство с подходами, рассмотренными в предыдущих разделах, например сходство с мелкомодульной многопоточностью и векторами.

Чтобы помочь вам с переходом к новым словам, будет представлено краткое введение в термины и идеи, используемые в архитектуре Tesla GPU и в программной среде CUDA.

Отдельная микросхема GPU расположена на карте, которая подключена к персональному компьютеру через разъем шины PCI-Express. Так называемые GPU, встроенные в материнскую плату, интегрированы в чипсет материнской платы, например в северный мост или в южный мост (см. главу 6).

Графические процессоры обычно предлагаются в виде семейства микросхем с разными диапазонами стоимости—производительности, сохраняющими программную совместимость друг с другом. Наборы микросхем на основе Tesla GPU предлагаются в диапазоне от 1 до 16 узлов, которые NVIDIA называет *мультипроцессорами*. В начале 2008 года самая старшая версия называлась GeForce 8800 GTX, имела 16 мультипроцессоров и тактовую частоту 1,35 ГГц. Каждый мультипроцессор имел восемь многопоточных блоков для работы с числами с плавающей точкой одинарной точности и блоки обработки целых чисел, которые NVIDIA называла *потоковыми процессорами*.

Поскольку архитектура включает инструкции умножения-сложения чисел с плавающей точкой одинарной точности, пиковая производительность умножения-сложения одинарной точности чипсета 8800 GTX равна:

$$16\ MP \times \frac{8\ SP}{MP} \times \frac{2\ \text{Флоп/инстр}}{SP} \times \frac{1\ \text{инстр}}{\text{такт}} \times \frac{1{,}35 \times 10^9\ \text{тактов}}{\text{в секунду}} =$$

$$= \frac{16 \times 8 \times 1{,}35\ \text{Гфлоп}}{\text{в секунду}} = \frac{345{,}6\ \text{Гфлоп}}{\text{в секунду}}.$$

Каждый из 16 мультипроцессоров в составе GeForce 8800 GTX имеет локальное программно-управляемое запоминающее устройство объемом 16 Кбайт плюс 8192 32-разрядных регистра. Система памяти 8800 GTX состоит из шести сегментов 900 МГц графической DDR3 DRAM-памяти, каждый шириной 8 байт и объемом 128 Мбайт. Общий объем памяти, таким образом, составляет 768 Мбайт. Пиковая пропускная способности памяти GDDR3 равна

$$6 \times \frac{8\ \text{байт}}{\text{на передачу}} \times \frac{2\ \text{передачи}}{\text{за такт}} \times \frac{0{,}9 \times 10^9\ \text{тактов}}{\text{в секунду}} = \frac{6 \times 8 \times 2 \times 0{,}9\ \text{Гбайт}}{\text{в секунду}} = \frac{86{,}4\ \text{Гбайт}}{\text{в секунду}}.$$

Чтобы скрыть латентность памяти, каждый потоковый процессор имеет аппаратно-поддерживаемые потоки. Каждая группа из 32 потоков называется варпом (warp). Варп является блоком диспетчеризации, и активные потоки в варпе, до 32, выполняются в параллельном режиме SIMD-способом. Но многопоточная система справляется с условиями, позволяя потокам расходиться по разным путям условных переходов. Когда потоки варпа идут по расходящимся путям, варп последовательно выполняет код по обоим маршрутам, делая неактивными некоторые потоки, что приводит к более медленному выполнению активных потоков. Как только условная часть завершится, оборудование снова объединяет потоки в полностью активный варп. Для достижения наивысшей производительности все 32 потока варпа нуждаются в совместном параллельном выполнении. Похожим

образом оборудование также следит за течением адресов, поступающих от разных потоков, чтобы попытаться объединить отдельные запросы в меньшее количество передач более крупных блоков памяти для увеличения производительности при работе с памятью.

На рис. 6.5 продемонстрирована совокупность всех этих свойств и дано сравнение мультипроцессора Tesla с ядром Sun UltraSPARC T2, которое рассматривается в разделах 7.5 и 7.11. В обоих реализована аппаратная многопоточность на основе диспетчеризации потоков по времени, показанная на вертикальной оси. Каждый микропроцессор Tesla состоит из восьми потоковых процессоров, которые выполняют восемь параллельных потоков за тактовый цикл, что показано на горизонтальной оси. Как ранее уже упоминалось, наивысшая производительность достигается, когда все 32 принадлежащих варпу потока выполняются вместе в SIMD-подобной манере, что в архитектуре Tesla называется «одна инструкция — несколько потоков» (single-instruction multiple-thread, SIMT). SIMT в динамическом режиме обнаруживает, какие принадлежащие варпу потоки могут выполнять вместе одну и ту же инструкцию и какие независимые потоки будут простаивать в данном цикле. Ядро T2 содержит только один многопоточный процессор. В каждом цикле он выполняет одну инструкцию для одного потока.

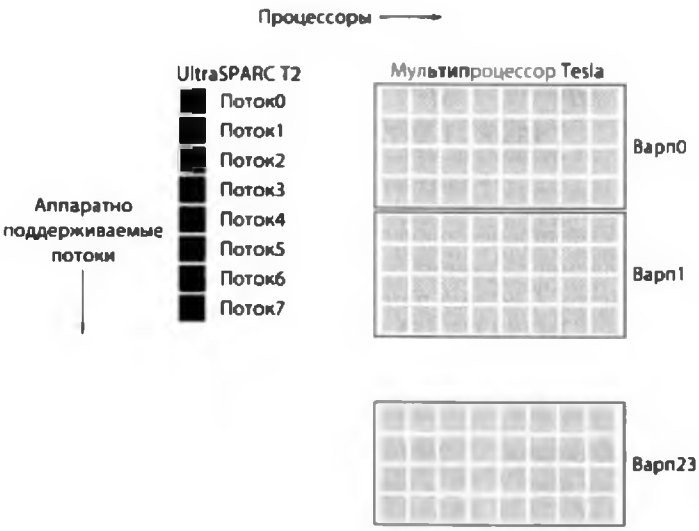


Рис. 7.5. Сравнение одного ядра Sun UltraSPARC T2 (Niagara 2) с одним мультипроцессором Tesla. Ядро T2 является одним процессором и использует аппаратно поддерживаемую многопоточность с восемью потоками. Мультипроцессор Tesla состоит из восьми потоковых процессоров и использует аппаратно поддерживаемую многопоточность с 24 варпами по 32 потока (восемь процессоров умножить на четыре тактовых цикла). T2 может осуществлять переключение каждый тактовый цикл, а Tesla может осуществлять переключение только каждые два или четыре тактовых цикла. Один из способов сравнения этих двух устройств состоит в том, что T2 может только осуществлять многопоточность процессора по времени, а Tesla может осуществлять многопоточность по времени и по пространству, то есть по восьми потоковым процессорам, а также по сегментам из четырех тактовых циклов

В микропроцессоре Tesla используется мелкомодульная аппаратная многопоточность для диспетчеризации по времени 24 вампов, которые показаны вертикально в блоках из четырех тактовых циклов. Аналогично этому, UltraSPARC T2 осуществляет диспетчеризацию по времени восьми аппаратно поддерживаемых потоков, показанных по вертикали, по одному в каждом цикле. Таким образом, так же, как оборудование T2 осуществляет переключение между потоками для поддержания занятости ядра T2, оборудование Tesla осуществляет переключение между варпами, чтобы сохранить занятость мультипроцессора Tesla. Основное различие состоит в том, что у ядра T2 имеется один процессор, способный переключать потоки с каждым тактовым циклом, в то время как минимальной единицей переключения варпов в мультипроцессоре Tesla является интервал в два тактовых цикла параллельно для восьми потоковых ядрер. Хотя Tesla предназначен для программ с большим объемом параллелизма на уровне данных, разработчики считали, что между переключениями каждые два или четыре цикла по сравнению с переключениями на каждом цикле различий в производительности немного и что при ограничении частоты переключений оборудование было бы намного проще.

У среды программирования CUDA также есть своя собственная терминология. Программа CUDA является унифицированной C/C++-программой, предназначенной для гетерогенного центрального процессора и системы GPU. Она выполняется на центральном процессоре и осуществляет диспетчеризацию параллельной работы для GPU. Эта работа состоит из передачи данных от оперативной памяти и из *диспетчеризации потоков*. Поток является частью программы для GPU. Программисты указывают количество потоков в поточном блоке и количество поточных блоков, которое требуется запустить на выполнение на GPU. Программисты заботятся о поточных блоках, поскольку все потоки в блоке подвергаются диспетчеризации для запуска на одном и том же мультипроцессоре, поэтому они совместно используют одну и ту же локальную память. Благодаря этому они могут обмениваться данными через загрузки и сохранения, а не через сообщения. Компилятор CUDA распределяет регистры каждому потоку при том ограничении, что количество регистров на один поток, умноженное на количество потоков на поточный блок, не должно превышать число, равное 8192 регистрам на один мультипроцессор.

Потоковый блок может иметь до 512 потоков. Каждая группа из 32 потоков в потоковом блоке запакована в варпы. Большие потоковые блоки более эффективны, чем небольшие, и они могут уменьшаться до одного блока. Как уже ранее упоминалось, потоковые блоки и варпы, состоящие менее чем из 32 потоков, работают менее эффективно, чем заполненные.

Аппаратная диспетчеризация старается, по возможности, спланировать выполнение нескольких потоковых блоков на мультипроцессоре. Если это удается, диспетчер также динамически делит 16 Кбайт локального запоминающего устройства между различными потоковыми блоками.

Перспективы графических процессоров

Графические процессоры, подобные NVIDIA Tesla, не вполне соответствуют предыдущим классификациям компьютеров, например в такой, как в табл. 7.2. Несомненно, GeForce 8800 GTX с 16 мультипроцессорами Tesla относится к MIMD. Вопрос в том, как классифицировать каждый из мультипроцессоров Tesla и восемь потоковых процессоров, составляющих мультипроцессор Tesla.

Ранее утверждалось, что SIMD больше всего подходит для циклов `for` и меньше всего подходит для операторов `case` и `switch`. Tesla стремится к достижению высокой эффективности при параллелизме на уровне данных, облегчая программистам работу с независимым параллелизмом на уровне потоков. Tesla позволяет программисту считать мультипроцессор многопоточным MIMD-устройством, состоящим из восьми потоковых процессоров, но оборудование старается собрать вместе восемь потоковых процессоров для работы в качестве SIMT-устройства, когда несколько потоков одного и того же варпа могут выполняться вместе. Когда потоки работают независимо друг от друга и следуют по независимым путям выполнения, они выполняются медленнее, чем в режиме SIMT, поскольку все 32 потока в варпе совместно используют один блок извлечения инструкций. Если бы все 32 потока варпа выполняли независимые инструкции, каждый поток работал бы в 1/16 пиковой производительности полного варпа из 32 потоков, выполняемых на восьми потоковых процессорах в течение четырех тактовых циклов.

Таким образом, у каждого независимого потока есть свой собственный эффективный персональный компьютер, поэтому программисты могут представлять мультипроцессор Tesla в виде MIMD-устройства, но, чтобы добиться желаемой производительности, они должны позаботиться о написании инструкций потока управления, позволяющих SIMT-оборудованию выполнять CUDA-программы в SIMD-режиме.

Таблица 7.3. Аппаратная классификация процессорных архитектур и примеров, основанная на статическом способе в противоположность динамическому и на параллелизме на уровне инструкций в противоположность параллелизму на уровне данных

	Статический способ: обнаруживается во время компиляции	**Динамический способ: обнаруживается в процессе выполнения**
Параллелизм на уровне инструкций	VLIW	Суперскаляр
Параллелизм на уровне данных	SIMD или вектор	Мультипроцессор Tesla

В отличие от векторной архитектуры, которая для распознавания параллелизма на уровне данных и генерации векторных инструкций полагается на векторизирующий компилятор, аппаратная реализация архитектуры Tesla обнаруживает параллелизм на уровне данных среди потоков в процессе выполнения. Таким образом, графические процессоры Tesla не нуждаются в векторизирующих компиляторах, и они облегчают программисту работу с частями программы, не имеющими парал-

лелизма на уровне данных. Чтобы разобраться с перспективами этого уникального подхода, в табл. 7.3 приведена классификация, рассматривающая параллелизм на уровне инструкций в противоположность параллелизму на уровне данных, и показывается, где обнаруживается параллелизм — во время компиляции или в процессе выполнения. Эта классификация свидетельствует о том, что графический процессор Tesla является новым словом в компьютерной архитектуре.

Самопроверка

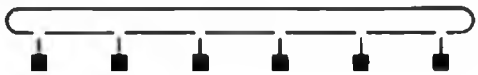
Определите истинность или ложность следующего утверждения: графические процессоры рассчитаны на использование графических DRAM-модулей в целях сокращения латентности памяти и увеличения таким образом производительности графических приложений.

7.8. Введение в топологию мультипроцессорных сетей

Для соединения ядер в единую систему многоядерным микросхемам нужна сеть на кристалле. В этом разделе рассматриваются все за и против различных мультипроцессорных сетей.

Когда сеть проецируется на кристалл, в сетевые издержки включается количество коммутаторов, количество связей на коммутатор для подключения к сети, ширина (количество разрядов) связи и длина связей. Например, некоторые ядра могут примыкать друг к другу, а другие могут находиться на другой стороне кристалла. Сетевая производительность также обладает множеством аспектов. В нее включается латентность незагруженной сети при передаче и получении сообщения, пропускная способность в понятиях максимального количества сообщений, которые могут быть переданы за определенный период времени, задержки, вызванные конкуренцией за часть сети, и изменение производительности в зависимости от структуры связи. Еще одной обязанностью сети может быть отказоустойчивость, поскольку системы могут быть востребованы для работы при наличии вышедших из строя компонентов. И наконец, в эру микросхем, ограниченных по потребляемой мощности, энергоэффективность различных устройств может превалировать над всеми остальными вопросами, требующими решения.

Сети обычно изображаются в виде графов, где каждая дуга графа представляет собой связь с коммуникационной сетью. Узел процессор–память, показывается в виде черного квадрата, а коммутатор показывается в виде выделенного оттенком кружка. В этом разделе все связи являются *двунаправленными*, то есть информация может идти в обоих направлениях. Все сети состоят из *коммутаторов*, чьи связи следуют к узлам процессор—память и к другим коммутаторам. Первым усовершенствованием относительно шины является сеть, соединяющая вместе последовательность узлов:



Эта топология называется *кольцом*. Поскольку некоторые узлы подключены не напрямую, некоторые сообщения должны будут перескакивать через промежуточные узлы, пока они не доберутся до пункта назначения.

В отличие от шины, кольцо способно на множество одновременных передач. Поскольку существует множество топологий, из которых приходится выбирать, необходимы показатели производительности, отличающие все эти конструкции друг от друга. Наиболее популярны два показателя. Первым является *общая* **пропускная способность сети**, представленная пропускной способностью каждой связи, умноженной на количество связей. Этот показатель представляет наилучший вариант. Для показанной выше кольцевой сети с P процессорами общая пропускная способность сети будет равна P, умноженному на пропускную способность одной связи; общая пропускная способность шины — это всего лишь пропускная способность этой шины, или удвоенная пропускная способность этой связи.

Чтобы сбалансировать этот наилучший вариант, мы включили еще один показатель, близкий к наихудшему варианту: **пропускная способность сечения**. Он вычисляется путем деления машины на две части, в каждой из которых находится половина узлов. Затем суммируются пропускные способности связей, пересекающих эту воображаемую линию деления. Пропускная способность при делении кольца пополам представляет собой удвоенную пропускную способность и однократную полосу пропускания шины. Если отдельная связь обладает такой же скоростью, что и шина, то кольцо в наихудшем случае лишь вдвое быстрее шины, но в наилучшем случае оно в P раз быстрее.

Поскольку некоторые сетевые топологии симметрией не обладают, возникает вопрос, где проводить воображаемую линию при делении пополам. Эта оценка по наихудшему варианту, поэтому ответ состоит в выборе такого деления, в результате которого получается самая пессимистическая сетевая производительность. Иначе говоря, нужно вычислить все возможные показатели пропускной способности при приблизительных делениях пополам и выбрать из них наименьший. Этот пессимистический взгляд нужен, потому что возможности параллельных программ зачастую ограничены самой слабой связью в коммуникационной цепи.

Другим экстремальным состоянием кольца является **полностью подключенная сеть**, где каждый процессор имеет двунаправленную связь с каждым другим процессором. Для полностью подключенных сетей общая пропускная способность сети равна $P \times (P-1)/2$, а пропускная способность при делении пополам равна $(P/2)^2$.

Существенное повышение производительности полностью подключенных сетей компенсируется чрезвычайно большим увеличением стоимости. Это стимулирует инженеров на изобретение новых топологий, в которых делается выбор между стоимостью колец и производительностью полностью подключенных сетей.

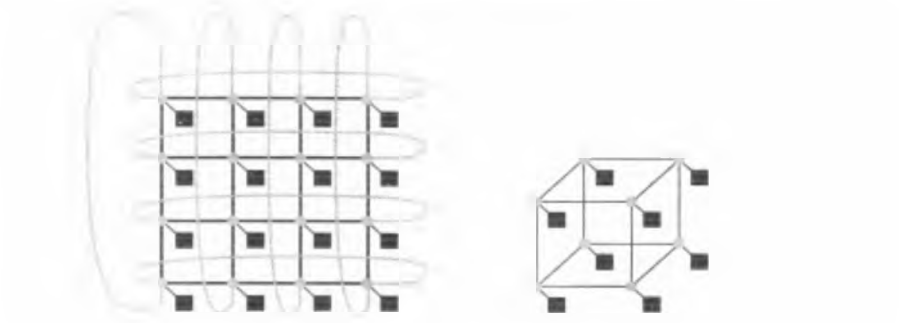
Пропускная способность сети
Неформально это пиковая скорость передачи данных по сети; может ссылаться на скорость отдельной связи или на общую скорость передачи данных всех связей сети.

Пропускная способность сечения
Показатель пропускной способности между двумя равными частями мультипроцессора. Это оценка проводится для наихудшего случая разбиения микропроцессора.

Полностью подключенная сеть
Сеть, соединяющая узлы «процессор—память» путем предоставления выделенной линии связи между каждой парой узлов.

Оценка успешности того или иного решения зависит в основном от природы связи при рабочей нагрузке параллельных программ, выполняемых на машине.

Количество различных топологий, рассмотренных в публикациях, подсчитать довольно трудно, но в коммерческих параллельных процессорах применение нашла лишь малая их часть. На рис. 6.6 показаны две популярные топологии. В реальных машинах для повышения производительности и надежности к этим простым топологиям часто добавляются дополнительные связи.



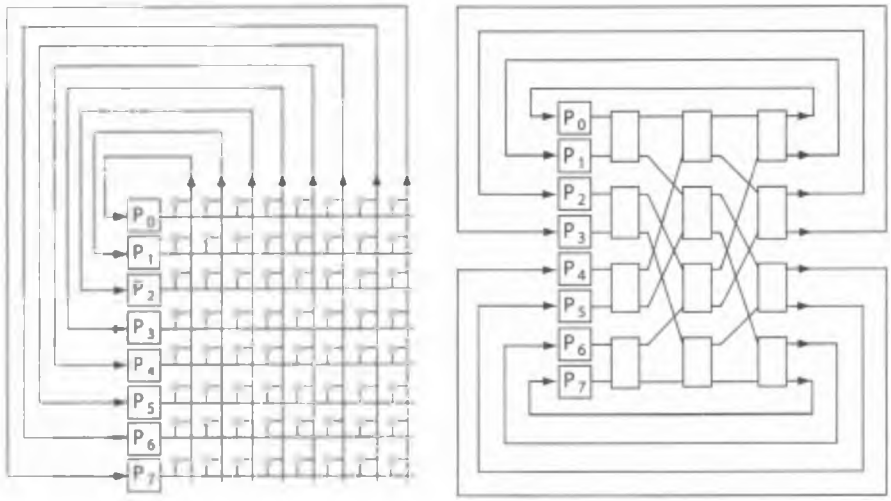
а) двумерная решетка, или сетка из 16 узлов

б) дерево n-куба из 8 узлов ($8 = 2^3$, $n = 3$)

Рис. 7.6. Сетевая топология, встречающаяся в промышленных параллельных процессорах. Серые кружочки представляют коммутаторы, а черные квадраты обозначают узлы «процессор—память». Даже при том, что коммутаторы имеют множество связей, как правило, только одна из них ведет к процессору. Булева топология n-куб является n-мерным соединением с 2^n узлами, требующими n связей на коммутатор (плюс одну для процессора), соединяя, таким образом, n ближайших друг к другу узлов. Зачастую эти основные топологии оснащаются дополнительными дугами для повышения производительности и надежности

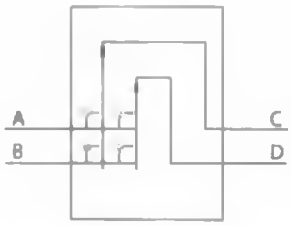
Альтернативой помещению процессора в каждый узел сети является оставление коммутатора только в некоторых из этих узлов. Коммутаторы меньше, чем узлы «процессор—память—коммутатор», и поэтому могут быть упакованы более плотно, уменьшая дистанцию и повышая производительность. Такие сети часто называют **многоступенчатыми сетями**, чтобы подчеркнуть существование ступеней, по которым могут путешествовать сообщения. Типы многоступенчатых сетей столь же многочисленны, что и типы одноступенчатых сетей; на рис. 7.7 показаны две популярные многоступенчатые структуры. Сеть с **полным подключением**, или сеть с **поперечной коммутацией**, позволяет любому узлу связываться с любым другим узлом за один проход по сети. Сеть *Omega* использует меньший объем оборудования, чем сеть с поперечной коммутацией ($2n \log_2 n$ против n^2 коммутаторов), но, в зависимости от применяемой схемы связи, между сообщениями может складываться состязательная ситуация. Например, сеть Omega на рис. 7.7 не может отправить сообщение от P_0 к P_6 одновременно с отправкой сообщения от P_1 к P_7.

Сеть с поперечной коммутацией
Сеть, позволяющая любому узлу связываться с любым другим узлом за один проход по сети.



а) сеть с поперечной коммутацией

б) сеть Omega



в) коммутатор сети Omega

Рис. 7.7. Популярные топологии многоступенчатых сетей. Коммутаторы на этих рисунках проще, чем на предыдущих рисунках, поскольку связи являются однонаправленными; данные входят в основании и выходят из правой связи. Коммутатор, показанный на рисунке *в*, может передать A в C и B в D или B в C и A в D. В сети с поперечной коммутацией используется n^2 коммутаторов, где n количество процессоров, а в сети Omega используется $2n \log_2 n$ больших блоков коммутации, каждый из которых логически состоит из четырех более мелких коммутаторов. В данном случае в сети с поперечной коммутацией используется 64 коммутатора против 12 коммутационных блоков, используемых в сети Omega. Но сеть с поперечной коммутацией может поддерживать любую комбинацию сообщений между процессорами, тогда как сеть Omega на это не способна

Реализация сетевых топологий

В упрощенном анализе всех сетей, представленном в данном разделе, проигнорированы важные практические соображения, касающиеся конструкции сетей. Расстояние каждой связи влияет на стоимость обмена данными на высокой тактовой частоте, в общем, чем длиннее дистанция, тем дороже запустить сеть на высокой тактовой частоте. Более короткие дистанции кроме этого облегчают назначение для связи большего количества проводников, поскольку мощность, затрачиваемая на управление множеством проводников на кристалле, меньше, если проводники

короче. Более короткие проводники также обходятся дешевле длинных. Еще одним практическим ограничением является то, что трехмерное изображение должно переноситься на кристаллы, являющиеся, в конечном счете, двумерной передающей средой. И конечной озабоченностью является потребляемая мощность. Именно она, к примеру, может заставить кристаллы полагаться на простые топологии в виде решетки. Можно сказать, что те топологии, которые элегантно выглядят на чертеже, могут быть непрактичными при их реализации.

7.9. Контрольные задачи для мультипроцессоров

В главе 1 уже было показано, что системы, проводящие тестирование с помощью контрольных задач, всегда популярны, потому что они представляют весьма наглядный способ определения, какая из систем лучше. Результаты оказывают влияние не только на продажи промышленных систем, но также и на репутацию их разработчиков. Следовательно, участники хотят выиграть соревнование, но они также хотят убедиться, что если побеждает кто-то другой, то он побеждает по праву, потому что у него действительно лучшая система. Это желание выражается в правилах, гарантирующих, что результаты выполнения контрольных задач — не простые технические трюки, разработанные для этих задач, а прогрессивное средство, способствующее улучшению производительности реальных приложений.

Чтобы избежать возможных трюков, обычно устанавливается правило, не позволяющее изменять контрольную задачу. Исходный код и наборы данных не изменяются, чему есть единственное надлежащее обоснование. Любое отклонение от этих правил делает результаты недействительными.

Этим традиционным правилам следуют многие контрольные задачи для мультипроцессоров. Самое распространенное исключение состоит в возможности увеличивать объем задач, чтобы можно было запускать контрольную задачу на системах с сильно отличающимся количеством процессоров. Таким образом, многие контрольные задачи допускают небольшое масштабирование, отступая от правил строгого масштабирования, но при сравнении результатов для программ, выполняющих разные по объему задачи, нужно проявлять особую осмотрительность.

В табл. 7.4 дана краткая сводка ряда параллельных контрольных задач, описание которых приводится ниже:

- *Linpack* — это коллекция подпрограмм линейной алгебры и подпрограмм для составления Гауссовых исключений, известная как Linpack benchmark. Подпрограмма DAXPY в упражнении раздела 7.6 представляет небольшой фрагмент исходного кода Linpack benchmark. В данной коллекции разрешено нестрогое масштабирование, позволяющее пользователю выбирать задачу любого размера. Кроме того, коллекция позволяет пользователю переписывать Linpack в любой форме и на любом языке, пока он будет вычислять приемлемые результаты. Дважды в год на сайте www.top500.org публикуется список из 500 самых быстрых компьютеров, показавших наивысшую производительность при решении

контрольных задач Linpack. Первый в этом списке, по мнению прессы, является самым быстрым компьютером в мире.

- *SPECrate* является средством оценки пропускной способности, основанным на таких контрольных задачах SPEC CPU, как SPEC CPU 2006 (см. главу 1). Вместо выдачи отчета о производительности отдельных программ SPECrate одновременно выполняет множество копий программы. Таким образом, этот пакет дает оценку параллелизма на уровне заданий, поскольку обмен данными между заданиями отсутствует. Можно запустить любое желаемое количество копий программ, что опять-таки является формой нестрогого масштабирования.
- *SPLASH* и *SPLASH2* (Stanford Parallel Applications for Shared Memory — Стэнфордские параллельные приложения для совместно используемой памяти) были результатом усилий исследователей, предпринятых в 1990-х годах и направленных на подготовку комплекта контрольных задач для параллельных вычислений, сходных по назначению с комплектом контрольных задач SPEC CPU. Он включает как базовые компоненты (kernels), так и приложения, многие из которых получены от сообщества высокопроизводительных вычислений. Этот комплект контрольных задач требует строгого масштабирования, хотя и поставляется с двумя наборами данных.
- *NAS (NASA Advanced Supercomputing) parallel benchmarks* представлял собой еще одну попытку тестирования производительности мультипроцессоров, предпринятую в 1990-х годах. Контрольные задачи, взятые из вычислительной гидрогазодинамики, состоят из пяти базовых компонентов. Они позволяют использовать нестрогое масштабирование за счет определения нескольких наборов данных. Как и Linpack, эти контрольные задачи могут быть переписаны, но правила требуют, чтобы в качестве языка программирования использовался только C или Фортран.
- Один из последних пакетов контрольных задач *PARSEC* (Princeton Application Repository for Shared Memory Computers — Принстонский репозиторий приложений для компьютеров с общей памятью) состоит из многопоточных программ, использующих **Pthreads** (POSIX threads — потоки POSIX) и **OpenMP** (Open MultiProcessing — открытая мультипроцессорная обработка).

Недостатком подобных традиционных ограничений, накладываемых на контрольные задачи, является то, что новшества в основном ограничены архитектурой и компилятором. Зачастую невозможно воспользоваться более совершенными структурами данных, алгоритмами, языками программирования и т. д., поскольку все это даст дезориентирующий результат. Система может победить благодаря, скажем, алгоритму, а не благодаря оборудованию или компилятору.

Pthread
UNIX API для создания потоков и манипулирования ими. Поставляется вместе с библиотекой.

OpenMP
API для мультипроцессорных вычислений с использованием общей памяти в C, C++ или Фортран, выполняемый на платформах UNIX и Microsoft. Включает в себя директивы компилятора, библиотеку и директивы времени выполнения.

Хотя эти нормативы были понятны в 1990-е годы и в первой половине следующего десятилетия, когда основные показатели вычислительной техники сохраняли относительную

стабильность, с началом компьютерной революции они перестали всех устраивать. Чтобы революционные темпы сохранялись, понадобилась поддержка инноваций на всех уровнях.

Таблица 7.4. Примеры параллельных контрольных задач

Контрольная задача	Масштабирование	Перепрограммирование	Описание
Linpack	Нестрогое	Да	Плотная матричная линейная алгебра (Dongarra, 1979)
SPECrate	Нестрогое	Нет	Независимый параллелизм заданий (Henning, 2007)
Параллельные приложения для совместно используемой памяти, разработанные в Стэндфордском университете, SPLASH 2 — Stanford Parallel Applications for Shared Memory SPLASH 2 (Woo et al., 1995)	Строгое (хотя допускается два объема задач)	Нет	Комплекс одномерных быстрых преобразований Фурье Блокирующаяся LU-декомпозиция Блокирующаяся разбросанная факторизация Чолески Целочисленная поразрядная сортировка Моделирование метода Барнеса-Хата Моделирование быстро адаптируемого мультиполя Моделирование океана Иерархическая излучательность Трассировка лучей Рендерер объема Моделирование воды с пространственной структурой данных Моделирование воды без пространственной структуры данных
Контрольные задачи для параллельных вычислений NAS — NAS Parallel Benchmarks (Bailey et al., 1991)	Нестрогое	Да (только C или Фортран)	EP: Чрезвычайная параллельность MG: Упрощенная мультирешетка CG: Неструктурированная решетка для метода сопряженных градиентов FT: Трехмерное частичное решение дифференциального уравнения с использованием быстрых преобразований Фурье IS: Объемная сортировка целых чисел
Набор контрольных задач PARSEC — PARSEC Benchmark Suite (Bienia et al., 2008)	Нестрогое	Нет	Blackscholes — Модель ценообразования опционов Блэка-Шоулза Bodytrack — Отслеживание телодвижений человека Canneal — Моделирование выравнивания для оптимизации маршрутизации на основе осведомленности о состоянии кэш-памяти Dedup — Перспективное сжатие с исключением дублирования данных Facesim — моделирование мимики лица Ferret — Сервер поиска похожего содержимого Fluidanimate — Гидрогазодинамика для анимации с использованием метода SPH Freqmine — Частая выемка набора данных Streamcluster — Онлайн-кластеризация входного потока Swaptions — Оценка портфеля своп-опционов Vips — Обработка изображений x264 — Видеокодирование H.264

продолжение ⇩

Таблица 7.4 (*продолжение*)

Контрольная задача	Масштабирование	Перепрограммирование	Описание
Шаблоны конструирования, разработанные в Университете Беркли — Berkeley Design Patterns (Asanovic et al., 2006)	Строгое или нестрогое	Да	Конечный автомат (Finite-State Machine) Комбинаторная логика (Combinational Logic) Обход графов (Graph Traversal) Структурированная решетка (Structured Grid) Плотная матрица (Dense Matrix) Разреженная матрица (Sparse Matrix) Спектральные методы (быстрое преобразование Фурье) — Spectral Methods (FFT) Динамическое программирование (Dynamic Programming) Задача N тел (N-Body) Функции Map и Reduce (MapReduce) Обратное отслеживание / ветвление и граница (Backtrack/Branch and Bound) Выведение графической модели (Graphical Model Inference) Неструктурированная решетка (Unstructured Grid)

Один из последних подходов был поддержан исследованиями Калифорнийского университета в Беркли. Были определены 13 шаблонов конструирования, заявленные частью будущих приложений. Эти шаблоны конструирования реализованы с помощью структур, или ядер. В качестве примеров можно привести разреженные матрицы, структурированные решетки, конечные автоматы, функции map и reduce и обход графа. Позиционируя определения на высоком уровне, специалисты надеются стимулировать новшества на любом уровне системы. Таким образом, система, способная быстрее всех выдавать решение разреженной матрицы, наряду с самыми новыми архитектурами и компиляторами должна поддерживать использование любой структуры данных, алгоритма и языка программирования. Примеры таких контрольных задач будут показаны в разделе 7.11.

Самопроверка

Определите истинность или ложность следующего утверждения: основным недостатком традиционных подходов к контрольным задачам для параллельных компьютеров является то, что правила, обеспечивающие справедливость, наряду с этим подавляют новшества.

7.10. Roofline: простая модель производительности

Этот раздел основан на статье Уильямса и Паттерсона (Williams, Patterson, 2008). В недавнем прошлом обычный здравый смысл в компьютерной архитектуре привел к сходным конструкциям микропроцессоров. Почти каждый настольный и серверный компьютер использует устройства кэш-памяти, конвейеризацию, выдачу

суперскалярных инструкций, прогнозирование условных переходов и выполнение инструкций с изменением последовательности. Наборы инструкций варьируются, но все микропроцессоры принадлежат к одной и той же школе конструирования.

Переход к мультиядерности означал, что микропроцессоры, возможно, становятся более разнообразными, поскольку никакой здравый смысл не мог подсказать, какая архитектура сможет упростить написание правильно обрабатываемых в параллельном режиме программ, работающих достаточно эффективно и масштабирующихся по мере роста со временем количества ядер. Кроме того, по мере роста количества ядер на одном кристалле отдельный производитель, скорее всего, будет в одно и то же время предлагать различное количество ядер на кристалл по разным ценам.

Учитывая растущее разнообразие, было бы весьма полезно иметь простую модель, предлагающую способ проникновения в суть производительности различных конструкций. Она не должна быть идеальной, достаточно, чтобы с ее помощью можно было постичь суть конструкции.

Аналогией может послужить модель трех «С», рассмотренная в главе 5. Эта модель неидеальна, поскольку в ней игнорируются потенциально важные факторы, такие как размер блока, политика распределения блоков и политика замены блоков. Кроме того, у нее есть свои особенности. Например, промах при обращении к кэш-памяти в одной конструкции может быть отнесен на счет ее объема, а в другой — на счет конфликта при кэш-памяти того же объема. Тем не менее модель трех «С» была популярной в течение 20 лет, поскольку она предлагала проникновение в суть поведения программ, помогая как разработчикам архитектур, так и программистам, повышая их творческий потенциал, основанный на понимании сути происходящего благодаря этой модели.

Чтобы найти нужную нам модель, давайте начнем с 13 шаблонов конструирования, определенных в Университете Беркли и показанных в табл. 7.4. Идея шаблонов конструирования состоит в том, что производительность заданных приложений на самом деле является взвешенной суммой нескольких базовых компонентов, реализующих эти шаблоны конструирования. Здесь будет дана оценка отдельным базовым компонентам, но следует иметь в виду, что реальные приложения являются комбинацией множества базовых компонентов.

Хотя есть версии с различными типами данных, в ряде реализаций популярны числа с плавающей точкой. Следовательно, пиковая производительность при работе с числами с плавающей точкой выражается пределом скорости выполнения таких базовых компонентов на отдельно взятом компьютере. Для многоядерных кристаллов пиковая производительность при работе с числами с плавающей точкой является совокупностью пиковых производительностей всех ядер. Если в системе имеется несколько микропроцессоров, пиковая производительность, приходящаяся на один кристалл, должна быть умножена на общее количество кристаллов.

Требования к системе памяти могут быть оценены путем деления этой пиковой производительности при работе с числами с плавающей точкой на среднее количество операций с плавающей точкой, приходящееся на одно обращение к байту:

$$\frac{\text{Количество операций с плавающей точкой в секунду}}{\text{Количество операций с плавающей точкой на байт}} = \text{Байт в секунду.}$$

Соотношение количества операций с плавающей точкой на байт обращений к памяти называется **арифметической интенсивностью**. Этот показатель может быть вычислен путем деления общего количества операций с плавающей точкой, имеющегося в программе, на общее количество байтов данных, переданных оперативной памяти в процессе выполнения программы. На рис. 6.8 показана арифметическая интенсивность ряда шаблонов конструирования, разработанных в Беркли и показанных в табл. 7.4.

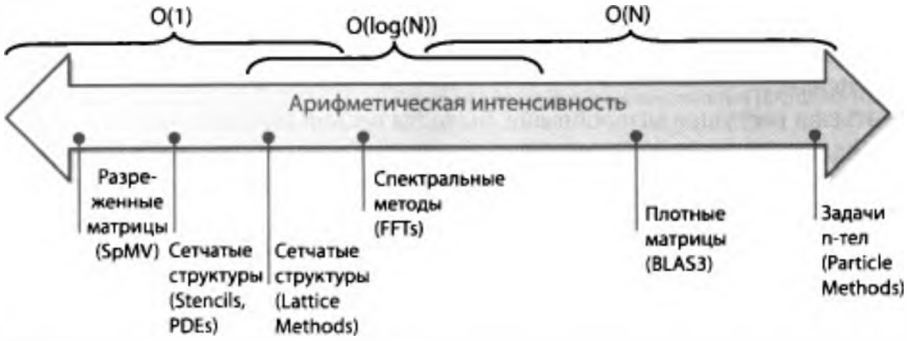


Рис. 7.8. Арифметическая интенсивность, определенная как количество операций с плавающей точкой в выполняемой программе, деленное на количество байтов оперативной памяти, к которым было обращение (Williams, Patterson, 2008). Некоторые базовые компоненты имеют арифметическую интенсивность, которая масштабируется параллельно объему задачи, например Плотная матрица, но существует множество базовых компонентов, где арифметическая интенсивность не зависит от объема задачи. Для базовых компонентов из этой последней категории нестрогое масштабирование может привести к различным результатам, из-за того, что к системе памяти предъявляется меньше требований

Модель Roofline

Предлагаемая простая модель связывает производительность работы с числами с плавающей точкой, арифметическую интенсивность и производительность памяти (Williams, Patterson, 2008). Пиковая производительность работы с числами с плавающей точкой может быть определена с использованием вышеупомянутых спецификаций оборудования. Рассматриваемый здесь рабочий набор базовых компонентов не помещается в кэш-память, имеющуюся на кристалле, поэтому пиковая производительность памяти может быть определена в отношении системы памяти с отставанием от устройств кэш-памяти. Один из способов определения пиковой производительности памяти — использование контрольной задачи Stream. (См. уточнение в разделе 5.2.) На рис. 7.9 показана модель, выполненная один раз на компьютере, но не для каждого базового компонента. По вертикальной оси Y показана достижимая производительность при работе с числами с плавающей точкой от 0,5 до 64,0 Гфлоп/с. По горизонтальной оси X показана арифметическая интенсивность, варьирующаяся от 1/8 флоп/обращений к байту DRAM до

Арифметическая интенсивность
Соотношение общего количества операций с плавающей точкой, имеющегося в программе, к общему количеству байтов данных в оперативной памяти, к которому обращалась программа в процессе своего выполнения

16 флоп/обращений к байту DRAM. Учтите, что график выполнен с использованием логарифмической шкалы в обеих осях.

Для заданного базового компонента можно найти точку на оси X по его арифметической интенсивности. Если через эту точку провести вертикальную линию, то производительность базового компонента на этом компьютере должна лежать где-то вдоль этой линии. Можно начертить горизонтальную линию, показывающую пиковую производительность этого компьютера при работе с числами с плавающей точкой. Вполне очевидно, что настоящая производительность при работе с числами с плавающей точкой не может быть выше этой горизонтальной линии в силу ограничений, накладываемых оборудованием.

Рис. 7.9. Модель Roofline Williams, Patterson, 2008). У данного примера пиковая производительность работы с числами с плавающей точкой равна 16 Гфлоп/с, а пиковая пропускная способность памяти, полученная по результатам выполнения контрольной задачи Stream, равна 16 Гбайт/с. (Поскольку при ее выполнении делается четыре измерения, эта линия является их средним показателем.) Выделенная другим оттенком точечная вертикальная линия слева представляет Базовый компонент 1 (Kernel 1), имеющий арифметическую интенсивность 0,5 флоп/байт. Его производительность ограничена пропускной способностью (bandwidth — BW) памяти и на данном процессоре Opteron X2 не может превышать 8 Гфлоп/с. Вертикальная точечная линия справа представляет Базовый компонент 2 (Kernel 2), с арифметической интенсивностью 4 флоп/байт. Производительность ограничена только вычислительными возможностями, равными 16 Гфлоп/с. (Эти данные основаны на использовании AMD Opteron X2, Модификация F, использующем по два ядра, работающие на частоте 2 ГГц в системе с двумя гнездами под кристалл.)

Как изобразить графически пиковую производительность памяти? Поскольку по оси X откладываются значения в флоп/байт, а по оси Y откладываются значения в флоп/с, то на этом рисунке байт/с — это просто диагональная линия под углом 45 градусов. Следовательно, можно нарисовать еще одну линию, показывающую максимальную производительность работы с числами с плавающей точкой, поддерживаемую системой памяти данного компьютера для заданной арифметической

интенсивности. Ограничения можно выразить формулой для построения линии на графике рис. 7.9:

Достижимый показатель Гфлоп/с = Min (Пиковая пропускная способность памяти × Арифметическая интенсивность, пиковая производительность работы с числами с плавающей точкой).

Горизонтальные и диагональные линии дали этой простой модели имя и показывают ее значение. «Линия крыши» («roofline») устанавливает верхнюю границу производительности базового компонента в зависимости от его арифметической интенсивности. Если арифметическая интенсивность представляется в виде столба, упирающегося в крышу, то этот столб упирается либо в ее плоскую часть — ограничение производительности вычислительными возможностями, либо упирается в наклонную часть крыши — производительность ограничена пропускной способностью памяти. На рис. 6.9 базовый компонент 2 является примером первого ограничения, а базовый компонент 1 — примером второго ограничения. При заданной для компьютера линии крыши ее можно использовать не один раз, поскольку она не зависит от базового компонента.

Учтите, что «ребро», где встречаются диагональ и горизонталь крыш, предлагает интересный взгляд на возможности компьютера. Если пойти от него вправо, то максимальной производительности этого компьютера могут достичь только базовые компоненты с высокой арифметической интенсивностью. Если пойти влево, то максимальной производительности может достичь практически любой базовый компонент. Скоро мы увидим примеры того и другого случая.

Сравнение двух поколений процессора Opteron

Процессор AMD Opteron X4 (Barcelona), имеющий четыре ядра, является последователем процессора Opteron X2, у которого два ядра. Чтобы упростить конструкцию материнской платы, они используют одно и то же гнездо. Следовательно, у них одинаковые DRAM-каналы и, соответственно, одинаковая пиковая пропускная способность. Кроме удвоенного количества ядер, у Opteron X4 также имеется удвоенная пиковая производительность работы с числами с плавающей точкой на одно ядро: ядра Opteron X4 могут запускать две SSE2-инструкции с плавающей точкой в одном тактовом цикле, а ядра Opteron X2 запускают не больше одной. Поскольку обе сравниваемые системы имеют похожие тактовые частоты — 2,2 ГГц у Opteron X2 и 2,3 ГГц у Opteron X4 — процессор Opteron X4 обладает более чем в четыре раза лучшей пиковой производительностью при работе с числами с плавающей точкой, чем процессор Opteron X2 при одинаковых показателях пропускной способности DRAM. У Opteron X4 также имеется 2 Мбайт кэш-памяти уровня L3, которой нет у Opteron X2.

На рис. 6.10 представлены в сравнении roofline-модели обеих систем. Нетрудно предположить, что ребро переместилось с 1 в Opteron X2 на 5 в Opteron X4. Следовательно, чтобы увидеть производительность, полученную в следующем поколении, базовые компоненты должны иметь арифметическую интенсивность

выше, чем 1, или их рабочие наборы должны помещаться в устройства кэш-памяти Opteron X4.

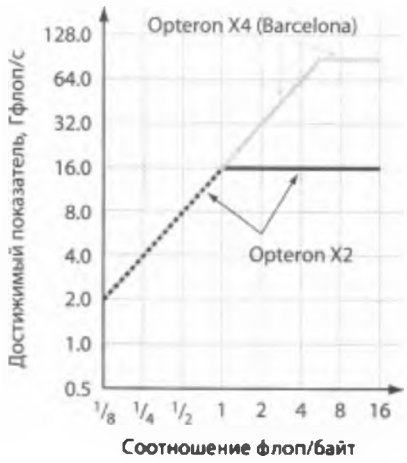


Рис. 7.10. Roofline-модели двух поколений процессоров Opteron. Roofline-модель Opteron X2, аналогичная показанной на рис. 7.9, нарисована более темными линиями, а roofline-модель Opteron X4 нарисована более светлыми линиями. Более высокое ребро Opteron X4 означает, что те базовые компоненты, которые в Opteron X2 были ограничены вычислительными возможностями, в Opteron X4 могут быть ограничены производительностью памяти

Roofline-модель определяет верхнюю границу производительности. Предположим, что ваша программа имеет показатели значительно ниже границы. Какую оптимизацию нужно провести и в каком порядке?

Для сокращения количества узких мест в вычислениях почти любому базовому компоненту могут помочь следующие две оптимизации:

1. *Смешивание операций с плавающей точкой.* Достижение пиковой производительности работы компьютера с числами с плавающей точкой обычно требует одинакового количества практически одновременно производимых сложений и умножений. Этот баланс необходим либо потому, что компьютер поддерживает совмещенную операцию умножения-сложения (см. второе уточнение в подразделе «Точная арифметика» раздела 3.5 главы 3), либо потому, что блок арифметики с плавающей точкой имеет одинаковое количество сумматоров чисел с плавающей точкой и мультипликаторов чисел с плавающей точкой. Более высокая производительность также требует, чтобы существенная доля смеси инструкций была представлена операциями с плавающей точкой, а не целочисленными операциями.
2. *Повышение параллелизма на уровне инструкций (ILP) и применение SIMD.* Для суперскалярных архитектур наивысшая производительность достигается при извлечении, выполнении и завершении выполнения от трех до четырех инструкций за каждый тактовый цикл (см. главу 4). Цель состоит в улучшении кода компилятора для повышения параллелизма на уровне инструкций. Один из способов достижения этого заключается в развертывании циклов. Для архи-

тектур x86 отдельная SIMD-инструкция может работать с парами операндов двойной точности, поэтому они должны использоваться везде, где это возможно.

Для сокращения количества узких мест в работе памяти могут помочь следующие две оптимизации:

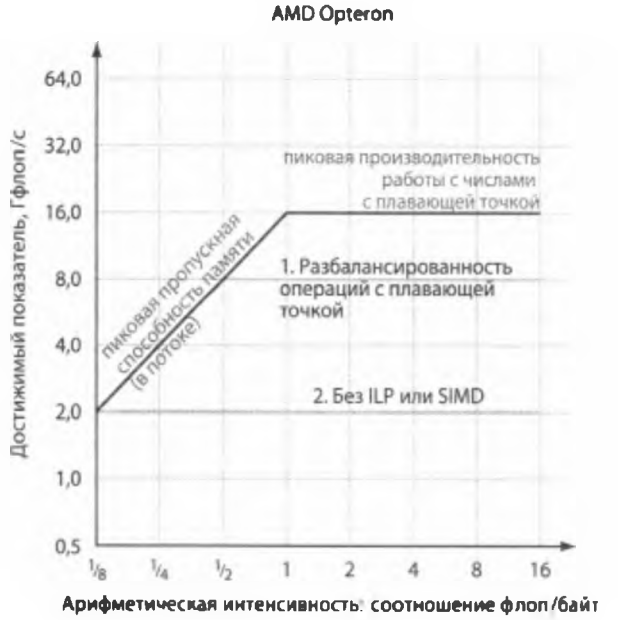
1. *Программная предвыборка*. Обычно для достижения наивысшей производительности требуется очень быстрое выполнение множества операций обращения к памяти, что проще реализовать, выполняя программную предвыборку инструкций, не ожидая, пока данные понадобятся для вычислений.
2. *Близость памяти*. Большинство микропроцессоров сегодня включают контроллер памяти, расположенный на том же кристалле, что и микропроцессор. Если система включает несколько кристаллов, это означает, что некоторые адреса ведут к DRAM-модулю, который является локальным по отношению к одному кристаллу, а остальные адреса требуют обращений через устройство соединения кристаллов, чтобы получить доступ к DRAM-модулю, который является локальным по отношению к другому кристаллу. Данная оптимизация пытается распределить данные и потоки, на которые возложена задача по их обработке, одной и той же паре память—процессор, чтобы процессору пришлось обращаться к памяти других кристаллов лишь в редких случаях.

Модель roofline может помочь принять решение, какую из этих оптимизаций предпринять и в каком порядке. Каждую из этих оптимизаций можно рассматривать как некий «потолок» ниже соответствующей линии крыши, означающий, что пробить потолок без осуществления связанной с ним оптимизации невозможно.

Линия крыши вычислительной производительности может быть найдена в руководствах, а линия крыши производительности памяти может быть определена путем запуска контрольной задачи Stream. Потолки вычислительной производительности, такие как баланс вычислений с числами с плавающей точкой, также берутся из руководства к компьютеру. Определение потолков производительности памяти требует проведения экспериментов на каждом компьютере, чтобы найти оптимум. Хорошо то, что этот процесс нужно провести для каждого компьютера только один раз, — как только кто-нибудь снимет характеристики потолка для данного компьютера, результатами для определения приоритетов оптимизации может воспользоваться кто угодно.

На рис. 6.11 к roofline-модели, показанной на рис. 7.9, добавлены потолки, для вычислительной производительности — на верхнем графике, а для пропускной способности памяти — на нижнем. Хотя более высокие потолки для обеих оптимизаций не помечены, но на данном рисунке они подразумеваются. Чтобы пробиться через самый высокий потолок, нужно пробиться через все потолки, расположенные ниже.

Толщина промежутка между потолком и следующим более высоким пределом будет наградой за попытку проведения этой оптимизации. Так, на рис. 7.11 подсказывается, что оптимизация 2, улучшающая ILP, приносит большую пользу для улучшения вычислительной производительности на данном компьютере, а оптимизация 4, улучшающая близость памяти, приносит на нем большую пользу для повышения пропускной способности памяти.



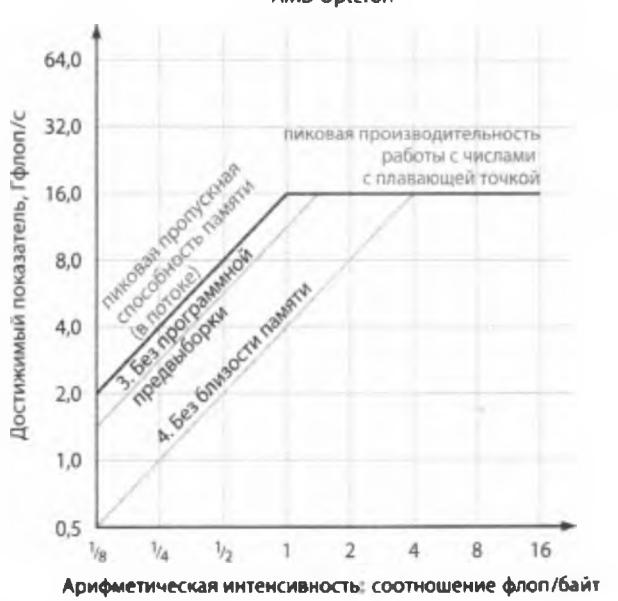


Рис. 7.11. Roofline-модель с потолками. На верхнем графике показаны вычислительные «потолки» в 8 Гфлоп/с при несбалансированности смеси операций с плавающей точкой, и в 2 Гфлоп/с при отсутствии оптимизации, повышающей параллелизм на уровне инструкций и отсутствии SIMD. На нижнем графике показаны «потолки» пропускной способности памяти в 11 Гбайт/с при отсутствии программной предвыборки, и в 4,8 Гбайт/с при отсутствии к тому же и оптимизации памяти

На рис. 6.12 потолки, показанные на рис. 7.11, объединены в одном графике. Арифметическая интенсивность базового компонента определяет область оптимизации, которая в свою очередь подсказывает, какие именно оптимизации проводить. Учтите, что для большинства показателей арифметической интенсивности вычислительные оптимизации и оптимизации пропускной способности памяти накладываются друг на друга. Три области на рис. 7.12 выделены разными цветами, чтобы показать различные стратегии оптимизации. Например, Базовый компонент 2 попадает в более светлую трапецию справа, предлагающую поработать только над вычислительными оптимизациями. Базовый компонент 1 попадает в более темный параллелограмм в средней части графика, предлагающий попробовать оба типа оптимизаций. Кроме того, он предлагает начать с оптимизаций 2 и 4. Обратите внимание на то, что вертикальные линии базового компонента 1 попадают ниже оптимизации дисбаланса операций с плавающей точкой, поэтому необходимости в оптимизации 1 может и не быть. Если базовый компонент попадает в серый треугольник в нижней левой части графика, это станет предложением на проведения оптимизаций, касающихся только памяти.

До сих пор мы предполагали, что арифметическая интенсивность имела фиксированное значение, но на самом деле так не бывает. Во-первых, есть базовые компоненты, где арифметическая интенсивность растет по мере увеличения объема задачи, например как для задачи плотной матрицы или задачи N-тел (см. рис. 7.8). Разумеется, могут быть причины того, что программисты более успешно решают свою задачу при нестрогом масштабировании, чем при строгом. Во-вторых, на количество обращений к памяти влияют устройства кэш-памяти, поэтому оптимизация, улучшающая производительность кэш-памяти, также повышает арифметическую интенсивность. Один из примеров состоит в улучшении локальности, связанной со временем, за счет развертывания циклов с последующей группировкой вместе инструкций с похожими адресами. У многих компьютеров имеются специальные инструкции для работы с кэш-памятью, которые распределяют данные по кэш-памяти, но сначала они не заполняют данный адрес данными из памяти, поскольку вскоре они будут переписаны. Обе эти оптимизации сокращают трафик памяти, перемещая тем самым столбец арифметической интенсивности вправо, скажем, на полтора порядка. Это смещение вправо может поместить базовый компонент в другую область оптимизации.

В следующем разделе roofline-модель будет использована для демонстрации разницы между четырьмя недавно выпущенными многоядерными микропроцессорами для двух базовых компонентов реального приложения. Хотя ранее рассмотренные упражнения показали, как помочь программистам повысить производительность, эта модель может также использоваться разработчиками для принятия решений по оптимизации оборудования с целью повышения производительности того базового компонента, который, по их мнению, играет важную роль.

Уточнение. Потолки, которые расположены ниже, проще пробить с помощью оптимизации. Понятно, что программист может проводить оптимизацию в любом порядке, но использование данной последовательности снижает затраты на оптимизацию, не дающую преимуществ из-за ограничений другого рода. Как и в модели трех «С», по-

скольку roofline-модель дает возможность проникновения в суть вопроса, у нее могут быть свои странности. Например, она предполагает, что программа загрузила все процессоры поровну.

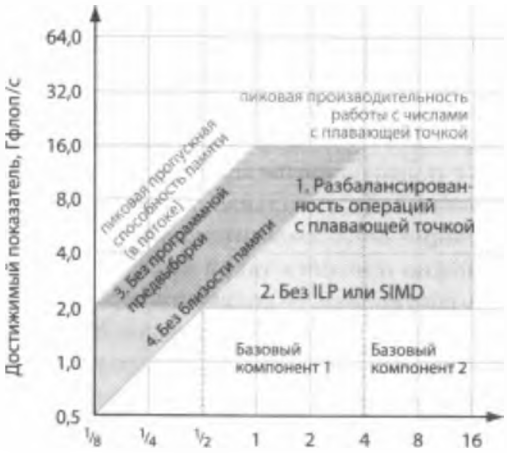


Рис. 7.12. Roofline-модель с потолками, закрашенными перекрывающимися областями и двумя базовыми компонентами из рис. 7.9. Базовые компоненты, чья арифметическая интенсивность приходится на более светлую трапецию справа, должны сосредоточиться на вычислительных оптимизациях, а базовые компоненты, чья арифметическая интенсивность попадает в серый треугольник в левой нижней части графика, должны сосредоточиться на оптимизациях пропускной способности памяти. А те базовые компоненты, чья арифметическая интенсивность попадает в темно-серый параллелограмм, расположенный в средней части, должны позаботиться о проведении обеих типов оптимизации. Поскольку столбец базового компонента 1 попадает в параллелограмм, расположенный в средней части, нужно попробовать провести оптимизации, касающиеся ILP и SIMD, близости памяти и программной предвыборки. Столбец базового компонента 2 попадает в трапецию в правой части, поэтому нужно попробовать провести оптимизации, касающиеся ILP и SIMD, а также сбалансированности операций с плавающей точкой

Уточнение. Альтернативой контрольной задачи Stream может послужить использование приблизительной оценки пропускной способности DRAM в качестве одной из линий крыши. Поскольку DRAM-модули явно устанавливают жесткую границу, реальная производительность памяти зачастую отстоит довольно далеко от этой границы, поэтому данный показатель не настолько полезен, как верхняя граница производительности.

То есть ни одна из программ не может подойти вплотную к этой границе. Недостаток использования Stream состоит в том, что очень тщательное программирование может дать более высокие результаты по сравнению с результатами Stream, поэтому линия крыши, относящаяся к памяти, может быть не настолько жестким ограничением, как линия крыши, относящаяся к вычислительной производительности. Мы выбрали Stream, потому что далеко не каждому программисту удастся добиться более высокой пропускной способности памяти, чем та, которая определяется с помощью Stream.

Уточнение. Две использовавшиеся выше оси отображали количество операций с плавающей точкой в секунду и арифметическую интенсивность доступа к оперативной памяти. Модель roofline может использоваться для других базовых компонентов и компьютеров, где производительность была функцией от других оценочных показателей.

Например, если рабочий набор помещается в компьютере в кэш-память уровня L2, то пропускной способностью, выражаемой на рисунке диагональной линией крыши, может быть показатель не оперативной памяти, а кэш-памяти уровня L2, а арифметическая интенсивность, откладываемая по оси X, могла бы основываться на количестве операций с плавающей точкой, приходящемся на одно обращение к байту кэш-памяти уровня L2. Диагональная линия производительности L2 поднимется вверх, а ребро, скорее всего, сместится влево.

В качестве второго можно привести такой пример. Если базовый компонент представлял собой задачу сортировки, то количество операций с плавающей точкой в секунду, отложенное по оси Y, может быть заменено количеством отсортированных в секунду записей, а арифметическая интенсивность может превратиться в количество записей, приходящееся на одно обращение к байту DRAM-модуля.

Модель roofline может работать даже для базового компонента, относящегося к интенсивному вводу-выводу. По оси Y может быть отложено количество операций ввода-вывода в секунду, а по оси X — среднее количество инструкций, приходящееся на одну операцию ввода-вывода, а линия крыши будет показывать пиковую пропускную способность ввода-вывода.

Уточнение. Хотя показанная модель roofline предназначена для многоядерных процессоров, она, несомненно, работает также и для одноядерных процессоров.

7.11. Реальное оборудование: выполнение контрольных задач для четырех многоядерных процессоров с использованием модели Roofline

Учитывая неопределенность относительно лучшего способа продолжения параллельной революции, не удивительно, что мы увидели столько разных конструкций, сколько имеется многоядерных кристаллов. В данном разделе будут рассмотрены четыре многоядерные системы для двух базовых компонентов из шаблонов конструирования, показанных в табл. 7.4: разреженная матрица и структурированная решетка. Информация для данного раздела позаимствована из: Williams, Oliker, et al., 2007, Williams, Carter, et al., 2008, Williams and Patterson, 2008.

Четыре многоядерные системы

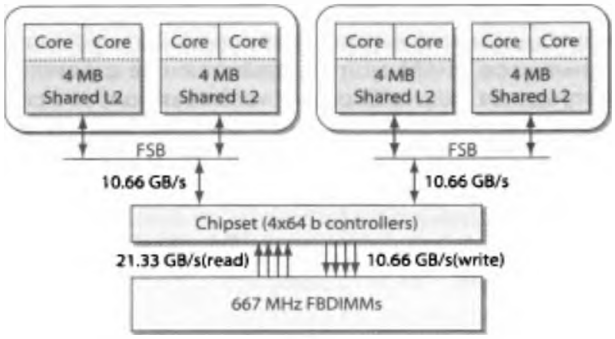
На рис. 6.13 показана основная организация четырех систем, а в табл. 7.5 перечислены ключевые характеристики, необходимые для примеров данного раздела. Все эти системы имеют по два процессорных гнезда. На рис. 6.14 показана roofline-модель производительности для каждой системы.

Таблица 7.5. Характеристики четырех сравнительно недавно выпущенных многоядерных микропроцессоров. Хотя у Xeon e5345 и у Opteron X4 имеются DRAM-модули, работающие с одинаковой скоростью, контрольная задача Stream показывает более высокую практическую пропускную способность из-за неэффективности фронтальной шины процессора Xeon e5345

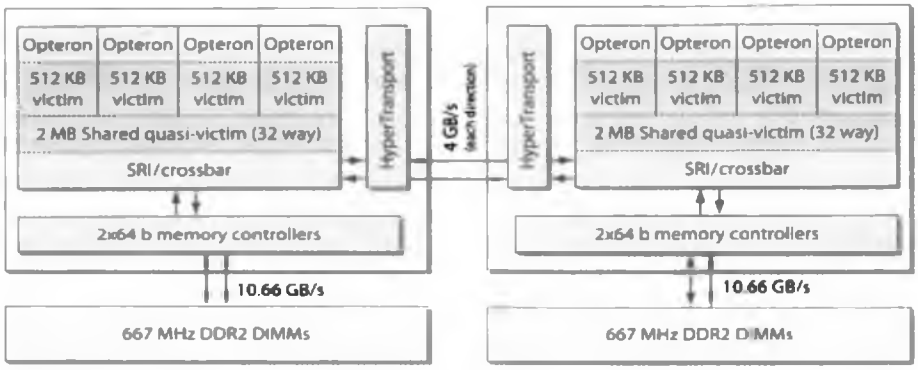
Тип MPU	ISA	Кол-во потоков	Кол-во ядер	Кол-во гнезд	Тактовая частота, ГГц	Пиковая, Гфлоп/с	DRAM	
							Пиковая, Гбайт/с	Тактовая частота, тип
Intel Xeon e5345 (Clovertown)	x86/64	8	8	2	2,33	75	FSB: 2 x 10,6	667 МГц; FBDIMM
AMD Opteron X4 2356 (Barcelona)	x86/64	8	8	2	2,30	74	2 x 10,6	667 МГц; DDR2
Sun UltraSPARC T2 5140 (Niagara 2)	Sparc	128	16	2	1,17	22	2 x 21,3 (красный); 2 x 10,6 (белый)	667 МГц; FBDIMM
IBM Cell QS20	Cell	16	16	2	3,20	29	2 x 25,6	XDR

Процессор Intel Xeon e5345 (кодовое название «Clovertown») содержит по четыре ядра на гнездо за счет объединения двух двуядерных кристаллов в одном процессорном гнезде. Эти два кристалла совместно используют фронтальную шину, подключенную к отдельному чипсету северного моста (см. главу 6). Этот чипсет северного моста поддерживает две фронтальные шины, а следовательно, два гнезда. Он включает контроллер памяти для 667 МГц полностью буферизованных DRAM-модулей DIMM (FBDIMM). Эта система, имеющая два процессорных гнезда, использует тактовую частоту процессора 2,33 ГГц и имеет наивысшую пиковую производительность этих четырех кристаллов, равную 75 Гфлоп/с. Но roofline-модель, изображенная на рис. 7.14, показывает, что она может быть получена только при арифметической интенсивности, равной и более 8. Двойные фронтальные шины мешают друг другу, что приводит к относительно низкой пропускной способности памяти для программ.

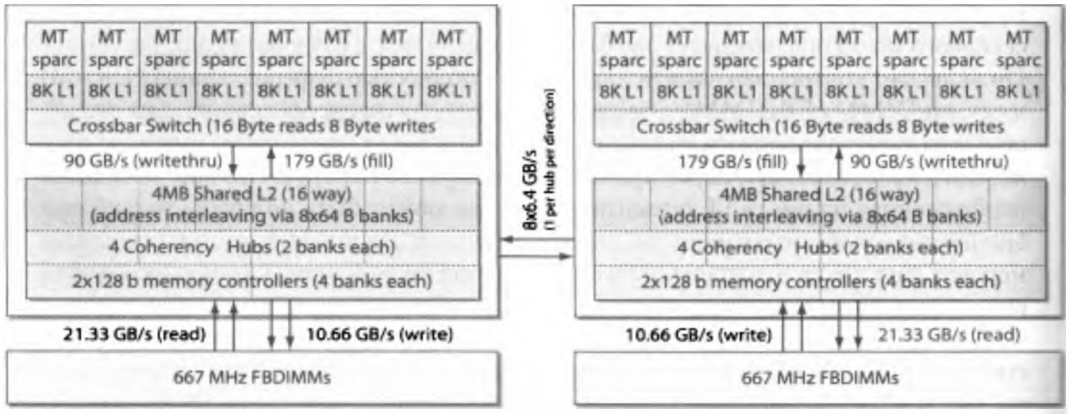
Процессор AMD Opteron X4 2356 (Barcelona) содержит четыре ядра на кристалл, и в каждом гнезде находится один кристалл. Каждый кристалл имеет в своем составе контроллер памяти и свой собственный маршрут к 667 МГц DDR2 DRAM. Эти два гнезда обмениваются данными через отдельные, специальные гипертранспортные соединения, позволяющие создавать «несклеенные» многокристальные системы. Эта система с двумя процессорными гнездами использует тактовую частоту процессора 2,30 ГГц и обладает пиковой производительностью около 74 Гфлоп/с. На рис. 6.14 показано, что ребро в roofline-модели находится левее, чем у процессора Xeon e5345 (Clovertown), на отметке арифметической интенсивности около 5 флоп/байт.


Core
Core
4 MB
Shared L2
FSB
10.66 GB/s
Chipset (4x64 b controllers)
21.33 GB/s(read)
10.66 GB/s(write)
667 MHz FBDIMMs

а


Opteron
512 KB victim
2 MB Shared quasi-victim (32 way)
SRI/crossbar
2x64 b memory controllers
HyperTransport
4 GB/s (each direction)
10.66 GB/s
667 MHz DDR2 DIMMs

б


MT sparc
8K L1
Crossbar Switch (16 Byte reads 8 Byte writes
90 GB/s (writethru)
179 GB/s (fill)
4MB Shared L2 (16 way)
(address interleaving via 8x64 B banks)
4 Coherency Hubs (2 banks each)
2x128 b memory controllers (4 banks each)
21.33 GB/s (read)
10.66 GB/s (write)
8x6.4 GB/s
(1 per hub per direction)
667 MHz FBDIMMs

в

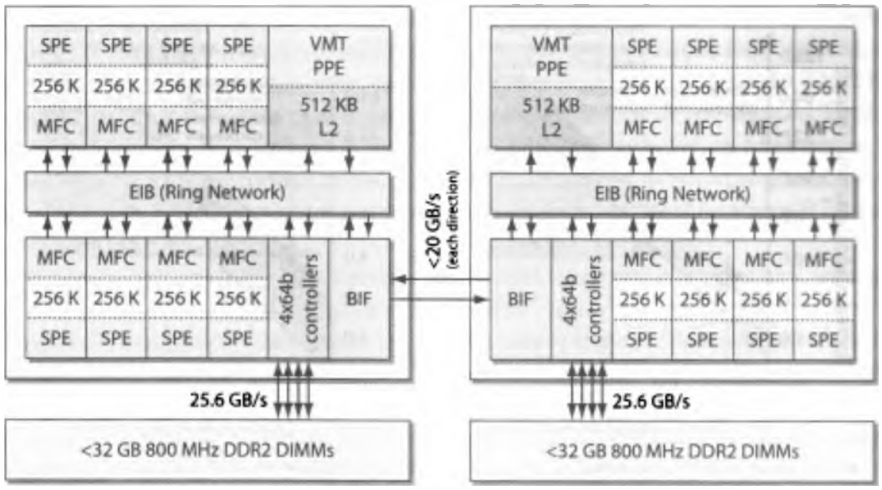


Рис. 7.13. Четыре сравнительно недавно выпущенных мультипроцессора, каждый из которых использует два процессорных гнезда. Начиная с левого верхнего угла здесь представлены сведения о компьютерах: *а*) Intel Xeon e5345 (Clovertown), *б*) AMD Opteron X4 2356 (Barcelona), *в*) Sun UltraSPARC T2 5140 (Niagara 2) и *г*) IBM Cell QS20. Учтите, что у Intel Xeon e5345 (Clovertown) имеется отдельная микросхема северного моста, чего нет у других микропроцессоров

Процессор Sun UltraSPARC T2 5140 (кодовое название «Niagara 2») сильно отличается от двух микроархитектур x86. В нем используется восемь относительно простых ядер на кристалл с куда более низкой тактовой частотой. Он также предоставляет мелкомодульную многопоточность с восемью потоками на ядро. Отдельная микросхема имеет четыре контроллера памяти, способные управлять четырьмя наборами 667 МГц FBDIMM-модулей. Чтобы объединить два кристалла UltraSPARC T2, подключены два из четырех каналов памяти, оставляя по два канала памяти на кристалл. Эта система с двумя процессорными гнездами имеет пиковую производительность около 22 Гфлоп/с, а ребро находится на невероятно низкой отметке арифметической интенсивности, составляющей всего 1/3 флоп/байт.

Процессор IBM Cell QS20 также отличается от двух микроархитектур x86 и от UltraSPARC T2. Это гетерогенная конструкция с относительно простыми ядрами PowerPC и с восемью SPE (Synergistic Processing Elements — совместно работающими обрабатывающими элементами), у которых имеется свой собственный набор инструкций SIMD-типа. Каждый SPE также имеет вместо кэш-памяти свою собственную локальную память. Элемент SPE должен передавать данные из оперативной памяти в локальную память, чтобы работать с ними, а затем, по окончании обработки, возвращать их в оперативную память. Он использует блок DMA, у которого имеется нечто подобное программной предвыборке. Два процессорных гнезда соединены друг с другом посредством связей, специализированных под обмен данными между кристаллами. Тактовая частота этой системы самая высокая для этих четырех многоядерных систем — 3,2 ГГц, и в данной системе используются микросхемы XDR DRAM, которые обычно применяются в игровых консолях. Они

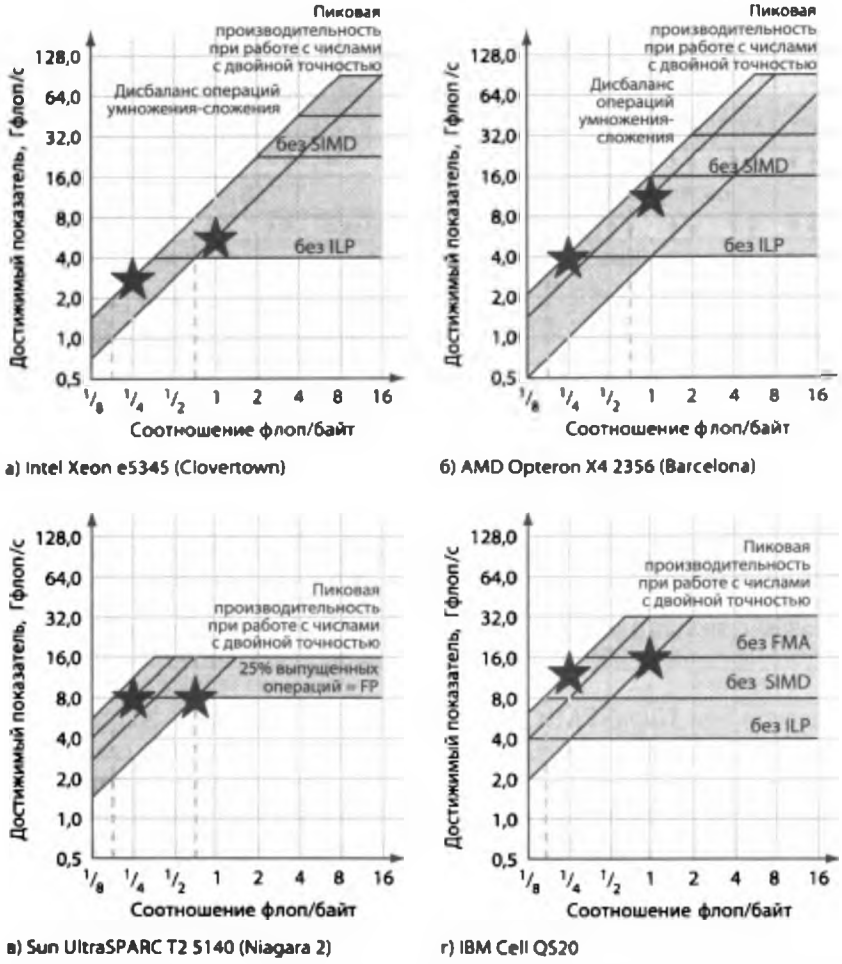


Рис. 7.14. Roofline-модель для многоядерных микропроцессоров, показанная на рис. 7.11. Потолки здесь такие же, как и на рис. 7.11. Начиная с левого верхнего угла здесь представлены сведения о компьютерах: *а*) Intel Xeon e5345 (Clovertown), *б*) AMD Opteron X4 2356 (Barcelona), *в*) Sun UltraSPARC T2 5140 (Niagara 2) и *г*) IBM Cell QS20. Обратите внимание на то, что для четырех микропроцессоров проекции из точек ребер на ось X, пересекают эту ось в точках арифметической интенсивности 6, 4, 1/3 и 3/4 соответственно. Пунктирные вертикальные линии показаны для двух базовых компонентов, рассматриваемых в данном разделе, а звездочками помечена производительность, достигнутая для этих базовых компонентов после всех оптимизаций. Базовый компонент SpMV (Sparse Matrix—Vector multiply — разреженная матрица — векторное умножение) представлен парой пунктирных вертикальных линий слева. У него две линии, потому что его арифметическая интенсивность благодаря оптимизации блокировки регистров улучшается с 0,166 до 0,255. Базовый компонент LBHMD (Lattice-Boltzmann Magneto—Hydrodynamics — модель имитации магнитогидродинамики Латтиса—Больцмана) представлен пунктирными вертикальными линиями справа. На рисунках *а* и *б* у него пара линий, потому что благодаря оптимизации кэш-памяти пропускается заполнение блока кэш-памяти при промахах, когда процессор мог бы записать новые данные в целый блок. Эта оптимизация повышает арифметическую интенсивность с 0,70 до 1,07. На рисунке *в* этот базовый компонент представлен одной линией, пересекающей ось X в точке 0,70, потому что UltraSPARC T2 не предлагает оптимизации кэш-памяти. На рисунке *г* он также представлен одной линией, пересекающей ось X в точке 1,07, поскольку Cell имеет локальное хранилище данных, загружаемое блоком DMA, программа не извлекает ненужные данные, как это делается в случает использования устройств кэш-памяти

обладают высокой пропускной способностью, но небольшой емкостью. Поскольку основные приложения для Cell имели графический характер, эта система обладает намного более высокой производительностью при работе с числами с одинарной точностью, чем при работе с числами с двойной точностью. Пиковая производительность при работе с числами с двойной точностью элементов SPE в системе с двумя процессорными гнездами составляет 29 Гфлоп/с, и проекция точки ребра на ось X пересекает ее в точке арифметической интенсивности 0,75 флоп/байт.

Хотя две архитектуры x86 имеют намного меньше ядер на кристалл, чем предлагалось компаниями IBM и Sun в начале 2008 года, это только лишь текущие показатели. В ожидании удвоения количества ядер с появлением каждого технологического поколения интересно узнать, закроют ли архитектуры x86 этот «ядерный разрыв», если IBM и Sun смогут поддержать большее количество ядер с учетом того, что они главным образом ориентированы на серверы, а не на настольные компьютеры.

Обратите внимание на то, что эти машины используют совершенно разные подходы к системам памяти. Процессор Xeon e5345 использует обычную закрытую кэш-память уровня L1, а затем каждая пара процессоров совместно использует устройство кэш-памяти уровня L2. Связь с обычной памятью осуществляется через контроллер памяти на отдельном кристалле по двум шинам. В отличие от этого у процессора Opteron X4 имеется отдельный контроллер памяти и память для каждого кристалла, и у каждого ядра есть закрытые устройства кэш-памяти уровней L1 и L2. У процессора UltraSPARC T2 имеется контроллер памяти на кристалле и четыре отдельных DRAM-канала на каждый кристалл, и все ядра совместно используют кэш-память уровня L2, которая для повышения пропускной способности имеет четыре банка. Мелкомодульная многопоточность этого процессора, надстроенная над его многоядерной конструкцией, позволяет ему на лету поддерживать множественные обращения к памяти. Наиболее радикальным является процессор Cell. У него имеются локальные закрытые устройства памяти на каждый элемент SPE и для передачи данных между DRAM-модулями, прикрепленными к каждому кристаллу, и локальной памятью используется блок DMA. Он на лету поддерживает множественные обращения к памяти за счет имеющегося большого количества ядер, а также за счет множества DMA-передач на ядро.

Давайте посмотрим, как эти настолько отличающиеся друг от друга четыре многоядерные системы выполняют два базовых компонента пакета контрольных задач.

Разреженная матрица

Первый пример базового компонента, относящийся к вычислительному шаблону конструирования Разреженная матрица, — это Sparse Matrix–Vector multiply (SpMV — разреженная матрица — векторное умножение). SpMV приобрел популярность в научных вычислениях, экономическом моделировании и в информационном поиске. К сожалению, обычные реализации зачастую работают менее чем на 10% от пиковой производительности однопроцессорных систем. Одна из

причин состоит в нерегулярных обращениях к памяти, которые могут ожидаться от базовой части, работающей с разреженными матрицами. Вычисление можно представить в виде

$$y = A \times x,$$

где A — это разреженная матрица, а x и y — это уплотненные вектора. Для вычисления SpMV-производительности используются четырнадцать разреженных матриц, взятых из различных приложений, но на выходе получается только средняя производительность. Арифметическая интенсивность варьируется от 0,166 до оптимизации блокировки регистров и до 0,250 флоп/байт после этой оптимизации.

Сначала, чтобы задействовать все ядра, код подвергается распараллеливанию. С учетом низкой арифметической интенсивности и того, что SpMV была ниже точки ребра всех четырех многоядерных систем, показанных на рис. 7.14, большинство оптимизаций касается системы памяти:

- *Предвыборка*. Чтобы получить наибольшую отдачу от систем памяти, была использована как программная, так и аппаратная предвыборка.
- *Близость памяти*. Эта оптимизация сокращает количество обращений к DRAM-памяти, подключенной к другому процессорному гнезду в трех системах, имеющих локальную DRAM-память.
- *Сжатые структуры данных*. Поскольку пропускная способность памяти ограничивает производительность, эта оптимизация использует более мелкие структуры данных для повышения производительности — например, использование 16-разрядного индекса вместо 32-разрядного, и использование более эффективного с точки зрения пространства представления ненулевых значений в строках разряженной матрицы.

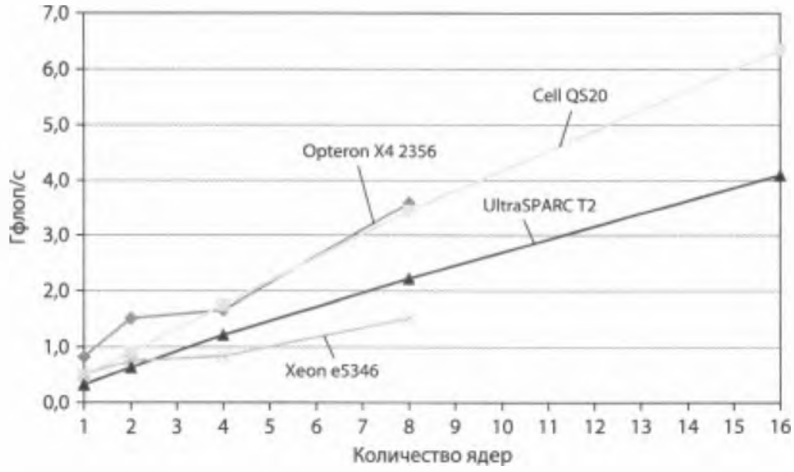


Рис. 7.15. Производительность SpMV на четырех многоядерных системах

На рис. 6.15 показана производительность при выполнении SpMV для четырех систем, отличающихся количеством ядер. (Такие же результаты можно найти на рис. 7.14, но там трудно сравнить производительность из-за логарифмической шкалы.) Обратите внимание, что несмотря на наличие наивысшей пиковой производительности, показанной в табл. 7.5, и наивысшей производительности отдельного ядра процессор Intel Xeon e5345 имеет самую низкую предоставленную производительность среди всех четырех многоядерных систем.

Opteron X4 удваивает его производительность. Узкое место процессора Xeon e5345 в двойных фронтальных шинах. Несмотря на самую низкую тактовую частоту, большее количество простых ядер процессора Sun UltraSPARC T2 позволяет ему превзойти производительность двух процессоров x86. Самая высокая производительность из всех четырех систем у процессора IBM Cell. Обратите внимание на то, что кроме Xeon e5345 все остальные системы неплохо масштабируются с ростом количества ядер, хотя Opteron X4 с четырьмя и более ядрами масштабируется медленнее.

Структурированная решетка

Вторым базовым компонентом является пример шаблона конструирования Структурированная решетка. Модель имитации магнитогидродинамики Латтиса–Больцмана – Lattice–Boltzmann Magneto–Hydrodynamics (LBMHD) популярна для вычислительной гидроаэродинамики; это структурированный решетчатый код с последовательностью временных интервалов.

Каждая точка вызывает чтение и запись около 75 чисел с плавающей точкой двойной точности и около 1300 операций с плавающей точкой. Как и SpMV, LBMHD обычно получает лишь небольшую часть пиковой производительности однопроцессорной системы из-за сложности структур данных и нерегулярности схем обращения к памяти. Соотношение флоп/байт намного выше 0,70 против менее 0,25 в SpMV. За счет отсутствия заполнения блока кэш-памяти из памяти при промахах записи, когда программа собирается переписать весь блок, интенсивность возрастает до 1,07. Такую оптимизацию кэш-памяти предлагают все многоядерные системы, кроме UltraSPARC T2 (Niagara 2).

На рис. 6.14 показано, что арифметическая интенсивность LBMHD достаточно высока для того, чтобы имелся смысл проведения оптимизаций, относящихся как к вычислениям, так и к пропускной способности памяти на всех многоядерных системах за исключением UltraSPARC T2, чья точка ребра линии крыши находится ниже LBMHD. UltraSPARC T2 может достичь линии крыши с использованием только вычислительных оптимизаций.

Вдобавок к распараллеливанию кода для LBMHD использовались следующие оптимизации:

- *Близость памяти*. Эта оптимизация полезна по тем же обоснованиям, которые приводились выше.
- *Минимизация промахов при обращении к TLB*. Для существенного сокращения TLB-промахов в LBMHD следует использовать структуру массивов и объ-

единять некоторые циклы, а не применять обычный подход – использование массива структур.

- *Развертывание и переупорядочение циклов.* Для раскрытия достаточного параллелизма и повышения задействованности кэш-памяти циклы подвергаются развертыванию и переупорядочению в группы инструкций со схожими адресами.
- *«SIMD-изация».* Компиляторы двух систем x86 не могут генерировать хороший SSE-код, поэтому он должен быть написан вручную на языке ассемблера.

На рис. 6.16 показана производительность четырех систем, отличающихся количеством ядер при выполнении контрольной задачи LBMHD. Как и при выполнении SpMV, процессор Intel Xeon e5345 имеет наихудшую масштабируемость. На этот раз более мощные ядра Opteron X4 превосходят по производительности простые ядра UltraSPARC T2. IBM Cell и на этот раз является самой быстрой системой. Все системы кроме Xeon e5345, масштабируются с ростом числа ядер, хотя T2 и Cell масштабируются более сглаженно, чем Opteron X4.

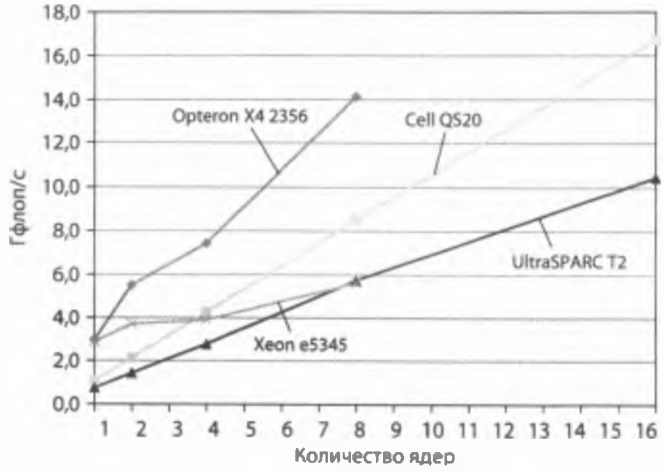


Рис. 7.16. Производительность LBMHD на четырех многоядерных системах

Продуктивность

Кроме производительности еще одним важным показателем революции параллельного вычисления является продуктивность, или программные трудности достижения производительности. Чтобы проиллюстрировать различия, в табл. 7.6 сравнивается простая производительность с производительностью после полной оптимизации для четырех систем и двух базовых компонентов.

Оказалось, что проще всего работать с системой UltraSPARC T2 благодаря ее высокой пропускной способности памяти и простым для понимания ядрам. Для этих двух базовых компонентов в UltraSPARC T2 можно просто посоветовать попытаться добиться высокопроизводительного кода от компилятора, а затем

использовать как можно больше потоков. При использовании UltraSPARC T2 можно впасть в заблуждение, касающееся необязательности обеспечения степени ассоциативности в соответствии с количеством аппаратно реализуемых потоков (см. раздел «Заблуждения и недоразумения» главы 5). Каждый кристалл поддерживает 64 аппаратных потока, а кэш-память уровня L2 имеет четырехканальную ассоциативность. Это несоответствие может потребовать реструктуризации циклов для сокращения конфликтов, связанных с промахами.

Таблица 7.6. Соотношение базовой производительности и производительности после полной оптимизации четырех систем и двух базовых компонентов. Обратите внимание на высокую долю производительности после полной оптимизации, предоставленную Sun UltraSPARC T2 (Niagara 2). Для IBM Cell столбец базовой производительности отсутствует из-за невозможности портировать код элементам SPE, не имеющим устройств кэш-памяти. Хотя код можно запустить на ядре Power, он имеет на порядок меньшую величину производительности, чем SPE-элементы, поэтому в данной таблице этот показатель проигнорирован

Тип MPU	Базовый компонент	Базовая производительность, Гфлоп/с	Оптимизированная производительность, Гфлоп/с	Приблизительный % оптимизированного
Intel Xeon e5345 (Clovertown)	SpMV LBMHD	1,0 4,6	1,5 5,6	64% 82%
AMD Opteron X4 2356 (Barcelona)	SpMV LBMHD	1,4 7,1	3,6 14,1	38% 50%
Sun UltraSPARC T2 (Niagara 2)	SpMV LBMHD	3,5 9,7	4,1 10,5	86% 93%
IBM Cell QS20	SpMV LBMHD	— —	6,4 16,7	0% 0%

Работать с Xeon e5346 оказалось сложнее, потому что трудно было понять поведение памяти при наличии двойных фронтальных шин, также трудно было разобраться, как работает аппаратная предвыборка, и получить от компилятора хороший SIMD-код. Код C для этой системы и для Opteron X4 щедро усыпан встроенными операторами, вызывающими SIMD-инструкции для получения хорошей производительности.

Система Opteron X4 получает преимущества от большинства типов оптимизаций, поэтому для нее нужно приложить больше усилий, чем для Xeon e5345, хотя поведение памяти Opteron X4 было проще понять, чем ее поведение в Xeon e5345.

У системы Cell есть два типа сложностей. Во-первых, SIMD-инструкции для SPE были неудобными для компиляции, поэтому приходилось постоянно помогать компилятору, вставляя в код на языке C нужные операторы, составленные из инструкций на языке ассемблера. Во-вторых, система памяти была намного

интереснее. Поскольку у каждого SPE имеется локальная память в отдельном адресном пространстве, код невозможно просто портировать и сразу же запускать на SPE. Поэтому для IBM Cell в табл. 7.6 отсутствует столбец базового кода, и вам нужно изменять программу для выдачи DMA-команд на передачу данных туда и обратно между локальным хранилищем и оперативной памятью. Благоприятным обстоятельством являлось то, что DMA играла роль программной предвыборки в устройствах кэш-памяти и DMA намного проще использовать и получать при этом высокую производительность памяти. Система Cell могла предоставить этим базовым компонентам почти 90% максимальной пропускной способности памяти, которая определяется «линией крыши», а не 50% и менее, предоставляемых другими многоядерными системами.

7.12. Заблуждения и недоразумения

> Более десяти лет пророки утверждали, что организация отдельного компьютера достигла своих пределов и что по-настоящему значимые усовершенствования могут быть сделаны только путем объединения большого количества компьютеров таким образом, чтобы можно было получить совокупные решения...
> Оказалось, что обоснованность применения отдельного процессора себя еще не исчерпала...
>
> *Джин Амдал*

Многочисленные атаки на параллельную обработку данных вскрыли большое количество заблуждений и недоразумений. В данном разделе будут рассмотрены три заблуждения.

Заблуждение. *Закон Амдала не применим к параллельным компьютерам.*

В 1987 году глава исследовательской организации заявил, что закон Амдала был нарушен на мультипроцессорной машине. Чтобы попытаться понять основные положения информационных сообщений, рассмотрим цитату из закона Амдала (публикация 1967 г., с. 483):

По данному вопросу можно сделать довольно очевидный вывод — усилия, потраченные на достижение высоких показателей параллельной обработки, будут тщетными, если они не сопровождаются почти такими же значимыми достижениями в скоростях последовательной обработки.

Это утверждение не должно терять свою истинность, так как небрежно написанная часть программы должна ограничивать производительность. Одно из толкований закона приводит к следующей лемме: части каждой программы не могут не быть последовательными, поэтому для количества процессоров должна быть экономически оправданная граница, скажем, 100. При демонстрации линейного ускорения при 1000 процессорах эта лемма опровергается, в связи с чем заявляется о несоблюдении закона Амдала.

В качестве подхода исследователи использовали нестрогое масштабирование: вместо того чтобы следовать условиям увеличения скорости в 1000 раз при том же наборе данных, они задали в 1000 раз больший объем вычислительной работы в сопоставимое время. Для их алгоритма последовательная часть программы была неизменной, независимой от объема вводимых данных, а вся остальная часть была полностью параллельной — следовательно, было получено линейное ускорение с использованием 1000 процессоров.

Совершенно очевидно, что закон Амдала применим и к параллельным процессорам. Эти исследования установили, что одним из основных применений более быстрых компьютеров является решение объемных задач, но при этом нужно понимать, как масштабируется алгоритм по мере роста объема задачи.

Заблуждение. *Пиковая производительность является проекцией наблюдаемой производительности.*

В разделе 7.11 показывается, к примеру, что Intel Xeon e5345, микропроцессор с наивысшей пиковой производительностью, был самым медленным из четырех многоядерных микропроцессоров для двух базовых категорий контрольных задач.

Компьютерная промышленность использует этот показатель в маркетинговых целях, а при использовании параллельных машин данное заблуждение только усиливается. Мало того что маркетологи используют почти недосягаемую пиковую производительность однопроцессорного узла, но затем они умножают этот показатель на количество процессоров, предполагая достижение абсолютного ускорения!

Закон Амдала показывает, насколько трудно достичь любой вершины; при перемножении возможностей двух составляющих перемножаются и их недостатки. Модель roofline помогает поместить пиковую производительность на место перспективного показателя.

***Недоразумение**: Программное обеспечение, позволяющее воспользоваться преимуществами мультипроцессорной архитектуры или оптимизированное под ее использование, не разработано.*

У отставания программного обеспечения от параллельных процессоров довольно долгая история. Чтобы показать суть проблемы, мы приведем только один пример, но таких примеров существует великое множество!

Довольно часто возникает проблема адаптации под мультипроцессорную среду тех программ, которые были разработаны под однопроцессорную систему. Например, операционная система SGI изначально защищала таблицу страниц одной блокировкой на основе предположения о нечастом характере распределения страниц. При этом на однопроцессорной системе проблем производительности не возникало. Но на мультипроцессорной системе для ряда программ это обстоятельство стало главным узким местом производительности. Рассмотрим программу, которая использует большое количество страниц, инициализируемых при запуске, что делает UNIX в отношении статически распределяемых страниц.

Предположим, что программа прошла распараллеливание, позволившее распределять страницы сразу нескольким процессам. Поскольку распределение страниц требует использования таблицы страниц, которая блокируется при любом случае

ее использования, даже ядро операционной системы, допускающее использование в операционной системе нескольких потоков, будет переведено в последовательный режим работы, если все процессы пытаются одновременно распределить свои страницы (а это именно то, что можно было бы ожидать во время инициализации!).

Такое переведение работы с таблицей страниц в последовательный режим исключает использование параллельного режима при инициализации и оказывает существенное влияние на общую производительность параллельной системы. Это узкое место производительности сохраняется даже для параллелизма на уровне заданий. Предположим, к примеру, что мы разбиваем программу параллельной обработки на отдельные задания и запускаем эти задания по одному на каждом процессоре, не имея для заданий ничего общего. (Именно так поступил один из пользователей, резонно полагая, что проблема производительности в его программе возникала из-за непреднамеренного использования общих ресурсов или создания взаимных помех.) К сожалению, блокировка по-прежнему переводит все задания в режим последовательной обработки, поэтому низкая производительность наблюдается даже при выполнении независимых друг от друга заданий.

Это недоразумение показывает, что есть такой тип ошибок производительности, которые при всей своей значимости имеют весьма тонкую природу и могут возникать при запуске программ на мультипроцессорных системах. Как и многие другие ключевые компоненты программного обеспечения, алгоритмы операционной системы и структуры данных при работе в мультипроцессорной среде должны быть пересмотрены. А только что рассмотренная проблема с заданиями довольно эффективно решается путем распространения блокировок на небольшие участки таблицы страниц.

7.13. Заключительные комментарии

> Все наши будущие разработки мы посвящаем многоядерным конструкциям. Мы уверены, что это ключевой пункт, изменяющий направление развития промышленности. <...> Это не какое-то новое течение. Это полная трансформация вычислительной техники...
>
> *Пол Отеллини*

Мечта создания компьютеров путем простого объединения процессоров витала в воздухе с дней зарождения компьютерной техники. Но создание и использование эффективных и практичных параллельных процессоров не отличалось особой скоростью. Такой невеликий прогресс ограничивался сложностью решения проблем программного обеспечения, а также затянувшимся процессом развития архитектуры мультипроцессоров, направленным на повышение удобств эксплуатации и увеличение отдачи.

В этой главе были рассмотрены многие сложные задачи, связанные с программным обеспечением, включая трудности написания программ, достигающих хоро-

шего ускорения, вызванные законом Амдала. Трудности создания программного обеспечения связаны с широким разнообразием архитектурных подходов и весьма ограниченным успехом и непродолжительностью жизни многих параллельных архитектур прошлого.

Как упоминалось в главе 1, индустрия информационных технологий связывает свое будущее с параллельными вычислениями. Хотя очень просто возразить, что и эти усилия потерпят неудачу, как и многие другие, предпринятые в прошлом, существует ряд причин, вселяющих надежду:

- Несомненно, растет роль **программного обеспечения, предоставляемого в виде службы**, и кластеры доказали, что их использование может быть весьма успешным. Предоставляя избыточность на самом высоком уровне, включая географически разбросанные по разным местам дата-центры, эти службы дают круглосуточный доступ клиентам по всему миру в режиме 24 × 7 × 365. Нетрудно представить, что и количество служб, приходящихся на один дата-центр, и количество дата-центров будет расти. Разумеется, такие дата-центры воспользуются многоядерными конструкциями, поскольку они уже могут использовать в своих приложениях тысячи процессоров.
- Набрало популярность использование параллельной обработки данных в таких областях, как научные и конструкторские вычисления. Эти области применения всегда требовали возрастающих объемов вычислений. Имеется также множество приложений, обладающих большим объемом естественной параллельности. В этой области применения также доминируют кластеры. Например, если воспользоваться отчетом Linpack за 2007 год, станет ясно, что кластеры представляли более 80% из 500 самых быстрых компьютеров. И тем не менее не все так гладко: программирование параллельных процессоров даже для этих приложений остается весьма сложной задачей. И все же эта группа также несомненно воспользуется многоядерными кристаллами, так как и здесь есть опыт применения сотен и тысяч процессоров.
- Все производители микропроцессоров для настольных компьютеров и серверов создают мультипроцессоры для достижения более высокой производительности, поэтому, в отличие от прежних времен, уже нет легких путей к повышению производительности последовательных приложений. Программисты, нуждающиеся в повышении производительности, должны распараллеливать свои программы или создавать новые программы, выполняющиеся в параллельном режиме.
- Несколько процессоров на одном кристалле позволяют достичь совершенно иной скорости коммуникаций, чем конструкции, состоящие из нескольких кристаллов, предлагая намного меньшую латентность и намного более высокую пропускную способность. Эти улучшения могут упростить достижение хорошей производительности.

Программное обеспечение, предоставляемое в виде службы
Вместо распространения программного обеспечения, устанавливающегося и запускающегося на компьютерах клиентов, это программное обеспечение запускается в удаленном месте и предоставляет доступ клиентам через Интернет, обычно через веб-интерфейс. Плата с клиентов взимается на основании использования этого программного обеспечения.

- В прошлом успехи микропроцессоров и мультипроцессоров определялись по-разному. При оценке производительности однопроцессорной системы разработчики микропроцессоров радовались, когда производительность одного потока росла как корень квадратный от увеличения площади кристалла кремния. Таким образом, в понятиях ресурсов они были довольны сублинейным повышением производительности. Успешность мультипроцессоров привыкли определять линейной функцией от количества процессоров, предполагая, что стоимость приобретения или стоимость администрирования n процессоров была в n раз больше, чем те же показатели для одного процессора. Теперь, когда параллелизм перешел на многоядерность, представленную на одном кристалле, мы можем воспользоваться традиционным определением успешности микропроцессора по сублинейному повышению производительности.
- Успехи компиляции в процессе выполнения со своевременным предоставлением нужного кода позволяют задуматься о самоадаптирующемся программном обеспечении, использующем преимущества увеличения количества ядер на кристалле и предоставляющем гибкость, недоступную при тех ограничениях, которые накладываются статическими компиляторами.
- В отличие от прежних времен, движение по распространению открытого кода привело к тому, что он стал составлять существенную часть разрабатываемого программного обеспечения. Лучшие инженерные решения могут победить устаревшие представления в умах разработчиков. Это движение использует новшества, стимулирующие внесение изменений в старое программное обеспечение, и приветствует появление новых языков и программных продуктов. Такое движение по поддержке открытого кода может быть очень полезным в наше время быстрых перемен.

Эта революция в аппаратно-программном интерфейсе, наверное, самый большой вызов в данной области за последние 50 лет. Она предоставит новые возможности по исследованиям и деловому применению как в области информационных технологий, так и за ее пределами, и компании, доминирующие в эру многоядерных систем, могут быть совсем не теми компаниями, которые доминировали в эру однопроцессорных систем. Может быть, именно вы будете тем самым новатором, который воспользуется возможностями, которые наверняка появятся в обозримом будущем.

7.14. Упражнения

Предоставлены Дэвидом Каэли (David Kaeli) из Северо-Восточного университета.

Упражнение 7.1

Сначала составьте список повседневных будничных дел. Например, вы можете встать с постели, принять душ, одеться, позавтракать, высушить волосы, почистить зубы и т. д. Сократите список до 10 дел.

7.1.1 [5] <7.2> Теперь разберитесь, какие из этих дел уже используют некую форму параллельности (например, одновременная чистка сразу всех зубов, а не одного в отдельно взятый период времени, переноска одной книги по сравнению с тем, что в портфель кладется сразу несколько книг и все они переносятся «параллельно»). Рассмотрите каждое из ваших дел, выполняется ли оно в параллельном режиме, а если нет, то почему.

7.1.2 [5] <7.2> Теперь рассмотрите, какие из дел могут выполняться параллельно (например, завтрак и прослушивание новостей). Расскажите при рассмотрении каждого дела, с каким из других дел оно может быть выполнено в паре.

7.1.3 [5] <7.2> Что можно изменить в текущих системах (например, в душе, одежде, телевизоре, машине) применительно к решению задач упражнения 7.1.2, чтобы увеличить количество параллельно выполняемых дел?

7.1.4 [5] <7.2> Оцените степень сокращения времени на все эти дела, если попытаться выполнить как можно больше дел в параллельном режиме.

Упражнение 7.2

Во многих компьютерных приложениях используются поиск и сортировка данных. Для сокращения времени выполнения этих трудоемких задач была придумана масса эффективных алгоритмов поиска и сортировки. В данном упражнении мы рассмотрим, как наилучшим образом можно распараллелить такие задачи.

7.2.1 [10] <7.2> Рассмотрим следующий алгоритм двоичного поиска (классический алгоритм типа «разделяй и властвуй»), который ищет значение `X` в отсортированном массиве `A`, состоящем из `N` элементов, и возвращает индекс соответствующего элемента:

```
BinarySearch(A[0..N-1], X) {
        low = 0
        high = N - 1
        while (low <= high) {
            mid = (low + high) / 2
            if (A[mid] > X)
                high = mid - 1
            else if (A[mid] < X)
                low = mid + 1
            else
                return mid // найден
        }
        return -1 // не найден
}
```

Предположим, что у вас для запуска `BinarySearch` имеется `Y` ядер на многоядерном процессоре. При условии, что число `Y` намного меньше числа `N`, определите коэффициент ускорения, который вы рассчитываете получить для значений `Y` и `N`. Начертите все в виде графика.

7.2.2 [5] <7.2> Теперь предположим, что `Y` равен `N`. Как это повлияет на ваши заключения по предыдущему ответу? Если вам поручили получить наилучший из возможных коэффициентов ускорения (при строгом масштабировании), объясните, как можно изменить данный код для его получения.

Упражнение 7.3

Рассмотрим следующий фрагмент кода на языке C:

```
for   (j=2;j<1000;j++)
    D[j] = D[j-1]+D[j-2];
```

Этому фрагменту будет соответствовать следующий фрагмент MIPS-кода:

```
      DADDIU  r2,r2,999
loop: L.D     f1, -16(f1)
      L.D     f2, -8(f1)
      ADD.D   f3, f1, f2
      S.D     f3, 0(r1)
      DADDIU  r1, r1, 8
      BNE     r1, r2, loop
```

С этими инструкциями связаны следующие показатели латентности (в циклах):

ADD.D	L.D	S.D	DADDIU	BNE
3	5	1	1	3

7.3.1 [10] <7.2> Сколько тактовых циклов займет выполнение всех инструкций в одной итерации показанного выше цикла?

7.3.2 [10] <7.2> Когда инструкция в более поздней итерации цикла зависит от значения данных, произведенного в более ранней итерации того же самого цикла, мы говорим, что это *зависимость, привнесенная циклом* между итерациями этого цикла. Определите зависимости, привнесенные циклом в показанном выше коде. Определите зависимые переменные программы и регистры, соответствующие им на уровне ассемблера. Переменную организации цикла `j` можно проигнорировать.

7.3.3 [10] <7.2> Развертывание цикла было описано в главе 4. Примените такое развертывание к данному циклу, а затем рассмотрите возможность запуска этого кода в системе с двумя узлами, распределенной памятью и передачей сообщений. Предположим, что мы собираемся использовать передачу сообщений, рассмотренную в разделе 7.4, где была представлена новая операция `send(x, y)`, отправляющая узлу `x` значение `y`, и операция `receive( )`, которая ждет посланное узлу значение. Предположим, что запуск операций `send` занимает один цикл (то есть более поздние операции на том же узле могут выполняться в следующем цикле), но отправка на узел-получатель занимает 4 цикла. Инструкции `receive` приостанавливают выполнение инструкций на узле до тех пор, пока они не получат сообщение. Создайте расписание для двух узлов при условии, что коэффициент развертывания тела цикла равен 4 (то есть тело цикла появится 4 раза). Подсчитайте количество тактовых циклов, которое займет выполнение программного цикла на системе с передачей сообщений.

7.3.4 [10] <7.2> Латентность внутренней сети играет большую роль в эффективности систем с передачей сообщений. Насколько быстрой должна быть внутренняя сеть, чтобы справиться с любым ускорением от использования распределенной системы, описанной в упражнении 7.3.3?

Упражнение 7.4

Рассмотрим следующий рекурсивный алгоритм mergesort (еще один классический алгоритм типа «разделяй и властвуй»). Впервые алгоритм mergesort был описан Джоном фон Нейманом (John von Neumann) в 1945 году. В его основу была положена идея о делении неотсортированного списка x из m элементов на два подсписка, размер которых составляет примерно половину исходного списка. Эта операция повторяется для каждого подсписка и продолжается до тех пор, пока не будут получены списки длиной в один элемент. Затем, начиная с подсписков длиной в один элемент, два подсписка «объединяются» (merge) в один отсортированный список.

```
Mergesort(m)
    var list left, right, result
    if length(m) ≤ 1
        return m
    else
        var middle = length(m) / 2
        for each x in m up to middle
            add x to left
        for each x in m after middle
            add x to right
        left = Mergesort(left)
        right = Mergesort(right)
        result = Merge(left, right)
        return result
```

Этап объединения выполняется следующим кодом:

```
Merge(left,right)
    var list result
    while length(left) > 0 and length(right) > 0
        if first(left) ≤ first(right)
            append first(left) to result
            left = rest(left)
        else
            append first(right) to result
            right = rest(right)
    if length(left) > 0
        append rest(left) to result
    if length(right) > 0
        append rest(right) to result
    return result
```

7.4.1 [10] <7.2> Предположим, что для запуска MergeSort у вас есть Y ядер в многоядерном процессоре. При условии, что Y намного меньше, чем длина m, length(m), определите коэффициент ускорения, который вы рассчитываете получить для значений Y и length(m). Начертите все в виде графика.

7.4.2 [10] <7.2> Теперь представьте, что Y равен length(m). Как это повлияет на ваши заключения по предыдущему ответу? Если вам поручили получить наилучший из возможных коэффициентов ускорения (при строгом масштабировании), объясните, как можно изменить данный код для его получения.

Упражнение 7.5

Вы пытаетесь испечь три черничных пирога по фунту весом. В пироге используются следующие ингредиенты:

1 чашка размягченного масла
1 чашка сахара
4 больших яйца
1 чайная ложка ванильного экстракта
1/2 чайной ложки соли
1/4 чайной ложки мускатного ореха
1 1/2 чашки муки
1 чашка черники

Рецепт приготовления одного пирога следующий:

Разогрейте духовку до 325°F (160°C). Смажьте маслом и посыпьте мукой форму для пирога. В большой чаше взбейте вместе миксером на средней скорости масло и сахар, пока смесь не станет светлой и рыхлой. Добавьте яйца, ваниль, соль и мускатный орех. Взбивайте все до полного перемешивания. Переключите миксер на малую скорость и добавьте муку частями по 1/2 чашки, взбивая все до полного перемешивания.

Аккуратно добавьте чернику. Распределите все равномерно в готовой для пирога форме. Выпекайте в течение 60 минут.

7.5.1 [5] <7.2> Ваша задача — как можно эффективнее испечь три пирога. Предполагая, что у вас имеется только одна духовка емкостью в один пирог, одна большая чаша, одна форма для выпечки и один миксер, составьте план самого быстрого изготовления трех пирогов. Определите узкие места в выполнении этой задачи.

7.5.2 [5] <7.2> Теперь предположим, что у вас есть три чаши, три формы для выпечки и три миксера. Насколько быстрее пойдет процесс при наличии этих дополнительных ресурсов?

7.5.3 [5] <7.2> Теперь предположим, что у вас есть два друга, готовых помочь вам испечь пироги, и большая духовка, вмещающая все три пирога. Как это изменит план, составленный при выполнении упражнения 7.5.1?

7.5.4 [5] <7.2> Сравните задачу по выпечке пирогов с выполнением трех итераций цикла на параллельном компьютере. Определите наличие параллелизма на уровне данных и параллелизма на уровне задач в цикле выпечки пирогов.

Упражнение 7.6

Перемножение матриц играет важную роль в ряде приложений. Две матрицы могут быть перемножены только в том случае, если количество столбцов первой матрицы будет равно количеству строк второй матрицы.

Давайте предположим, что у нас есть матрица A размером $m \times n$ и нам нужно перемножить ее на матрицу B размером $n \times p$. Их произведение можно выразить матрицей $m \times p$, обозначенной как AB (или $A \cdot B$). Если назначить матрицу $C = AB$ и с помощью c_{ij} обозначить элемент в матрице C на позиции (i, j), тогда

$$c_{i,j} = \sum_{r=1}^{n} a_{i,r} b_{r,j} = a_{i,1} b_{1,j} + a_{i,2} b_{2,j} + \ldots + a_{i,n} b_{n,j}$$

для каждого элемента i и j, где $1 \leqslant i \leqslant m$ и $1 \leqslant j \leqslant p$. Теперь мы хотим узнать, можно ли распараллелить вычисление C. Предположим, что элементы матрицы расположены в памяти последовательно в следующем порядке: $a_{1,1}$, $a_{2,1}$, $a_{3,1}$, $a_{4,1}$, ..., и т. д.

7.6.1 [10] <7.3> Предположим, что мы собираемся вычислить C на машине с одним ядром и общей памятью и на машине с четырьмя ядрами и общей памятью. Вычислите ускорение, которое ожидалось бы получить на машине с четырьмя ядрами, игнорируя любые проблемы памяти.

7.6.2 [10] <7.3> Выполните упражнение 7.6.1 еще раз при условии, что обновление до C сталкивается с промахом при обращении к кэш-памяти, вызванным отказом в совместном использовании, когда обновляются следующие друг за другом элементы в строке (то есть обновляется индекс i).

7.6.3 [10] <7.3> Как вы будете решать проблему отказа в совместном использовании, которая может возникнуть?

Упражнение 7.7

Рассмотрим следующие части двух разных программ, запущенных одновременно на четырех процессорах в симметричном многоядерном процессоре — symmetric multicore processor (SMP). Предположим, что до запуска этого кода обе переменные x и y имели значение 0.

Ядро 1: $x = 2$;

Ядро 2: $y = 2$;

Ядро 3: $w = x + y + 1$;

Ядро 4: $z = x + y$.

7.7.1 [10] <7.3> Какими в конечном результате будут все возможные значения w, x, y и z? Для каждого возможного результата объясните, как мы можем его получить. Вам потребуется изучить все возможные чередования инструкций.

7.7.2 [5] <7.3> Как можно сделать выполнение более определенным, чтобы была возможность получить только один набор значений?

Упражнение 7.8

В совместно используемой системе памяти с кэш-когерентным неоднородным доступом к памяти (cache-coherent, nonuniform-memory access — CC-NUMA) центральные процессоры и физическая память поделены на вычислительные узлы. У каждого центрального процессора имеются локальные устройства кэш-памяти. Для поддержки когерентности (целостности, или согласованности) памяти к каждому блоку кэш-памяти мы можем добавить бит состояния или же ввести специализированные каталоги памяти. При использовании каталогов каждый узел предоставляет специализированную аппаратную таблицу для управления

состоянием каждого блока памяти, которая является «локальной» по отношению к данному узлу. Размер каждого каталога является функцией от размера совместно используемого пространства CC-NUMA (для каждого блока памяти, локального по отношению к узлу, предоставляется отдельная запись). Если в кэш-памяти сохраняется информация, требующая согласованности, эта же информация добавляется к каждой кэш-памяти в каждой системе (то есть объем пространства памяти является функцией от количества блоков кэш-памяти, доступных во всех кэш-устройствах).

При решении следующих задач представим, что все узлы имеют одинаковое количество центральных процессоров и одинаковый объем памяти (то есть центральные процессоры и память равномерно поделены между узлами CC-NUMA-машины).

7.8.1 [15] <7.3> Если у нас есть P центральных процессоров, распределенных по T узлам CC-NUMA-системы, а у каждого центрального процессора имеется C блоков памяти и мы поддерживаем байт информации о когерентности в каждом блоке кэш-памяти, составьте уравнение, представляющее объем памяти, который будет иметься в устройствах кэш-памяти отдельного узла системы для поддержки когерентности. Не включайте в это уравнение фактическое пространство, используемое для хранения данных, берите в расчет только пространство, используемое для хранения информации, требующей согласованности.

7.8.2 [15] <7.3> Если в каждой записи каталога хранится байт информации для каждого центрального процессора, если наша CC-NUMA-система всего имеет S блоков памяти и в системе имеется T узлов, составьте уравнение, представляющее объем памяти, имеющийся в каждом каталоге.

Упражнение 7.9

Предположим, что в CC-NUMA-системе, описание которой представлено в упражнении 7.8, имеется четыре узла, у каждого из которых имеется одноядерный центральный процессор (и у каждого процессора есть свои собственные устройства кэш-памяти данных уровней L1 и L2). Устройство кэш-памяти данных уровня L1 имеет сквозной режим записи, а устройство кэш-памяти данных уровня L2 имеет режим отложенной записи. Пусть система имеет рабочую нагрузку, при которой один центральный процессор ведет запись по определенному адресу, а другие центральные процессоры читают те данные, которые он записал. Также предположим, что адрес, по которому ведется запись, изначально находится в памяти и отсутствует в локальной кэш-памяти. После записи обновляемый блок присутствует только в устройствах кэш-памяти уровней L1 и L2 того ядра, которое осуществляло запись.

7.9.1 [10] <7.3> Для систем, обеспечивающих когерентность, используя состояния блоков кэш-памяти, дайте описание обмена данными между узлами, который будет генерироваться, когда каждый из четырех узлов будет вести запись по уникальным адресам, после чего данные из каждого адреса, куда велась запись, считываются остальными тремя ядрами.

7.9.2 [10] <7.3> Для механизма обеспечения когерентности на основе использования каталогов дайте описание обмена данными между узлами, который будет генерироваться при выполнении одного и того же образца кода.

7.9.3 [20] <7.3> Еще раз выполните упражнения 7.9.1 и 7.9.2 при условии, что каждый центральный процессор теперь имеет четыре ядра и у каждого из ядер имеется устройство кэш-памяти данных уровня L1, но устройство кэш-памяти данных уровня L2 одно на все четыре ядра. Каждое ядро будет вести запись, за которой последует чтение, осуществляемое всеми пятнадцатью другими ядрами.

7.9.4 [10] <7.3> Рассмотрим систему, описание которой дано в упражнении 7.9.3, но теперь предположим, что каждое ядро ведет запись в два разных байта, хранящихся в одном и том же блоке кэш-памяти. Как это повлияет на трафик шины? Обоснуйте ответ.

Упражнение 7.10

На CC-NUMA-системе стоимость обращения к нелокальной памяти может ограничить наши возможности эффективного использования мультипроцессорной обработки. В следующей таблице даны показатели стоимости, связанные с обращениями к данным в локальной памяти, в сравнении с показателями, относящимися к нелокальной памяти, и локальность наших приложений выражена в пропорции локальных обращений.

Локальные загрузки-сохранения (в циклах)	Нелокальные загрузки-сохранения (в циклах)	% локальных обращений
20	100	50

Ответьте на следующие вопросы. Предположим, что обращения к памяти равномерно распределены по приложению и что мы можем продолжать обработку при обращении к памяти (реальные зависимости отсутствуют). Также предположим, что в каждом цикле может быть активной только одна операция с памятью. Изложите все предположения об упорядочении локальных операций с памятью в сравнении с нелокальными операциями.

7.10.1 [10] <7.3> Если в среднем доступ к памяти требуется каждые 75 циклов, то каким будет воздействие на наше приложение?

7.10.2 [10] <7.3> Если в среднем доступ к памяти требуется каждые 50 циклов, то каким будет воздействие на наше приложение?

7.10.3 [10] <7.3> Если в среднем доступ к памяти требуется каждые 100 циклов, то каким будет воздействие на наше приложение?

Упражнение 7.11

Задача об обедающих философах является классической задачей синхронизации и параллелизма. Философы сидят за круглым столом, занимаясь одним из двух дел: едой или размышлением. Когда они едят, они не размышляют, а когда они размышляют, они не едят. В центре стоит миска со спагетти. Между каждой парой

философов лежит вилка. В результате у каждого философа есть одна вилка слева от него и одна вилка справа. Учитывая особенности поедания спагетти, философу нужны для еды две вилки, и он может использовать только те вилки, которые лежат непосредственно слева и справа от него. Друг с другом философы не разговаривают.

7.11.1 [10] <7.4> Дайте описание сценария, при котором ни один из философов никогда не ест (то есть зависает). Какова последовательность событий, приводящих к возникновению этой проблемы?

7.11.2 [10] <7.4> Опишите, как вы сможете решить эту проблему путем ввода понятия приоритета? Но можем ли мы гарантировать равноправие всем философам? Поясните ответ.

Представим теперь, что мы наняли официанта, которому поручено распределять вилки между философами. Никто не может взять вилку, пока не получит разрешения официанта. Официант владеет всей информацией обо всех вилках. Затем, если мы введем правило, по которому философы будут всегда сначала просить разрешения взять ту вилку, что находится слева от них, прежде чем просить разрешения взять ту вилку, что находится справа, то тем самым мы сможем гарантированно избежать взаимной блокировки.

7.11.3 [10] <7.4> Мы можем реализовать запросы официанту либо в виде очереди запросов, либо в виде периодических повторных попыток запроса. При использовании очереди запросы обрабатываются в порядке их получения. Проблема использования очереди состоит в том, что мы не всегда можем обслужить философа, чей запрос находится в начале очереди (в силу недоступности ресурсов). Опишите сценарий с участием пяти философов, в котором предусматривается очередь, но облуживание не предоставляется даже при доступности вилок, для того чтобы поел другой философ (чей запрос находится глубже).

7.11.4 [10] <7.4> Решится ли проблема из упражнения 7.11.3, если запросы к официанту реализованы путем периодического повторения нашего запроса до тех пор, пока ресурсы не станут доступными? Обоснуйте ответ.

Упражнение 7.12

Рассмотрим следующие три устройства центральных процессоров:

Центральный процессор SS: двуядерный суперскалярный микропроцессор, предоставляющий возможности выполнения инструкций на двух функциональных блоках — functional units (FU) с изменением порядка этих инструкций. На каждом ядре одновременно может быть запущен только один поток.

Центральный процессор MT: обладает мелкомодульной многопоточностью, позволяя инструкциям из двух потоков выполняться в параллельном режиме (то есть у него имеются два функциональных блока), хотя в каждом тактовом цикле может быть запущена инструкция только из одного потока.

Центральный процессор SMT: позволяет инструкциям из двух потоков запускаться в параллельном режиме (то есть у него имеются два функциональных блока), и в одном цикле могут быть запущены на выполнение инструкции любого из двух потоков или же инструкции из обоих потоков.

Предположим, что для запуска на этих центральных процессорах у нас есть два потока — *X* и *Y*, которые включают в себя следующие операции:

Поток X	Поток Y
A1 — выполнение занимает 2 цикла	B1 — не имеет зависимостей
A2 — зависит от результата выполнения A1	B2 — конфликтует с B1 за право использования функционального блока
A3 — конфликтует с A2 за право использования функционального блока	B3 — не имеет зависимостей
A4 — зависит от результата выполнения A2	B4 — зависит от результата выполнения B2

Предположим, что на выполнение всех инструкций уходит один тактовый цикл, если не оговорено ничто другое или не возник конфликт.

7.12.1 [10] < 7.5> Предположим, что у вас имеется один центральный процессор типа SS. Сколько циклов займет выполнение этих двух потоков? Сколько слотов будет запущено впустую из-за возникновения конфликтов?

7.12.2 [10] < 7.5> Теперь предположим, что у вас имеется один центральный процессор типа MT. Сколько циклов займет выполнение этих двух потоков? Сколько слотов будет запущено впустую из-за возникновения конфликтов?

7.12.3 [10] < 7.5> Предположим, что у вас имеется один центральный процессор типа SMT. Сколько циклов займет выполнение этих двух потоков? Сколько слотов будет запущено впустую из-за возникновения конфликтов?

Упражнение 7.13

Для уменьшения стоимости управления современными высокопроизводительными серверами в настоящее время активно распространяется программное обеспечение для создания виртуальной среды. Такие компании, как VMWare, Microsoft и IBM, разработали целую линейку продуктов для виртуализации. Основная концепция, рассмотренная в главе 5, предусматривает возможность введения между оборудованием и операционной системой уровня гипервизора, позволяющего нескольким операционным системам совместно использовать одно и то же физическое оборудование. Уровень гипервизора отвечает за распределение ресурсов центрального процессора и памяти, а также за предоставления тех служб, которые обычно находятся в ведении операционной системы (например, за предоставление ввода-вывода).

Виртуализация предоставляет абстрактный взгляд на базовое оборудование со стороны принятых на обслуживание операционных систем и прикладных программ. Это требует от нас переосмысления того, как многоядерные и мультипроцессорные системы будут конструироваться в будущем для поддержки совместного использования центральных процессоров и устройств памяти несколькими операционными системами в параллельном режиме.

7.13.1 [30] <7.5> Выберите из предлагаемых сегодня на рынке два гипервизора, и найдите сходства и различия в том, как они проводят виртуализацию и управление базовым оборудованием (центральными процессорами и памятью).

7.13.2 [15] <7.5> Проанализируйте, какие изменения могут быть необходимы в будущей многоядерной платформе, чтобы она больше соответствовала запросам ресурсов, помещенных на таких системах. Например, может ли многопоточность

играть важную роль в снижении уровня соревновательности за вычислительные ресурсы?

Упражнение 7.14

Нужно выполнить показанный цикл как можно эффективнее. В нашем распоряжении две разные машины, MIMD и SIMD.

```
for (i=0; i < 2000; i++)
    for (j=0; j<3000; j++)
        X_array[i][j] = Y_array[j][i] + 200.
```

7.14.1 [10] <7.6> Покажите для четырехпроцессорной MIMD-машины последовательность MIPS-инструкций, которые будут выполняться на каждом центральном процессоре. Какое ускорение будет достигнуто на этой MIMD-машине?

7.14.2 [20] <7.6> Напишите для SIMD-машины, имеющей восемь параллельных функциональных SIMD-блоков, программу на ассемблере, используя ваши собственные SIMD-расширения к MIPS для выполнения цикла. Сравните количество инструкций, выполненных на SIMD-машине и на MIMD-машине.

Упражнение 7.15

Примером MISD-машины может служить систолическая матрица. Она представляет собой конвейерную сеть или «волновой фронт» элементов обработки данных. Каждому из этих элементов не нужен счетчик команд, поскольку выполнение переключается поступающими данными. Синхронизированная систолическая матрица ведет вычисления в режиме «жесткой конфигурации», где каждый процессор выполняет поочередные фазы вычисления и связи.

7.15.1 [10] <7.6> Рассмотрите предложенную реализацию систолической матрицы (информацию об этом можно найти в Интернете или в технических публикациях). Затем попробуйте запрограммировать цикл из упражнения 7.14, используя эту MISD-модель. Проанализируйте все трудности, с которыми придется столкнуться.

7.15.2 [10] <7.6> Проанализируйте схожести и различия между MISD- и SIMD-машинами. Используйте при этом терминологию параллелизма на уровне данных.

Упражнение 7.16

Допустим, что нам нужно выполнить DAXP-цикл из упражнения «Сравнение векторов с обычным кодом» раздела 7.6 в MIPS-ассемблере на процессоре NVIDIA 8800 GTX GPU, рассмотренном в данной главе. При решении этой задачи мы будем считать, что все математические операции производятся над числами с плавающей точкой одинарной точности (мы переименуем цикл в SAXP). Предположим, что на выполнение инструкций затрачивается следующее количество тактовых циклов.

Loads	Stores	Add.S	Mult.S
4	1	2	5

7.16.1 [20] <7.7> Опишите, как вы будете составлять варпы для SAXP-цикла, чтобы задействовать восемь ядер, предоставляемых в одном мультипроцессоре.

Упражнение 7.17

Загрузите с веб-сайта www.nvidia.com/object/cuda_get.html CUDA Toolkit и SDK. Воспользуйтесь «emurelease» (режим эмуляции) версией кода (для этого назначения не понадобится настоящего оборудования NVIDIA). Создайте программы примеров, предоставляемые в SDK, и удостоверьтесь, что они выполняются на эмуляторе.

7.17.1 [90] <7.7> Используя в качестве отправной точки имеющийся в SDK образец «template», напишите CUDA-программу для выполнения следующих векторных операций:

1) $a - b$ (вычитание вектора из вектора);
2) $a \cdot b$ (скалярное произведение векторов).

Скалярное произведение двух векторов $a = [a_1, a_2, ..., a_n]$ и $b = [b_1, b_2, ..., b_n]$ определено как:

$$a \cdot b = \sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + ... + a_n b_n.$$

Представьте код для каждой программы, которая демонстрирует каждую операцию и проверяет правильность результатов.

7.17.2 [90] <7.7> Если вам доступно оборудование GPU, завершите анализ производительности вашей программы, проверив время вычисления для ее GPU-версии и для CPU-версии для диапазона размеров векторов. Объясните все полученные результаты.

Упражнение 7.18

Недавно компанией AMD было объявлено, что она интегрирует графическое процессорное устройство с их x86-ядрами в едином наборе, хотя и с разными тактовыми частотами для каждого из ядер. Это станет примером гетерогенной мультипроцессорной системы, чье промышленное производство мы ожидаем в ближайшем будущем. Одной из ключевых конструктивных особенностей станет разрешение быстрого обмена данными между центральным и графическим процессорами.

Сегодня обмен данными должен выполняться между отдельными кристаллами центрального и графического процессоров. Но это положение вещей изменяется в разрабатываемой AMD архитектуре Fusion (слияние). На данный момент для облегчения внутреннего обмена данными планируется использовать несколько (минимум 16) каналов PCI express. Компания Intel также стремительно врывается

на рынок со своим кристаллом Larrabee. Intel рассчитывает использовать свою технологию внутреннего обмена данными QuickPath.

7.18.1 [25] <7.7> Сравните пропускную способность и показатели латентности, связанные с этими двумя технологиями внутреннего обмена данными.

Упражнение 7.19

Посмотрите на рис. 7.6, *б*, где показана топология внутренних соединений *n*-куба третьего порядка, в котором объединены восемь узлов. Одной из привлекательных особенностей сети внутренних соединений *n*-куба является возможность справляться с повреждениями линий без потери возможности связи.

7.19.1 [10] <7.8> Выведите уравнение, вычисляющее количество связей в *n*-кубе (где *n* — это порядок куба), которые могут быть разорваны при сохранении гарантий, что существующие неразорванные связи обеспечат связь с любым узлом *n*-куба.

7.19.2 [10] <7.8> Сравните отказоустойчивость *n*-куба с полностью подключенной сетью внутренних соединений с таким же количеством узлов. Отобразите графически сравнение надежности в виде функции от количества связей, которые могут быть разорваны для обеих топологий.

Упражнение 7.20

Тестирование с помощью контрольных задач является областью исследований, часть которых — нахождение эталонных рабочих нагрузок для запуска на определенных компьютерных платформах с целью объективного сравнения производительности двух систем. В данном упражнении будут сравниваться два класса контрольных задач: Whetstone CPU benchmark и PARSEC benchmark. Выберите одну из программ из набора PARSEC. Все программы должны быть в свободном доступе в Интернете. Рассмотрите запуск нескольких копий Whetstone в сравнении с решением контрольной задачи PARSEC на любой из систем, приведенных в разделе 7.11.

7.20.1 [60] <7.9> В чем состоит существенное различие между этими двумя классами рабочей нагрузки при выполнении задач на этих многоядерных системах?

7.20.2 [60] <7.9, 7.10> Насколько в понятиях roofline-модели полученные вами результаты будут зависимы при выполнении этих контрольных задач от степени присутствия совместно используемых ресурсов и синхронизации в используемой рабочей нагрузке?

Упражнение 7.21

При выполнении вычислений с разреженными матрицами латентность в иерархии памяти становится одним из решающих факторов. У разреженных матриц наблюдается недостаток пространственной локальности в потоке данных, который обычно

встречается в матричных операциях. В результате этого были предложены новые матричные представления.

Одним из наиболее ранних представлений разреженной матрицы стал Yale Sparse Matrix Format. Он хранит исходную разреженную матрицу M, размером $m \times n$, в строковой форме, используя три одномерных массива. Пусть R обозначает количество ненулевых элементов в M; мы можем построить массив A длиной R, состоящий из всех ненулевых элементов M (в порядке слева направо и сверху вниз). Мы также построим массив IA длиной $m + 1$ (то есть один элемент на каждую строку плюс еще один). $IA(i)$ содержит индекс в A первого ненулевого элемента строки i. Строка i исходной матрицы простирается от $A(IA(i))$ до $A(IA(i + 1)-1)$. Третий массив JA содержит индекс столбцов каждого элемента A, поэтому он также имеет длину R.

7.21.1 [15] <7.9> Рассмотрим показанную ниже разреженную матрицу X и напишем код на языке C, который сохранит эту матрицу в формате Yale Sparse Matrix Format.

```
Строка 1 [0. 0. 0. 0. 10]
Строка 2 [0. 0. 0. 0.  0]
Строка 3 [8. 0. 0. 0.  6]
Строка 4 [0. 1. 8. 7.  0]
Строка 5 [7. 0. 0. 0.  0]
```

7.21.2 [10] <7.9> Используя понятие пространства памяти, предположим, что каждый элемент матрицы X имеет формат числа с плавающей точкой одинарной точности. Вычислите объем памяти, используемый для хранения показанной выше матрицы в формате Yale Sparse Matrix Format.

7.21.3 [15] <7.9> Выполните перемножение матрицы X с показанной ниже матрицей Y.

```
[9, 8, 7, 100, 2]
```

Поместите это вычисление в цикл и засеките время его выполнения. Увеличьте количество повторений этого цикла, чтобы получить приемлемые показатели измерения времени. Сравните время выполнения при использовании обычного представления матрицы и ее представления в формате Yale Sparse Matrix Format.

7.21.4 [15] <7.9> Можете ли вы найти более эффективный формат представления разреженной матрицы (в понятиях пространства и издержек вычислений)?

Упражнение 7.22

В будущих системах мы ожидаем увидеть гетерогенные вычислительные платформы, созданные на основе гетерогенных центральных процессоров. Их появление уже наблюдается на рынке встроенных вычислительных устройств в системах, содержащих в однокристальном модуле цифровые сигнальные процессоры и микроконтроллерные центральные процессоры.

Предположим, что у вас есть три класса центральных процессоров:

ЦПУ A — среднескоростной многоядерный центральный процессор (с блоком вычислений с плавающей точкой), способный выполнять несколько инструкций за один цикл.

ЦПУ *B* – быстрый одноядерный целочисленный центральный процессор (то есть не имеющий блока вычислений с плавающей точкой), способный выполнить одну инструкцию за один тактовый цикл.

ЦПУ *C* – медленный векторный центральный процессор (с возможностью вычислений с плавающей точкой), способный выполнять несколько копий одной и той же инструкции за один тактовый цикл.

Предположим, что наши процессоры работают на следующих тактовых частотах:

ЦПУ A	ЦПУ B	ЦПУ C
1,5 ГГц	3 ГГц	500 МГц

ЦПУ *A* может выполнить 2 инструкции за цикл, ЦПУ *B* может выполнить 1 инструкцию за цикл, а ЦПУ *C* может выполнить 8 (хотя и одинаковых) инструкций за цикл. Предположим, что все операции могут завершить выполнение с одинаковым количеством циклов латентности и без конфликтов.

У всех трех центральных процессоров есть возможность выполнять операции целочисленной арифметики, а ЦПУ *B* не может непосредственно выполнять арифметические операции с плавающей точкой. Центральные процессоры *A* и *B* имеют набор инструкций, похожий на тот, что имеется у MIPS-процессора. ЦПУ *C* может выполнять только операции сложения и вычитания с плавающей точкой, а также операции загрузки из памяти и сохранения в ней. Предположим, что все ЦПУ имеют доступ к общей памяти, а затраты на синхронизацию отсутствуют.

Задача состоит в том, чтобы сравнить две матрицы, *X* и *Y*, каждая из которых содержит 1024 × 1024 элементов, представляющих собой числа с плавающей точкой. Необходимо подсчитать количество случаев, в которых значение в *X* было больше значения в *Y*.

7.22.1 [10] <7.11> Опишите, как вы станете делить задачу между тремя различными ЦПУ для получения наилучшей производительности.

7.22.2 [10] <7.11> Какой тип инструкций вы добавили бы к векторному ЦПУ *C* для получения лучшей производительности?

Упражнение 7.23

Предположим, что компьютерная система, имеющая четыре ядра, может обработать транзакцию базы данных в установившемся скоростном режиме поступления запросов в секунду. Также предположим, что на обработку каждой транзакции уходит фиксированное количество времени. В следующей таблице показаны пары латентности транзакции и скорости ее обработки.

Средняя латентность транзакции	Максимальная скорость обработки транзакции
1 мс	5000/с
2 мс	5000/с
1 мс	10 000/с
2 мс	10 000/с

Ответьте на следующие вопросы для каждой показанной в таблице пары:

7.23.1 [10] <7.11> Сколько в среднем запросов обрабатывается в любой заданный момент времени?

7.23.2 [10] <7.11> Что в идеале произойдет с пропускной способностью при переходе на восьмиядерную систему (то есть сколько транзакций в секунду будет обрабатывать процессор)?

7.23.3 [10] <7.11> Объясните, почему мы редко получаем подобного рода ускорение путем простого увеличения количества ядер.

Ответы на вопросы для самопроверки

Раздел 7.1: Утверждение ложное. Параллелизм на уровне заданий может помочь последовательным приложениям, а последовательные приложения могут быть созданы для работы на параллельном оборудовании, хотя это и усложняет их разработку.

Раздел 7.2: Утверждение ложное. Нестрогое масштабирование может компенсировать последовательную часть программы, которая в противном случае ограничила бы масштабируемость.

Раздел 7.3: Утверждение ложное. Поскольку совместно используемым является физический адрес, несколько заданий, каждое из которых находится в своем собственном виртуальном адресном пространстве, может нормально выполняться на мультипроцессоре с общей памятью.

Раздел 7.4: 1. Утверждение ложное. Отправка и получение сообщений служит примером скрытой синхронизации, а также способом совместного использования данных. 2. Утверждение истинное.

Раздел 7.5: 1. Утверждение истинное. 2. Утверждение истинное.

Раздел 7.6: Утверждение истинное.

Раздел 7.7: Утверждение ложное. Графические DRAM DIMM-модули ценятся своей более высокой пропускной способностью.

Раздел 7.9: Утверждение истинное. Чтобы выиграть производственное соревнование в параллельных вычислениях, нам, наверное, нужны новшества во всем пакете аппаратного и программного обеспечения.

Дэвид Паттерсон, Джон Хеннесси

Архитектура компьютера и проектирование компьютерных систем. Классика Computers Science

4-е издание

Перевел с английского Н. Вильчинский

Заведующий редакцией	*А. Кривцов*
Руководитель проекта	*А. Юрченко*
Ведущий редактор	*Ю. Сергиенко*
Литературный редактор	*А. Гущин*
Художественный редактор	*К. Радзевич*
Корректоры	*С. Беляева, Н. Викторова*
Верстка	*Е. Егорова*

ООО «Мир книг», 198206, Санкт-Петербург, Петергофское шоссе, 73, лит. А29
Налоговая льгота — общероссийский классификатор продукции ОК 005-93, том 2; 95 3005 — литература учебная.
Подписано в печать 17.11.11. Формат 70x100/16. Усл. п. л. 63,210. Тираж 2000. Заказ 26874.
Отпечатано по технологии CtP в ОАО «Первая Образцовая типография», обособленное подразделение «Печатный двор».
197110, Санкт-Петербург, Чкаловский пр., 15.

ВАМ НРАВЯТСЯ НАШИ КНИГИ? ЗАРАБАТЫВАЙТЕ ВМЕСТЕ С НАМИ!

У Вас есть свой сайт?
Вы ведете блог?
Регулярно общаетесь на форумах? Интересуетесь литературой, любите рекомендовать хорошие книги и хотели бы стать нашим партнером?
ЭТО ВПОЛНЕ РЕАЛЬНО!

СТАНЬТЕ УЧАСТНИКОМ ПАРТНЕРСКОЙ ПРОГРАММЫ ИЗДАТЕЛЬСТВА «ПИТЕР»!

Зарегистрируйтесь на нашем сайте в качестве партнера по адресу ***www.piter.com/ePartners***

Получите свой персональный уникальный номер партнера

Выбирайте книги на сайте ***www.piter.com****, размещайте информацию о них на своем сайте, в блоге или на форуме и добавляйте в текст ссылки на эти книги (на сайт* ***www.piter.com****)*

ВНИМАНИЕ! *В каждую ссылку необходимо добавить свой персональный уникальный номер партнера.*

С этого момента получайте 10% от стоимости каждой покупки, которую совершит клиент, придя в интернет-магазин «Питер» по ссылке с Вашим партнерским номером. А если покупатель приобрел не только эту книгу, но и другие издания, Вы получаете дополнительно по 5% от стоимости каждой книги.

Деньги с виртуального счета Вы можете потратить на покупку книг в интернет-магазине издательства «Питер», а также, если сумма будет больше 500 рублей, перевести их на кошелек в системе Яндекс.Деньги или Web.Money.

Пример партнерской ссылки:
http://www.piter.com/book.phtml?9785388000282 – обычная ссылка
http://www.piter.com/book.phtml?9785388000282**&refer=0000** – партнерская ссылка, где 0000 – это ваш уникальный партнерский номер

Подробно о Партнерской программе ИД «Питер» читайте на сайте WWW.PITER.COM

ИЗДАТЕЛЬСКИЙ ДОМ ПИТЕР®
WWW.PITER.COM

КНИГА-ПОЧТОЙ

ЗАКАЗАТЬ КНИГИ ИЗДАТЕЛЬСКОГО ДОМА «ПИТЕР» МОЖНО ЛЮБЫМ УДОБНЫМ ДЛЯ ВАС СПОСОБОМ:

- на нашем сайте: **www.piter.com**
- по электронной почте: **postbook@piter.com**
- по телефону: **(812) 703-73-74**
- по почте: **197198, Санкт-Петербург, а/я 127, ООО «Питер Мейл»**
- по ICQ: **413763617**

ВЫ МОЖЕТЕ ВЫБРАТЬ ЛЮБОЙ УДОБНЫЙ ДЛЯ ВАС СПОСОБ ОПЛАТЫ:

- Наложенным платежом с оплатой при получении в ближайшем почтовом отделении.
- С помощью банковской карты. Во время заказа Вы будете перенаправлены на защищенный сервер нашего оператора, где сможете ввести свои данные для оплаты.
- Электронными деньгами. Мы принимаем к оплате все виды электронных денег: от традиционных Яндекс.Деньги и Web-money до USD E-Gold, MoneyMail, INOCard, RBK Money (RuPay), USD Bets, Mobile Wallet и др.
- В любом банке, распечатав квитанцию, которая формируется автоматически после совершения Вами заказа.

Все посылки отправляются через «Почту России». Отработанная система позволяет нам организовывать доставку Ваших покупок максимально быстро. Дату отправления Вашей покупки и предполагаемую дату доставки Вам сообщат по e-mail.

ПРИ ОФОРМЛЕНИИ ЗАКАЗА УКАЖИТЕ:

- фамилию, имя, отчество, телефон, факс, e-mail;
- почтовый индекс, регион, район, населенный пункт, улицу, дом, корпус, квартиру;
- название книги, автора, количество заказываемых экземпляров.

ИЗДАТЕЛЬСКИЙ ДОМ
ПИТЕР®
WWW.PITER.COM

КЛУБ ПРОФЕССИОНАЛ

Основанный Издательским домом «Питер» в 1997 году, книжный клуб «Профессионал» собирает в своих рядах знатоков своего дела, которых объединяет тяга к знаниям и любовь к книгам. Для членов клуба проводятся различные мероприятия и, разумеется, предусмотрены привилегии.

Привилегии для членов клуба:

- карта члена «Клуба Профессионал»;
- бесплатное получение клубного издания – журнала «Клуб Профессионал»;
- дисконтная скидка на всю приобретаемую литературу в размере 10 или 15%;
- бесплатная курьерская доставка заказов по Москве и Санкт-Петербургу;
- участие во всех акциях Издательского дома «Питер» в розничной сети на льготных условиях.

Как вступить в клуб?

Для вступления в «Клуб Профессионал» вам необходимо:

- совершить покупку на сайте **www.piter.com** или в фирменном магазине Издательского дома «Питер» на сумму от **1500** рублей без учета почтовых расходов или стоимости курьерской доставки;
- ознакомиться с условиями получения карты и сохранения скидок;
- выразить свое согласие вступить в дисконтный клуб, отправив письмо на адрес: postbook@piter.com;
- заполнить анкету члена клуба (зарегистрированным на нашем сайте этого делать не надо).

Правила для членов «Клуба Профессионал»:

- для продления членства в клубе и получения **скидки 10%** в течение каждых **6 месяцев** нужно совершать покупки на общую сумму от **1500** до **2500** рублей, без учета почтовых расходов или стоимости курьерской доставки;
- если же за указанный период вы выкупите товар на сумму от **2501** рубля, скидка будет увеличена до **15%** от розничной цены издательства.

Заказать наши книги вы можете любым удобным для вас способом:

- по телефону: (812)703-73-74;
- по электронной почте: postbook@piter.com;
- на нашем сайте: www.piter.com;
- по почте: 197198, Санкт-Петербург, а/я 127 ООО «Питер Мейл».

При оформлении заказа укажите:

- ваш регистрационный номер (если вы являетесь членом клуба), фамилию, имя, отчество, телефон, факс, e-mail;
- почтовый индекс, регион, район, населенный пункт, улицу, дом, корпус, квартиру;
- название книги, автора, количество заказываемых экземпляров.

ПИТЕР

Нет времени

ходить по магазинам?

наберите:

www.piter.com

Здесь вы найдете:

Все книги издательства сразу
Новые книги — в момент выхода из типографии
Информацию о книге — отзывы, рецензии, отрывки
Старые книги — в библиотеке и на CD

И наконец, вы нигде не купите наши книги дешевле!

ИЗДАТЕЛЬСКИЙ ДОМ
ПИТЕР®
WWW.PITER.COM

ПРЕДСТАВИТЕЛЬСТВА ИЗДАТЕЛЬСКОГО ДОМА «ПИТЕР»
предлагают эксклюзивный ассортимент компьютерной, медицинской, психологической, экономической и популярной литературы

РОССИЯ

Санкт-Петербург м. «Выборгская», Б. Сампсониевский пр., д. 29а
тел./факс: (812) 703-73-73, 703-73-72; e-mail: sales@piter.com

Москва м. «Электрозаводская», Семеновская наб., д. 2/1, корп. 1, 6-й этаж
тел./факс: (495) 234-38-15, 974-34-50; e-mail: sales@msk.piter.com

Воронеж Ленинский пр., д. 169; тел., факс: (4732) 39-61-70
e-mail: piterctr@comch.ru

Екатеринбург ул. Бебеля, д. 11а; тел./факс: (343) 378-98-41, 378-98-42
e-mail: office@ekat.piter.com

Нижний Новгород ул. Совхозная, д. 13; тел.: (8312) 41-27-31
e-mail: office@nnov.piter.com

Новосибирск ул. Станционная, д. 36; тел.: (383) 363-01-14
факс: (383) 350-19-79; e-mail: sib@msk.piter.com

Ростов-на-Дону ул. Ульяновская, д. 26; тел.: (863) 269-91-22, 269-91-30
e-mail: piter-ug@rostov.piter.com

Самара ул. Молодогвардейская, д. 33а; офис 223; тел.: (846) 277-89-79
e-mail: pitvolga@samtel.ru

УКРАИНА

Харьков ул. Суздальские ряды, д. 12, офис 10; тел.: (1038057) 751-10-02
758-41-45; факс: (1038057) 712-27-05; e-mail: piter@kharkov.piter.com

Киев Московский пр., д. 6, корп. 1, офис 33; тел.: (1038044) 490-35-69
факс: (1038044) 490-35-68; e-mail: office@kiev.piter.com

БЕЛАРУСЬ

Минск ул. Притыцкого, д. 34, офис 2; тел., факс: (1037517) 201-48-79, 201-48-81
e-mail: gv@minsk.piter.com

Ищем зарубежных партнеров или посредников, имеющих выход на зарубежный рынок.
Телефон для связи: **(812) 703-73-73**. **E-mail:** kuganov@piter.com

---

Издательский дом «Питер» приглашает к сотрудничеству авторов. Обращайтесь по телефонам: **Санкт-Петербург – (812) 703-73-72, Москва – (495) 974-34-50**

Заказ книг для вузов и библиотек по тел.: (812) 703-73-73.
Специальное предложение – e-mail: kozin@piter.com

Заказ книг по почте: на сайте **www.piter.com**; по тел.: (812) 703-73-74
по ICQ 413763617

ИЗДАТЕЛЬСКИЙ ДОМ ПИТЕР®
WWW.PITER.COM

**УВАЖАЕМЫЕ ГОСПОДА!
КНИГИ ИЗДАТЕЛЬСКОГО ДОМА «ПИТЕР»
ВЫ МОЖЕТЕ ПРИОБРЕСТИ
ОПТОМ И В РОЗНИЦУ У НАШИХ
РЕГИОНАЛЬНЫХ ПАРТНЕРОВ.**

ДАЛЬНИЙ ВОСТОК

Владивосток
«Приморский торговый дом книги»
тел./факс: (4232) 23-82-12
e-mail: bookbase@mail.primorye.ru

Хабаровск, «Деловая книга», ул. Путевая, д. 1а
тел.: (4212) 36-06-65, 33-95-31
e-mail: dkniga@mail.kht.ru

Хабаровск, «Книжный мир»
тел.: (4212) 32-85-51, факс: (4212) 32-82-50
e-mail: postmaster@worldbooks.kht.ru

Хабаровск, «Мирс»
тел.: (4212) 39-49-60
e-mail: zakaz@booksmirs.ru

ЕВРОПЕЙСКИЕ РЕГИОНЫ РОССИИ

Архангельск, «Дом книги», пл. Ленина, д. 3
тел.: (8182) 65-41-34, 65-38-79
e-mail: marketing@avfkniga.ru

Воронеж, «Амиталь», пл. Ленина, д. 4
тел.: (4732) 26-77-77
http://www.amital.ru

Калининград, «Вестер»,
сеть магазинов «Книги и книжечки»
тел./факс: (4012) 21-56-28, 6 5-65-68
e-mail: nshibkova@vester.ru
http://www.vester.ru

Самара, «Чакона», ТЦ «Фрегат»
Московское шоссе, д.15
тел.: (846) 331-22-33
e-mail: chaconne@chaccone.ru

Саратов, «Читающий Саратов»
пр. Революции, д. 58
тел.: (4732) 51-28-93, 47-00-81
e-mail: manager@kmsvrn.ru

СЕВЕРНЫЙ КАВКАЗ

Ессентуки, «Россы», ул. Октябрьская, 424
тел./факс: (87934) 6-93-09
e-mail: rossy@kmw.ru

СИБИРЬ

Иркутск, «ПродаЛитЪ»
тел.: (3952) 20-09-17, 24-17-77
e-mail: prodalit@irk.ru
http://www.prodalit.irk.ru

Иркутск, «Светлана»
тел./факс: (3952) 25-25-90
e-mail: kkcbooks@bk.ru
http://www.kkcbooks.ru

Красноярск, «Книжный мир»
пр. Мира, д. 86
тел./факс: (3912) 27-39-71
e-mail: book-world@public.krasnet.ru

Новосибирск, «Топ-книга»
тел.: (383) 336-10-26
факс: (383) 336-10-27
e-mail: office@top-kniga.ru
http://www.top-kniga.ru

ТАТАРСТАН

Казань, «Таис»,
сеть магазинов «Дом книги»
тел.: (843) 272-34-55
e-mail: tais@bancorp.ru

УРАЛ

Екатеринбург, ООО «Дом книги»
ул. Антона Валека, д. 12
тел./факс: (343) 358-18-98, 358-14-84
e-mail: domknigi@k66.ru

Екатеринбург, ТЦ «Люмна»
ул. Студенческая, д. 1в
тел./факс: (343) 228-10-70
e-mail: igm@lumna.ru
http://www.lumna.ru

Челябинск, ООО «ИнтерСервис ЛТД»
ул. Артиллерийская, д. 124
тел.: (351) 247-74-03, 247-74-09,
247-74-16
e-mail: zakup@intser.ru
http://www.fkniga.ru, www.intser.ru

САЛД

САНКТ-ПЕТЕРБУРГСКАЯ АНТИВИРУСНАЯ ЛАБОРАТОРИЯ ДАНИЛОВА

www.SALD.ru
8 (812) 336-3739

Антивирусные программные продукты

КЛАССИКА COMPUTER SCIENCE

Книга, выходящая уже в 4-м издании, посвящена структурной организации компьютера и отражает революционные изменения, происходящие в области аппаратного обеспечения, в частности стремительный переход от однопроцессорных систем к многоядерным микропроцессорам. В издании подробно описывается архитектура компьютера и устройство всех его компонентов: процессоров, блоков памяти, средств ввода-вывода и хранения данных. Отличительной особенностью книги является демонстрация взаимодействий между аппаратными средствами и системным программным обеспечением. Особое внимание уделяется многоядерным вычислительным системам и параллельному программированию. Многочисленные упражнения и задачи, приводимые после каждой темы, помогают закрепить материал.

Книга рассчитана на широкий круг читателей: от студентов, изучающих компьютерные технологии, до опытных разработчиков, которые хотят освоить современные концепции многопроцессорного программирования.

Тема: Архитектура компьютера / **Уровень пользователя:** опытный

ПИТЕР

Заказ книг:
197198, Санкт-Петербург, а/я 127, тел.: (812) 703-73-74, postbook@piter.com
61093, Харьков 93, а/я 9130, тел.: (057) 758-41-45, 751-10-02, piter@kharkov.piter.com

www.piter.com — вся информация о книгах и веб-магазин

ISBN 978-5-459-00291-1