Данное издание является дополнением к книге «Цифровая схемотехника и архитектура компьютера» с описанием отличий архитектуры ARM от MIPS, описанной в первой книге. Оно состоит из глав, посвященных архитектуре процессоров ARM, их микроархитектуре, описанию подсистемы памяти и системы ввода-вывода. Также в приложении приведена система команд ARM. Книгу рекомендуется использовать совместно с первым (основным) изданием по архитектуре MIPS.

Издание будет полезно студентам, инженерам, а также широкому кругу читателей, интересующихся современной схемотехникой.

Особенности книги:

- излагаются основы цифровой схемотехники с упором на логические концепции проектирования микропроцессора ARM;
- приводятся параллельные примеры на двух наиболее распространенных языках описания оборудования — SystemVerilog и VHDL, иллюстрирующие их применение при проектировании цифровых систем;
- по всему тексту разбросаны примеры, помогающие читателю лучше усвоить и запомнить важные концепции и технические приемы;
- на сопроводительном сайте имеются слайды к лекциям, лабораторные практикумы, решения упражнений, а также HDL-файлы для процессора ARM.

ISBN 978-5-97060-650-6

ДМК ПРЕСС
ИЗДАТЕЛЬСТВО
www.дмк.рф

Интернет-магазин www.dmkpress.com
Книга-почтой: orders@alians-kniga.ru
Оптовая продажа: «Альянс-книга»
(499)782-3889, books@alians-kniga.ru

Цифровая схемотехника и архитектура компьютера ARM

Цифровая схемотехника и архитектура компьютера

Дополнение по архитектуре ARM

Дэвид М. Харрис
Сара Л. Харрис

ELSEVIER
ДМК ПРЕСС
ИЗДАТЕЛЬСТВО

Дэвид М. Харрис
Сара Л. Харрис

ЦИФРОВАЯ СХЕМОТЕХНИКА И АРХИТЕКТУРА КОМПЬЮТЕРА

Дополнение по архитектуре ARM

Digital Design and Computer Architecture

ARM® Edition

David Money Harris
Sarah L. Harris

AMSTERDAM • BOSTON • HEIDELBERG • LONDON
NEW YORK • OXFORD • PARIS • SAN DIEGO
SAN FRANCISCO • SINGAPORE • SYDNEY • TOKYO
Morgan Kaufmann is an imprint of Elsevier

Цифровая схемотехника и архитектура компьютера

Дополнение по архитектуре ARM

Дэвид М. Харрис
Сара Л. Харрис

ДМК Пресс
издательство
Москва, 2019

УДК 004.2+744.4
ББК 32.971.3
Х20

Дэвид М. Харрис, Сара Л. Харрис
Х20 Цифровая схемотехника и архитектура компьютера. Дополнение по архитектуре ARM / пер. с англ. Слинкин А. А. / науч. ред. Косолобов Д. А. – М.: ДМК Пресс, 2019. – 356 с.: ил.

ISBN 978-5-97060-650-6

Данное издание является дополнением к книге «Цифровая схемотехника и архитектура компьютера» с описанием отличий архитектуры ARM от MIPS, описанной в первой книге. Оно состоит из глав, посвященных архитектуре процессоров ARM, их микроархитектуре, описанию подсистемы памяти и системы ввода-вывода. Также в приложении приведена система команд ARM. Книгу рекомендуется использовать совместно с первым (основным) изданием по архитектуре MIPS.

Издание будет полезно студентам, инженерам, а также широкому кругу читателей, интересующихся современной схемотехникой.

ISBN 978-0-12-800056-4 (анг.)
ISBN 978-5-97060-650-6 (рус.)

Нашим семьям

Оглавление

Похвальные отзывы на книгу «Цифровая схемотехника и архитектура компьютера». Дополнение по архитектуре ARM

Харрис и Харрис проделали замечательную похвальную работу по созданию действительно стоящего учебника, ясно показывающего их любовь и страсть к преподаванию и образованию. Студенты, прочитавшие эту книгу, будут благодарны Харрису и Харрис многие годы после окончания обучения. Стиль изложения, ясность, подробные диаграммы, поток информации, постепенное повышение сложности предмета, великолепные примеры по всем главам, упражнения в конце глав, краткие, но понятные объяснения, полезные примеры из реального мира, покрытие всех аспектов каждой темы – все эти вещи проделаны очень хорошо. Если вы студент, пользующийся этой книгой для подготовки к своему курсу, приготовьтесь получать удовольствие, поражаться, а также многому обучаться!

Мехди Хатамиан, старший вице-президент Broadcom

Харрис и Харрис проделали превосходную работу по созданию ARM версии своей популярной книги «Цифровая схемотехника и архитектура компьютера». Переориентация на ARM – это сложная задача, но авторы успешно справились с ней, при этом оставив свой ясный и тщательный стиль изложения, а также выдающееся качество включенной в текст документации. Я полагаю, что это новое издание будет очень хорошо принято как студентами, так и профессионалами.

Дональд Хунг, государственный университет Сан-Хосе

Из всех учебников, что я рецензировал и рекомендовал за 10 лет профессорства, «Цифровая схемотехника и архитектура компьютера» является одним из всего лишь двух, которые безусловно стоит купить (другой такой учебник – «Архитектура компьютера и проектирование компьютерных систем»). Изложение ясное и краткое, диаграммы просты для понимания, а процессор, который авторы используют в качестве рабочего примера, достаточно сложен, чтобы быть реалистичным, но достаточно прост, чтобы быть полностью понятным для моих студентов.

Захари Курмас, государственный университет Гранд Вэлли

Книга дает свежий взгляд на старую дисциплину. Многие учебники напоминают неухоженные заросли кустарника, но авторы данного учебника сумели отстричь засохшие ветви, сохранив основы и представив их в современном контексте. Эта книга поможет студентам справиться с техническими испытаниями завтрашнего дня.

Джим Френзел, Университет Айдахо

Книга написана в информативном, приятном для чтения стиле. Материал представлен на хорошем уровне для введения в проектирование компьютеров и содержит множество полезных диаграмм. Комбинационные схемы, микроархитектура и системы памяти изложены особенно хорошо.

Джеймс Пинтер-Люк, Колледж им. Дональда Маккенны, Клермонт

Харрис и Харрис написали очень ясную и легкую для понимания книгу. Упражнения хорошо разработаны, а примеры из реальной практики являются замечательным дополнением. Длинные и вводящие в заблуждение объяснения, часто встречающиеся в подобных книгах, здесь отсутствуют. Очевидно, что авторы посвятили много времени и усилий созданию доступного текста. Я настоятельно рекомендую книгу.

Пейи Чжао, Университет Чепмена

Предисловие

Эта книга уникальна тем, что описывает процесс проектирования цифровых систем с точки зрения компьютерной архитектуры, начиная от единиц и нулей и заканчивая разработкой микропроцессора.

Мы считаем, что построение микропроцессора – это особый обряд посвящения для студентов инженерных и компьютерных специальностей. Внутренняя работа процессора кажется почти волшебной для непосвященных, но после подробного объяснения оказывается простой для понимания. Проектирование цифровых систем – само по себе мощный и захватывающий предмет. Программирование на языке ассемблера позволяет увидеть внутренний язык, на котором говорит процессор. Микроархитектура является тем самым звеном, которое связывает эти части воедино.

Первые два издания этой все более набирающей популярность книги описывали архитектуру MIPS, следуя традиции широко распространенных книг по архитектуре Паттерсона и Хеннесси. Будучи одной из первых архитектур с сокращенным набором команд (RISC), MIPS опрятна и исключительно проста для понимания и разработки. MIPS остается важной архитектурой и получила приток свежих сил, после того как Imagination Technologies приобрела ее в 2013 году.

За последние два десятилетия архитектура ARM испытала взрыв популярности, причина которого – в ее эффективности и богатой экосистеме. Было произведено более 50 миллиардов процессоров ARM, и более 75% людей на планете пользуются продуктами с процессорами ARM. На момент написания данного текста почти каждый проданный сотовый телефон и планшет содержал один или несколько процессоров ARM. По прогнозам десятки миллиардов ARM систем вскоре будут контролировать интернет вещей (Internet of Things). Многие компании разрабатывают высокопроизводительные ARM-системы, чтобы бросить вызов Intel на рынке серверов. По причине такой коммерческой важности и интереса студентов мы написали данное ARM издание книги.

С педагогической точки зрения цели изданий MIPS и ARM одни и те же. Архитектура ARM имеет ряд особенностей, таких как режимы адресации и условное выполнение, которые вносят ощутимый вклад в эффективность, но при этом добавляют совсем немного сложности. К тому же эти микроархитектуры очень похожи, а условное выполнение и счетчик команд являются их самыми большими различиями. Глава о вводе-выводе содержит множество примеров, использующих Raspberry Pi – популярный одноплатный компьютер на основе ARM с Linux.

Мы рассчитываем предоставлять как MIPS-, так и ARM-издания до тех пор, пока рынок требует этого.

Особенности книги

Эта книга содержит ряд особенностей. В книге ссылки на главы основной книги «Цифровая схемотехника и архитектура компьютера» помечены.

Одновременное использование языков SystemVerilog и VHDL

Языки описания аппаратуры (hardware description languages, HDL) находятся в центре современных методов проектирования сложных цифровых систем. К сожалению, разработчики делятся на две примерно равные группы, использующие два разных языка – SystemVerilog и VHDL. Языки описания аппаратуры рассматриваются в **главе 4** (книга 1), сразу после глав, посвященных проектированию комбинационных и последовательных логических схем. Затем языки HDL используются в **главах 5 и 7** (книга 1) для разработки цифровых блоков большего размера и процессора целиком. Тем не менее **главу 4** (книга 1) можно безболезненно пропустить, если изучение языков HDL не входит в программу.

Эта книга уникальна тем, что использует одновременно и SystemVerilog, и VHDL, что позволяет читателю освоить проектирование цифровых систем сразу на двух языках. В **главе 4** (книга 1) сначала описываются общие принципы, применимые к обоим языкам, а затем вводится синтаксис и приводятся примеры использования этих языков. Этот двуязычный подход облегчает преподавателю выбор языка HDL, а читателю позволит перейти с одного языка на другой как во время учебы, так и в профессиональной деятельности.

Архитектура и микроархитектура классического процессора ARM

Главы 1 и 2 содержат первый всесторонний обзор архитектуры и микроархитектуры ARM. ARM – идеально подходящая для изучения архитектура, поскольку она является реальной архитектурой, поставляемой в составе миллионов продуктов ежегодно, но, несмотря на это, она рациональна и проста в освоении. Более того, ввиду популярности в коммерческом и любительском мирах существует немало средств разработки и эмуляции для архитектуры ARM. Все материалы, связанные с технологией ARM®, воспроизводятся с разрешения ARM Limited.

Живые примеры

В дополнение к живым примерам, обсуждаемым в связи с архитектурой ARM, в **главе 1** в качестве стороннего примера рассматривается архитектура процессоров Intel x86. **Глава 4** (доступная также в качестве онлайн-дополнения) описывает периферийные устройства в контексте одноплатного компьютера Raspberry Pi – весьма популярной платформы на базе ARM. Эти живые примеры показывают, как описанные в данных главах концепции применяются в реальных микросхемах, которые широко используются в персональных компьютерах и бытовой электронике.

Доступное описание высокопроизводительных архитектур

Глава 2 содержит краткий обзор современных высокопроизводительных микроархитектур: с предсказанием переходов, суперскалярной, с внеочередным выполнением команд, многопоточной и многоядерной. Материал изложен в доступной для первокурсников форме и показывает, как можно расширить микроархитектуры, описанные в книге, чтобы получить современный процессор.

Упражнения в конце глав и вопросы для собеседования

Лучшим способом изучения цифровой схемотехники является разработка устройств. В конце каждой главы приведены многочисленные упражнения. За упражнениями следует набор вопросов для собеседования, которые наши коллеги обычно задают студентам, претендующим на работу в отрасли. Эти вопросы предлагают читателю взглянуть на задачи, с которыми соискателям придется столкнуться в ходе собеседования при трудоустройстве. Решения упражнений доступны через веб-сайт книги и специальный веб-сайт для преподавателей.

Материалы в Интернете

Дополнительные материалы для этой книги доступны на веб-сайте по адресу http://booksite.elsevier.com/9780128000564. Этот веб-сайт доступен всем читателям и содержит:

- Решения нечетных упражнений.
- Ссылки на профессиональные средства автоматизированного проектирования (САПР) компании Altera®.
- Ссылку на Kiel ARM Microcontroller Development Kit (MDK-ARM) – инструменты для компиляции, ассемблирования и эмуляции Си и ассемблерного кода для процессоров ARM.
- HDL-код процессора ARM.
- Полезные советы по использованию САПР Altera Quartus II.
- Слайды лекций в формате PowerPoint.
- Образцы учебных и лабораторных материалов для курса.
- Список опечаток.

Также существует специальный веб-сайт для преподавателей, зарегистрировавшихся на http://booksite.elsevier.com/9780128000564, который содержит:

- Решения всех упражнений.
- Ссылки на профессиональные средства автоматизированного проектирования (САПР) компании Altera®.
- Рисунки из текста в форматах JPG и PPT.

Также на данном веб-сайте приведена дополнительная информация по использованию инструментов Altera, Raspberry Pi и MDK-ARM в вашем курсе. Там же находится информация о материалах для лабораторных работ.

Как использовать программный инструментарий в учебном курсе

Altera Quartus II

Quartus II Web Edition является бесплатной версией профессиональной САПР Quartus™ II, предназначенной для разработки на ПЛИС (FPGA). Она позволяет сту-

дентам проектировать цифровые устройства в виде принципиальных схем или на языках SystemVerilog и VHDL. После создания схемы или кода устройства студенты могут симулировать их поведение с использованием САПР ModelSim™ – Altera Starter Edition, которая доступна вместе с Altera Quartus II Web Edition. Quartus II Web Edition также включает в себя встроенный логический синтезатор, поддерживающий как SystemVerilog, так и VHDL.

Разница между Web Edition и Subscription Edition заключается в том, что Web Edition поддерживает только подмножество наиболее распространенных ПЛИС производства Altera. Разница между ModelSim – Altera Starter Edition и коммерческими версиями ModelSim заключается в том, что Starter Edition искусственно снижает производительность симуляции для проектов, содержащих больше 10 тысяч строк HDL-кода.

Kiel ARM Microcontroller Development Kit (MDK-ARM)

Kiel MDK-ARM – это инструмент для разработки кода для процессора ARM. Он доступен для скачивания онлайн бесплатно. MDK-ARM включает в себя коммерческий компилятор Си для ARM и эмулятор, который позволяет студентам писать программы на языке Си и ассемблере, компилировать их и затем эмулировать.

Лабораторные работы

Веб-сайт книги содержит ссылки на ряд лабораторных работ, которые охватывают все темы, начиная от проектирования цифровых систем и заканчивая архитектурой компьютера. Из лабораторных работ студенты узнают, как использовать САПР Quartus II для описания своих проектов, их симулирования, синтеза и реализации. Лабораторные работы также включают темы по программированию на языке Си и языке ассемблера с использованием средств разработки MDK-ARM и Raspberry Pi.

После синтеза студенты могут реализовать свои проекты, используя обучающие платы Altera DE2 (или DE2-115). Эта мощная и относительно недорогая плата доступна для заказа на веб-сайте www.altera.com. Плата содержит микросхему ПЛИС (FPGA), которую можно сконфигурировать для реализации студенческих проектов. Мы предоставляем лабораторные работы, которые описывают, как реализовать различные блоки на плате DE2 с использованием Quartus II Web Edition.

Для выполнения лабораторных работ студенты должны будут загрузить и установить САПР Altera Quartus II Web Edition и либо MDK-ARM, либо инструменты Raspberry Pi. Преподаватели могут также установить эти САПР в учебных лабораториях. Лабораторные работы включают инструкции по разработке проектов на плате DE2. Этап практической реализации проекта на плате можно пропустить, однако мы считаем, что он имеет большое значение для получения практических навыков.

Мы протестировали лабораторные работы на ОС Windows, но инструменты доступны и для ОС Linux.

Опечатки

Все опытные программисты знают, что любая сложная программа непременно содержит ошибки. Так же происходит и с книгами. Мы старались выявить и исправить все ошибки и опечатки в этой книге. Тем не менее некоторые ошибки могли остаться. Список найденных ошибок будет опубликован на веб-сайте книги.

Пожалуйста, присылайте найденные ошибки по адресу ddcabugs@gmail.com[1]. Первый человек, который сообщит об ошибке и предоставит исправление, которое мы используем в будущем издании, будет вознагражден премией в $1!

Признательность за поддержку

Мы ценим тяжелую работу Нэйта МакФаддена (Nate McFadden), Джо Хэйтона (Joe Hayton), Пунитавати Говиндараджана (Punithavathy Govindaradjane) и остальных членов команды издательства Morgan Kaufmann, которые сделали возможным появление этой книги. Нам нравится художественная работа Дуэйна Бибби (Duane Bibby), чьи иллюстрации вдохнули жизнь в главы.

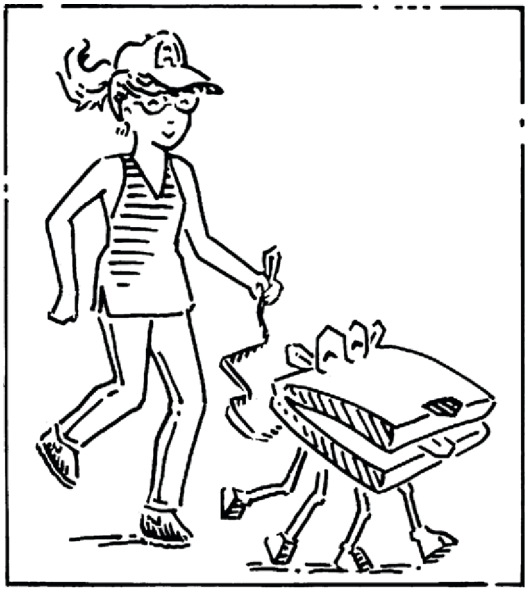
Мы хотели бы поблагодарить Мэтью Уоткинса (Matthew Watkins), который помог написать раздел о гетерогенных многопроцессорных системах в **главе 2**. Мы очень ценим работу Джошуа Васкеза (Joshua Vasquez), разработавшего код для Raspberry Pi из **главы 4**. Мы также благодарны Джозефу Спьюту (Josef Spjut) и Руйе Вангу (Ruye Wang), протестировавшим материал в классе.

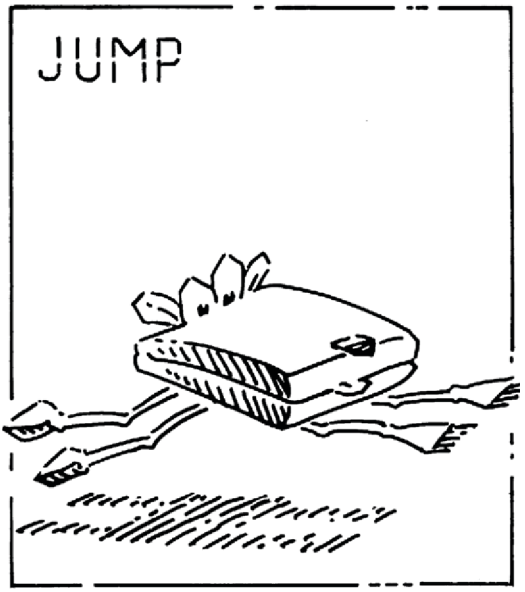
Огромный вклад в улучшение качества книги внесли многочисленные рецензенты, среди которых Бойанг Ванг (Boyang Wang), Джон Барр (John Barr), Джэк Брайнер (Jack V. Briner), Эндрю Браун (Andrew C. Brown), Карл Баумгартнер (Carl Baumgaertner), Утку Дирил (A. Utku Diril), Джим Френцель (Jim Frenzel), Джаэха Ким (Jaeha Kim), Филлип Кинг (Phillip King), Джеймс Пинтер-Лаки (James Pinter-Lucke), Амир Рот (Amir Roth), Джерри Ши (Z. Jerry Shi), Джеймс Стайн (James E. Stine), Люк Тэсье (Luke Teyssier), Пейуй Чжао (Peiyi Zhao), Зак Доддс (Zach Dodds), Натаниэл Гай (Nathaniel Guy), Эшвин Кришна (Aswin Krishna), Волней Педрони (Volnei Pedroni), Карл Ванг (Karl Wang), Рикардо Яасински (Ricardo Jasinski), Джозеф Спьют (Josef Spjut), Йорген Лиен (Jörgen Lien), Самеер Шарма (Sameer Sharma), Джон Нестор (John Nestor), Сайев Манзоор (Syed Manzoor), Джеймс Хо (James Hoe), Сриниваза Вемуру (Srinivasa Vemuru), Джозеф Хасс (K. Joseph Hass), Джафанта Херат (Jayantha Herath), Роберт Муллинс (Robert Mullins), Бруно Куоитин (Bruno Quoitin), Субраманьям Ганеша (Subramaniam Ganesan), Браден Филлипс (Braden Phillips), Джон Оливер (John Oliver), Яхсвант Малайа (Yahswant K. Malaiya), Мохаммад Аведх (Mohammad Awedh), Захари Курмас (Zachary Kurmas), Дональд Хунг (Donald Hung) и анонимный рецензент. Мы очень признательны Кхаледу Бенкриду (Khaled Benkrid) и его коллегам из ARM за тщательное рецензирование материалов, связанных с ARM.

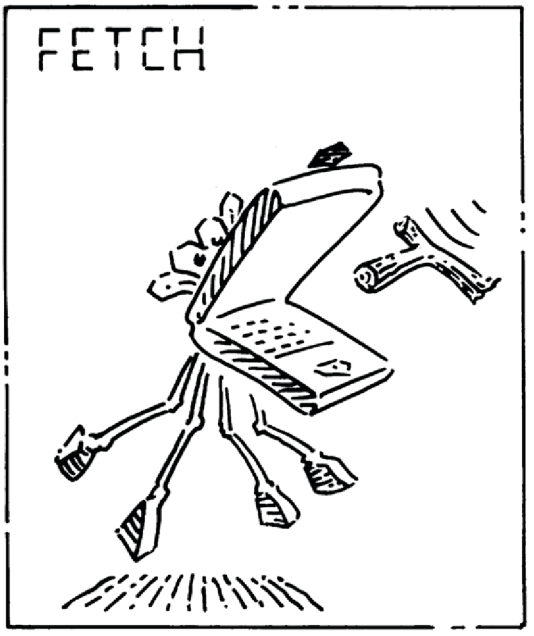
[1] Это относится лишь к ошибкам в исходном англоязычном издании. Ошибки в переводе присылайте на адрес dmkpress@gmail.com (хотя в этом случае издательство доллар не выдает). – *Прим. перевод.*

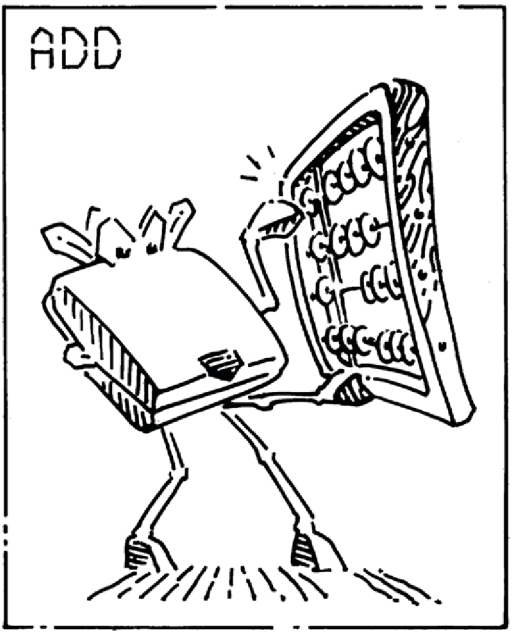
Мы также признательны нашим студентам из колледжа Harvey Mudd и UNLV, которые дали полезные отзывы на черновики этого учебника. Отдельного упоминания заслуживают Клинтон Барнс (Clinton Barnes), Мэтт Вайнер (Matt Weiner), Карл Уолш (Carl Walsh), Эндрю Картер (Andrew Carter), Кейси Шиллинг (Casey Schilling), Элис Клифтон(Alice Clifton), Крис Эйкон (Chris Acon) и Стивен Браунер (Stephen Brawner).

И, конечно же, мы благодарим наши семьи за их любовь и поддержку.



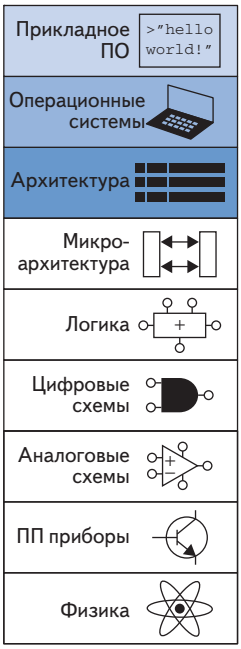

JUMP


FETCH


ADD

Глава 1

Архитектура



1.1. Введение

В предыдущих главах мы познакомились с принципами разработки цифровых устройств и основными строительными блоками. В этой главе мы поднимемся на несколько уровней абстракции и определим *архитектуру* компьютера. Архитектура – это то, как видит компьютер программист. Она определяется набором команд (языком) и местом нахождения операндов (регистры или память). Существует множество разных архитектур, например: x86, MIPS, SPARC и PowerPC.

Чтобы понять архитектуру любого компьютера, нужно в первую очередь выучить его язык. Слова в языке компьютера называются *командами*, а словарный запас компьютера – *набором*, или *системой команд*.

Даже сложные приложения – редакторы текста и электронные таблицы – в конечном итоге состоят из последовательности таких простых команд, как сложение, вычитание и переход. Команда компьютера определяет операцию, которую нужно исполнить, и ее операнды. Операнды могут находиться в памяти, в регистрах или внутри самой команды.

Говоря об «архитектуре ARM», мы имеем в виду ARM версии 4 (ARMv4) и составляющий ее базовый набор команд. В **разделе 1.7** дан краткий обзор возможностей, появившихся в версиях 5–8 этой архитектуры. В сети можно найти «Справочное руководство по архитектуре ARM» (ARM Architecture Reference Manual (ARM)), в котором содержится полное определение архитектуры.

Аппаратное обеспечение компьютера «понимает» только нули и единицы, поэтому команды закодированы двоичными числами в формате, который называется *машинным языком*. Так же как мы используем буквы и прочие символы на письме для представления речи, компьютеры используют двоичные числа, чтобы кодировать машинный язык. В архитектуре ARM каждая команда представлена 32-разрядным словом. Микропроцессоры – это цифровые системы, которые читают и выполняют команды машинного языка. Но для людей чтение компьютерных программ на машинном языке представляется нудным и утомительным занятием, поэтому мы предпочитаем представлять команды в символическом формате, который называется *языком ассемблера*.

Наборы команд в различных архитектурах можно сравнить с диалектами естественных языков. Почти во всех архитектурах определены такие базовые команды, как сложение, вычитание и переход, работающие с ячейками памяти или регистрами. Изучив один набор команд, понять другие уже довольно легко.

Архитектура компьютера не определяет структуру аппаратного обеспечения, которое ее реализует. Зачастую существуют разные аппаратные реализации одной и той же архитектуры. Например, компании Intel и Advanced Micro Devices (AMD) производят разные микропроцессоры, построенные на базе архитектуры x86. Все они могут выполнять одни и те же программы, но в их основе лежит разное аппаратное обеспечение, поэтому эти процессоры характеризуются различным соотношением производительности, цены и энергопотребления. Одни микропроцессоры оптимизированы для работы в высокопроизводительных серверах, другие рассчитаны на продление срока службы батареи в ноутбуках. Конкретное сочетание регистров, памяти, АЛУ и других строительных блоков, из которых состоит микропроцессор, называют *микроархитектурой*, она будет рассмотрена в **главе 2**. Нередко для одной и той же архитектуры существует несколько разных микроархитектур.

В этой книге мы представим архитектуру ARM. Впервые она была разработана в 1980-х годах компанией Acorn Computer Group, от которой затем отпочковалась компания Advanced RISC Machines Ltd., известная ныне под названием ARM. Ежегодно продается свыше 10 млрд процессоров ARM. Почти все сотовые телефоны и планшеты оснащены несколькими процессорами ARM. Эта архитектура встречается повсеместно: в автоматах для игры в пинбол, в фотокамерах, в роботах, в серверах, смонтированных в стойке. Компания ARM необычна тем, что продает не сами процессоры, а лицензии, разрешающие другим компаниям самостоятельно производить процессоры, которые зачастую являются составной частью более крупной системы на кристалле. Например,

процессоры ARM изготавливают компании Samsung, Altera, Apple и Qualcomm – они построены на базе либо микроархитектуры, приобретенной у ARM, либо собственной микроархитектуры, разработанной по лицензии ARM. Мы остановились на архитектуре ARM, потому что она занимает лидирующие коммерческие позиции и в то же время является чистой и почти свободной от странностей. Начнем с описания команд языка ассемблера, мест нахождения операндов и таких общеупотребительных программных конструкций, как ветвления, циклы, операции с массивами и вызовы функций. Затем опишем, как язык ассемблера транслируется в машинный язык, и продемонстрируем, как программа загружается в память и выполняется.

В этой главе мы покажем, как архитектура ARM формировалась на основе четырех принципов, сформулированных Дэвидом Паттерсоном и Джоном Хеннесси в книге «Computer Organization and Design»:

1) единообразие способствует простоте;
2) типичный сценарий должен быть быстрым;
3) чем меньше, тем быстрее;
4) хороший проект требует хороших компромиссов.

1.2. Язык ассемблера

Язык ассемблера – это удобное для восприятия человеком представление внутреннего языка компьютера. Каждая команда языка ассемблера задает операцию, которую необходимо выполнить, а также операнды, над которыми производится эта операция. Далее мы познакомимся с простыми арифметическими командами и покажем, как они записываются на языке ассемблера. Затем определим операнды для команд ARM: регистры, ячейки памяти и константы.

В этой главе предполагается знакомство с каким-нибудь высокоуровневым языком программирования, например C, C++ или Java (эти языки практически равнозначны для большинства примеров в данной главе, но мы будем использовать C). В **приложении C** (книга 1) приведено введение в язык C для тех, у кого мало или совсем нет опыта программирования.

В этой главе для компиляции, ассемблирования и эмуляции ассемблерного кода мы пользуемся комплектом средств разработки ARM Microcontroller Development Kit (MDK-ARM) компании Keil. MDK-ARM – свободный инструмент разработки, в состав которого входит полный компилятор для ARM. В лабораторных работах на сопроводительном сайте для этой книги (см. **предисловие**) описано, как установить и использовать этот инструмент для написания, компиляции, эмуляции и отладки программ на C и ассемблере.

1.2.1. Команды

Наиболее частая операция, выполняемая компьютером, – сложение. В **примере кода 1.1** показан код, который складывает переменные `b` и `c` и записывает результат в переменную `a`. Слева показан вариант на языке высокого уровня (C, C++ или Java), а справа – на языке ассемблера

ARM. Обратите внимание, что предложения языка C оканчиваются точкой с запятой.

Пример кода 1.1. СЛОЖЕНИЕ

Код на языке высокого уровня	Код на языке ассемблера ARM
`a = b + c;`	`ADD a, b, c`

Слово «мнемоника» происходит от греческого слова *μνημονικός*. Мнемоники языка ассемблера запомнить проще, чем наборы нулей и единиц машинного языка, представляющих ту же операцию.

Первая часть команды ассемблера, `ADD`, называется *мнемоникой* и определяет, какую операцию нужно выполнить. Операция осуществляется над *операндами-источниками* `b` и `c`, а результат записывается в *операнд-приемник* `a`.

Пример кода 1.2. ВЫЧИТАНИЕ

Код на языке высокого уровня	Код на языке ассемблера ARM
`a = b - c;`	`SUB a, b, c`

В **примере кода 1.2** демонстрируется, что вычитание похоже на сложение. Формат команды такой же, как у команды `ADD`, только операция называется `SUB`. Единообразный формат команд — пример применения первого принципа хорошего проектирования:

> **Первый принцип хорошего проектирования:** единообразие способствует простоте.

Команды с одинаковым количеством операндов — в данном случае с двумя операндами-источниками и одним операндом-приемником — проще закодировать и выполнить на аппаратном уровне. Более сложный высокоуровневый код транслируется в несколько команд ARM, как показано в **примере кода 1.3**.

Пример кода 1.3. БОЛЕЕ СЛОЖНЫЙ КОД

Код на языке высокого уровня	Код на языке ассемблера ARM

`a = b + c - d; // однострочный комментарий`
`/* многострочный`
`комментарий */`
	`ADD t, b, c ; t = b + c`
	`SUB a, t, d ; a = t - d`

В примерах на языках высокого уровня однострочные комментарии начинаются с символов `//` и продолжаются до конца строки. Много-

строчные комментарии начинаются с `/*` и завершаются `*/`. В языке ассемблера ARM допустимы только однострочные комментарии. Они начинаются знаком `;` и продолжаются до конца строки. В программе на языке ассемблера в **примере 1.3** используется временная переменная `t` для хранения промежуточного результата. Использование нескольких команд ассемблера для выполнения более сложных операций – иллюстрация второго принципа хорошего проектирования компьютерной архитектуры:

> **Второй принцип хорошего проектирования:** типичный сценарий должен быть быстрым.

При использовании системы команд ARM типичный сценарий оказывается быстрым, потому что включает только простые, часто используемые команды. Количество команд специально оставляют небольшим, чтобы аппаратура для их поддержки была простой и быстрой. Более сложные операции, используемые реже, представляются в виде последовательности нескольких простых команд. По этой причине ARM относится к компьютерным архитектурам с *сокращенным набором команд* (reduced instruction set computer, RISC). Архитектуры с большим количеством сложных инструкций, такие как архитектура x86 корпорации Intel, называются *компьютерами со сложным набором команд* (complex instruction set computer, CISC). Например, в x86 определена команда «перемещение строки», которая копирует строку (последовательность символов) из одного участка памяти в другой. Такая операция требует большого количества, иногда до нескольких сотен, простых команд на RISC-машине. С другой стороны, реализация сложных команд в архитектуре CISC требует дополнительного оборудования и увеличивает накладные расходы, что приводит к замедлению простых команд.

В архитектуре RISC используется небольшое множество различных команд, что уменьшает сложность аппаратного обеспечения и размер команды. Например, код операции в системе, состоящей из 64 простых команд, требует $\log_2 64 = 6$ бит, а в системе из 256 сложных команд потребуется уже $\log_2 256 = 8$ бит. В CISC-машинах наличие сложных команд, даже очень редко используемых, увеличивает накладные расходы на выполнение всех команд, включая и самые простые.

1.2.2. Операнды: регистры, память и константы

Команда работает с операндами. В **примере кода 1.1** переменные `a`, `b` и `c` являются операндами. Но компьютеры оперируют нулями и единицами, а не именами переменных. Команда должна знать, откуда она сможет брать двоичные данные. Операнды могут находиться в регистрах или в памяти, а также могут быть *константами*, записанными в теле самой

команды. Для хранения операндов используются различные места, чтобы повысить скорость исполнения и (или) более эффективно разместить данные. Обращение к операндам-константам или операндам, находящимся в регистрах, происходит быстро, но так можно разместить лишь небольшое количество данных. Остальные данные хранятся в емкой, но медленной памяти. Архитектуру ARM (до версии ARMv8) называют 32-битовой потому, что она оперирует 32-битовыми данными.

В версии 8 архитектура ARM расширена до 64 бит, но в этой книге мы будем рассматривать только 32-битовую версию.

Регистры

Чтобы команды выполнялись быстро, они должны быстро получать доступ к операндам. Но чтение операндов из памяти занимает много времени, поэтому в большинстве архитектур имеется небольшое количество регистров для хранения наиболее часто используемых операндов. В архитектуре ARM используется 32 регистра, которые называют *набором регистров*, или *регистровым файлом*. Чем меньше регистров, тем быстрее к ним доступ. Это приводит нас к третьему принципу хорошего проектирования компьютерной архитектуры:

> **Третий принцип хорошего проектирования:** чем меньше, тем быстрее.

Найти информацию гораздо быстрее в нескольких тематически подобранных книгах, лежащих на столе, чем в уйме книг на полках в библиотеке. Точно так же прочитать данные из небольшого набора регистров быстрее, чем из большой памяти. Небольшие регистровые файлы обычно состоят из маленького массива статической памяти SRAM (см. **раздел 5.5.3** (книга 1)).

В **примере кода 1.4** показана команда `ADD` с регистровыми операндами. Имена регистров ARM начинаются буквой `'R'`. Переменные `a`, `b` и `c` произвольно размещены в регистрах `R0`, `R1` и `R2`. Имя `R1` произносят как «регистр 1», или «R1», «регистр R1». Команда складывает 32-битовые значения, хранящиеся в R1 `(b)` и R2 `(c)`, и записывает 32-битовый результат в R0 `(a)`. В **примере кода 1.5** показан ассемблерный код ARM, в котором для хранения промежуточного результата вычисления `b + c` используется регистр R4:

Пример кода 1.4. РЕГИСТРОВЫЕ ОПЕРАНДЫ

Код на языке высокого уровня

```
a = b + c
```

Код на языке ассемблера ARM

```
; R0 = a, R1 = b, R2 = c
  ADD R0, R1, R2        ; a = b + c
```

Пример кода 1.5. ВРЕМЕННЫЕ РЕГИСТРЫ

Код на языке высокого уровня	Код на языке ассемблера ARM

`a = b + c - d;`	`; R0 = a, R1 = b, R2 = c, R3 = d; R4 = t`
	`ADD R4, R1, R2 ; t = b + c`
	`SUB R0, R4, R3 ; a = t - d`

Пример 1.1. ТРАНСЛЯЦИЯ КОДА С ЯЗЫКА ВЫСОКОГО УРОВНЯ НА ЯЗЫК АССЕМБЛЕРА

Транслируйте приведенный ниже код на языке высокого уровня в код на языке ассемблера. Считайте, что переменные `a`, `b` и `c` находятся в регистрах R0–R2, а переменные `f`, `g`, `h`, `i` и `j` – в регистрах R3–R7.

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

Решение: в программе используется четыре ассемблерные команды.

```
#; Код на языке ассемблера ARM
; R0 = a, R1 = b, R2 = c, R3 = f, R4 = g, R5 = h, R6 = i, R7 = j
  SUB R0, R1, R2      ; a = b - c
  ADD R8, R4, R5      ; R8 = g + h
  ADD R9, R6, R7      ; R9 = i + j
  SUB R3, R8, R9      ; f = (g + h) - (i + j)
```

Набор регистров

В **Табл. 1.1** перечислены имена и роли каждого из 16 регистров ARM. Регистры R0–R12 служат для хранения переменных, регистры R0–R3 используются также особым образом при вызове процедур. Регистры R13–R15 называют еще SP, LR и PC, мы опишем их ниже в этой главе.

Таблица 1.1. Набор регистров ARM

Название	Назначение
R0	Аргумент, возвращаемое значение, временная переменная
R1–R3	Аргумент, временная переменная
R4–R11	Сохраненная переменная
R12	Временная переменная
R13 (SP)	Указатель стека
R14 (LR)	Регистр связи
R15 (PC)	Счетчик команд

Константы, или непосредственные операнды

Помимо операций с регистрами, в командах ARM можно использовать константы, или *непосредственные* операнды. Они называются так потому, что значение операнда является частью самой команды, никакого доступа к регистрам или к памяти не требуется. В **примере кода 1.6** показана команда `ADD`, прибавляющая непосредственный операнд к регистру. В ассемблерном коде непосредственному операнду предшествует знак `#`, а значение операнда можно записывать в десятичном или шестнадцатеричном виде. На языке ассемблера ARM шестнадцатеричные константы начинаются символами 0x, как в C. Непосредственные операнды являются 8- или 12-битовыми числами без знака, порядок их кодирования описан в **разделе 1.4**.

Пример кода 1.6. НЕПОСРЕДСТВЕННЫЕ ОПЕРАНДЫ

Код на языке высокого уровня

```
a = a + 4;
b = a - 12;
```

Код на языке ассемблера ARM

```
; R7 = a, R8 = b
  ADD R7, R7, #4    ; a = a + 4
  SUB R8, R7, #0xC  ; b = a - 12
```

Команда перемещения (`MOV`) – удобный способ инициализации регистров. В **примере кода 1.7** переменные `i` и `x` инициализированы значениями 0 и 4080 соответственно. Операндом-источником команды `MOV` может быть также регистр. Например, команда `MOV R1, R7` копирует содержимое регистра R7 в R1.

Пример кода 1.7. ИНИЦИАЛИЗАЦИЯ С ПОМОЩЬЮ НЕПОСРЕДСТВЕННЫХ ОПЕРАНДОВ

Код на языке высокого уровня

```
i = 0;
x = 4080;
```

Код на языке ассемблера ARM

```
; R4 = i, R5 = x
  MOV R4, #0      ; i = 0
  MOV R5, #0xFF0  ; x = 4080
```

Память

Если бы операнды хранились только в регистрах, то мы могли бы писать лишь простые программы, в которых не более 15 переменных. Однако данные можно хранить также в памяти. По сравнению с регистровым файлом, память располагает большим объемом для хранения данных, но доступ к ней занимает больше времени. По этой причине часто используемые переменные хранятся в регистрах. В архитектуре ARM команды работают только с регистрами, поэтому данные, хранящиеся в памяти,

до обработки следует переместить в регистры. Комбинируя память и регистры, программа может получать доступ к большим объемам данных достаточно быстро. Как было описано в **разделе 5.5** (книга 1), память устроена как массив слов. В архитектуре ARM используются 32-битовые адреса памяти и 32-битовые слова данных.

В ARM используется *побайтовая адресация* памяти. Это значит, что каждый байт памяти имеет уникальный адрес, как показано на **Рис. 1.1 (a)**. 32-битовое слово состоит из четырех 8-битовых байтов, поэтому адрес слова кратен 4. Старший байт слова занимает крайние левые биты, а младший байт – крайние правые. На **Рис. 1.1 (b)** 32-битовые адрес слова и его значение записаны в шестнадцатеричном виде. Так, слово 0xF2F1AC07 хранится в памяти по адресу 4. При графическом изображении памяти меньшие адреса традиционно размещают снизу, а большие – сверху.

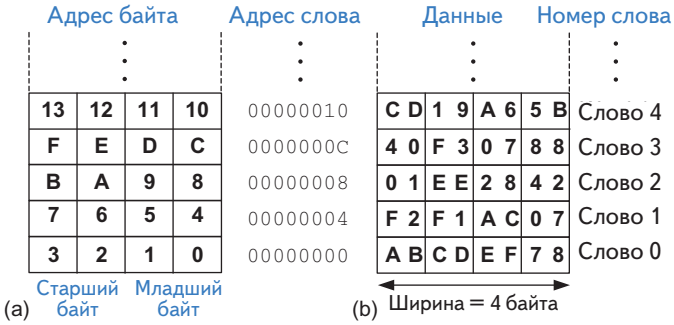


Рис. 1.1. Память в ARM с побайтовой адресацией: (a) адреса байтов, (b) данные

В ARM имеется команда *загрузить регистр* (load register), `LDR`, которая читает слово из памяти в регистр. В **примере кода 1.8** слово 2 загружается в переменную `a` (регистр R7). В языке C число внутри квадратных скобок называется *индексом*, или номером слова, мы вернемся к этому вопросу в **разделе 1.3.6**. Команда `LDR` задает адрес в памяти с помощью *базового регистра* (R5) и *смещения* (8). Напомним, что длина каждого слова равна 4 байтам, поэтому слово с номером 1 размещается по адресу 4, слово с номером 2 – по адресу 8 и т. д. Адрес слова в четыре раза больше его номера. Адрес в памяти образуется путем сложения значения базового регистра (R5) и смещения. В ARM предлагается несколько режимов доступа к памяти, они рассматриваются в **разделе 1.3.6**.

Чтение из ячейки памяти по базовому адресу (когда индекс равен 0) – специальный случай, в котором указывать смещение в ассемблерном коде не нужно. Например, для чтения из ячейки с базовым адресом, хранящимся в регистре R5, следует написать `LDR R3, [R5]`.

Пример кода 1.8. ЧТЕНИЕ ИЗ ПАМЯТИ

Код на языке высокого уровня	Код на языке ассемблера ARM

`a = mem[2];`	`; R7 = a`
	`MOV R5, #0 ; базовый адрес = 0`
	`LDR R7, [R5, #8] ; R7 <= данные по адресу (R5+8)`

В ARMv4 в командах `LDR` и `STR` адреса должны быть *выровнены на границу слова*, т. е. адрес слова должен делиться на 4. Начиная с версии ARMv6 это ограничение можно снять, установив бит в регистре управления системой, но производительность *невыровненных* операций загрузки гораздо ниже. В некоторых архитектурах, например x86, операции чтения и записи по адресу, не выровненному на границу слова, разрешены, тогда как в других, например MIPS, для упрощения оборудования требуется строгое выравнивание. Разумеется, адреса байтов в командах загрузки и сохранения байта, `LDRB` и `STRB` (см. раздел 1.3.6), не обязательно должны быть выровнены на границу слова.

После выполнения команды загрузки регистра (`LDR`) в **примере кода 1.8** регистр R7 будет содержать значение 0x01EE2842, т. е. данные, хранящиеся в памяти по адресу 8 на **Рис. 1.1**.

Для записи слова из регистра в память в ARM служит команда *сохранить регистр* (store register), `STR`. В **примере кода 1.9** значение 42 записывается из регистра R9 в слово памяти с номером 5.

Есть два способа организации памяти с побайтовой адресацией: с *прямым порядком* следования байтов (от младшего к старшему; little-endian) или с *обратным порядком* (от старшего к младшему; big-endian), как показано на **Рис. 1.2**. В обоих случаях *старший байт* (most significant byte, MSB) 32-битового слова находится слева, а *младший байт* (least significant byte, LSB) – справа. Адреса слов одинаковы в обеих моделях, то есть один и тот же адрес слова указывает на одни и те же четыре байта. Отличаются только *адреса байтов* внутри слова. В машинах с *прямым порядком следования* байты пронумерованы от 0, начиная с младшего байта, а в машинах с *обратным порядком следования* байты пронумерованы от 0, начиная со старшего байта.

Пример кода 1.9. ЗАПИСЬ В ПАМЯТЬ

Код на языке высокого уровня	Код на языке ассемблера ARM

`mem[5] = 42;`	`MOV R1, #0 ; базовый адрес = 0`
	`MOV R9, #42`
	`STR R9, [R1, #0x14] ; значение сохраняется в`
	`; памяти по адресу (R1+20) = 42`

В процессоре PowerPC компании IBM, который ранее использовался в компьютерах Apple Macintosh, порядок следования байтов обратный, а в архитектуре x86 компании Intel, применяемой в персональных компьютерах, – прямой. В ARM предпочтительным является прямой порядок, но в некоторых версиях поддерживается *переключаемая* (bi-endian) адре-

сация, при которой разрешено загружать и сохранять данные в любом формате. Выбор порядка следования байтов абсолютно произволен, но ведет к проблемам при обмене данными между компьютерами с разным порядком байтов. В этой книге мы будем использовать прямой порядок в тех случаях, когда это имеет значение.

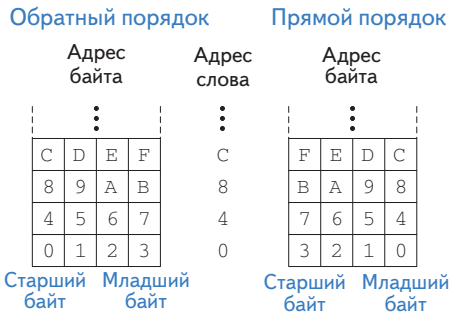


Рис. 1.2. Адресация данных с прямым и обратным порядком байтов

1.3. Программирование

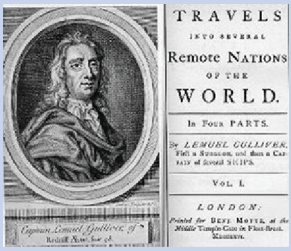
Языки типа C и Java называются языками программирования высокого уровня, поскольку они находятся на более высоком уровне абстракции, по сравнению с языком ассемблера. Во многих языках высокого уровня используются такие распространенные программные конструкции, как арифметические и логические операции, условное выполнение, предложения if/else, циклы for и while, индексируемые массивы и вызовы функций. Примеры таких конструкций для языка C см. в **приложении C** (книга 1). В этом разделе мы рассмотрим, как эти высокоуровневые конструкции транслируются на язык ассемблера ARM.

1.3.1. Команды обработки данных

В архитектуре ARM определен ряд команд *обработки данных* (в других архитектурах они часто называются логическими и арифметическими командами). Мы дадим их краткий обзор, потому что они необходимы для реализации конструкций более высокого уровня. В **приложении A** приведена сводка команд ARM.

Логические команды

К *логическим операциям* в ARM относятся команды `AND` (И), `ORR` (ИЛИ), `EOR` (ИСКЛЮЧАЮЩЕЕ ИЛИ) и `BIC` (сбросить бит). Это поразрядные операции над двумя



Происхождение английских терминов little-endian (прямой порядок) и big-endian (обратный порядок) восходит к произведению «Путешествия Гулливера» Джонатана Свифта, впервые опубликованному в 1726 году под псевдонимом Исаака Бикерстаффа. В его рассказах король лилипутов требовал от граждан (остроконечников, англ.: Little-Endians) разбивать яйцо с острого конца. Тупоконечники (англ.: Big-Endians) были повстанцами, разбивавшими яйца с тупого конца. Термины были впервые применены к компьютерным архитектурам Дэнни Коэном в его статье «О священных войнах и мольбе о мире», опубликованной в День дурака 1 апреля 1980 года (USC/ISI IEN 137).
(Фотография из коллекции Бразертона любезно предоставлена библиотекой Лидского университета.)

операндами-источниками, результат которых записывается в регистр-приемник. Первым источником всегда является регистр, а вторым может быть другой регистр или непосредственный операнд. Еще одна логическая операция, `MVN` (MoVe and Not – перемещение и отрицание), определена как поразрядное НЕ, применяемое ко второму источнику (непосредственному операнду или регистру) с последующей записью в регистр-приемник. На **Рис. 1.3** показано, как эти операции применяются к двум операндам-источникам 0x46A1F1B7 и 0xFFFF0000. Здесь же показано, какое значение оказывается в регистре-приемнике после выполнения команды.

Рис. 1.3. Логические операции

Регистры-источники

R1	0100 0110	1010 0001	1111 0001	1011 0111
R2	1111 1111	1111 1111	0000 0000	0000 0000

Ассемблерный код		Результат			
`AND R3, R1, R2`	R3	0100 0110	1010 0001	0000 0000	0000 0000
`ORR R4, R1, R2`	R4	1111 1111	1111 1111	1111 0001	1011 0111
`EOR R5, R1, R2`	R5	1011 1001	0101 1110	1111 0001	1011 0111
`BIC R6, R1, R2`	R6	0000 0000	0000 0000	1111 0001	1011 0111
`MVN R7, R2`	R7	0000 0000	0000 0000	1111 1111	1111 1111

Команда сброса бита (`BIC`) полезна для маскирования битов (сброса ненужных битов в 0). Команда `BIC R6, R1, R2` вычисляет `R1 AND NOT R2`. Иными словами, `BIC` сбрасывает биты, поднятые в R2. В данном случае два старших байта R1 очищаются, или *маскируются*, а два незамаскированных младших байта R1, 0xF1B7, помещаются в R6. Замаскировать можно любое подмножество бит регистра.

Команда `ORR` полезна, когда нужно объединить битовые поля, хранящиеся в двух регистрах. Например, 0x347A0000 `ORR` 0x000072FC = 0x347A72FC.

Команды сдвига

Команды сдвига сдвигают значение, находящееся в регистре, влево или вправо, при этом вышедшие за границу биты отбрасываются. Существует также команда циклического сдвига вправо на N бит, где N меньше или равно 31. Мы будем употреблять для сдвига и циклического сдвига общее название – операции сдвига. В ARM имеются следующие команды сдвига: `LSL` (логический сдвиг влево), `LSR` (логический сдвиг вправо), `ASR` (арифметический сдвиг вправо) и `ROR` (циклический сдвиг вправо). Команды `ROL` не существует, потому что циклический сдвиг влево эквивалентен циклическому сдвигу вправо на дополнительное количество разрядов.

Как отмечалось в **разделе 5.2.5** (книга 1), при сдвиге влево младшие биты всегда заполняются нулями. Но сдвиг вправо может быть как логическим (в старшие биты «вдвигаются» нули), так и арифметическим (в старшие биты помещается знаковый бит). Величина сдвига может задаваться непосредственным операндом или регистром.

На **Рис. 1.4** показаны ассемблерный код и значения регистра-приемника для команд `LSL`, `LSR`, `ASR` и `ROR` при сдвиге на константное число бит. Значение в регистре R5 сдвигается, и результат помещается в регистр-приемник.

	Регистры-источники			
R5	1111 1111	0001 1100	0001 0000	1110 0111

Ассемблерный код		Результат			
`LSL R0, R5, #7`	R0	1000 1110	0000 1000	0111 0011	1000 0000
`LSR R1, R5, #17`	R1	0000 0000	0000 0000	0111 1111	1000 1110
`ASR R2, R5, #3`	R2	1111 1111	1110 0011	1000 0010	0001 1100
`ROR R3, R5, #21`	R3	1110 0000	1000 0111	0011 1111	1111 1000

Рис. 1.4. Операции сдвига на величину, заданную непосредственно

Сдвиг влево на N бит эквивалентен умножению на 2^N. Аналогично арифметический сдвиг вправо на N бит эквивалентен делению на 2^N, как было сказано в **разделе 5.2.5** (книга 1). Операции логического сдвига применяются также для выделения или объединения битовых полей.

На **Рис. 1.5** показаны ассемблерный код и значения регистра-приемника для команд сдвига в случае, когда величина сдвига хранится в регистре R6. В этой команде используется режим адресации *регистр – сдвиговый регистр*, когда один регистр (R8) сдвигается на величину (20), хранящуюся во втором регистре (R6).

	Регистры-источники			
R8	0000 1000	0001 1100	0001 0110	1110 0111
R6	0000 0000	0000 0000	0000 0000	0001 0100

Ассемблерный код		Результат			
`LSL R4, R8, R6`	R4	0110 1110	0111 0000	0000 0000	0000 0000
`ROR R5, R8, R6`	R5	1100 0001	0110 1110	0111 0000	1000 0001

Рис. 1.5. Операции сдвига на величину, заданную в регистре

Команды умножения

Умножение отличается от других арифметических операций тем, что при перемножении двух 32-битовых чисел получается 64-битовое число. В архитектуре ARM предусмотрены *команды умножения*, дающие 32- или 64-битовое произведение. Команда `MUL` перемножает два 32-битовых числа и дает 32-битовый результат. Команда `MUL R1, R2, R3` перемножает значения в регистрах R2 и R3 и помещает младшие биты

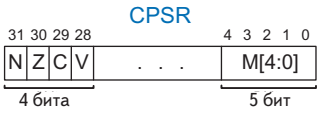


Рис. 1.6. Регистр текущего состояния программы (CPSR)

Младшие пять бит регистра CPSR называются битами *режима* и обсуждаются в **разделе 1.6.3**.

Для сравнения двух значений полезны также команды `CMN`, `TST` и `TEQ`. Каждая из них выполняет операцию, обновляет флаги условий и отбрасывает результат. Команда `CMN` (compare negative – сравнить с противоположным) сравнивает первый источник с величиной, противоположной второму, для чего складывает оба источника. В **разделе 1.4** мы увидим, что в командах ARM кодируются только положительные непосредственные операнды. Поэтому вместо `CMP R2, #-20` следует использовать `CMN R2, #-20`. Команда `TST` (test – проверить) вычисляет логическое И операндов-источников. Она полезна, когда нужно проверить, что некоторая часть регистра равна или, наоборот, не равна нулю. Например, `TST R2, #0xFF` установит флаг *Z*, если младший байт R2 равен 0. Команда `TEQ` (test if equal – проверить на равенство) проверяет эквивалентность источников, вычисляя для них ИСКЛЮЧАЮЩЕЕ ИЛИ. Если источники равны, то устанавливается флаг *Z*, а если различаются знаком, то флаг *N*.

произведения в R1; старшие 32 бита отбрасываются. Эта команда полезна для умножения небольших чисел, произведение которых умещается в 32 бита. Команды `UMULL` (unsigned multiply long – длинное умножение без знака) и `SMULL` (signed multiply long – длинное умножение со знаком) перемножают два 32-битовых числа и порождают 64-битовое произведение. Например, `UMULL R1, R2, R3, R4` вычисляет произведение значений в регистрах R3 и R4, рассматриваемых как целые без знака. Младшие 32 бита произведения помещаются в регистр R1, а старшие 32 бита – в регистр R2.

У каждой из этих команд имеется также вариант с накоплением, `MLA`, `SMLAL` и `UMLAL`, который прибавляет произведение к накопительной сумме, 32- или 64-битовой. Эти команды повышают производительность в некоторых математических расчетах, например при умножении матриц или обработке сигналов, когда требуется многократно выполнять операции умножения и сложения.

1.3.2. Флаги условий

Было бы скучно, если бы команды, составляющие программу, при каждом запуске выполнялись в одном и том же порядке. Команды ARM дополнительно устанавливают *флаги условий*, зависящие от того, оказался ли результат отрицательным, равным нулю и т. д. Следующие за ними команды могут выполняться *условно* – в зависимости от состояния флагов условий. В ARM есть следующие флаги условий (их еще называют *флагами состояния*): отрицательно (*N*), равно нулю (*Z*), перенос (*C*) и переполнение (*V*) (см. **Табл. 1.2**). Эти флаги устанавливает АЛУ (см. **раздел 5.2.4** (книга 1)), и занимают они старшие 4 бита 32-битового *регистра текущего состояния программы* (current program status register, CPSR), показанного на **Рис. 1.6**.

Таблица 1.2. Флаги условий

Флаг	Название	Описание
N	Negative	Результат выполнения команды отрицателен, т. е. бит 31 равен 1
Z	Zero	Результат выполнения команды равен нулю
C	Carry	Команда привела к переносу
V	oVerflow	Команда привела к переполнению

Самый распространенный способ установить биты состояния – выполнить команду сравнения (`CMP`), которая вычитает второй операнд из первого и устанавливает флаги условий в зависимости от результата. Например, если оба числа равны, то результат будет равен 0, поэтому поднимается флаг *Z*. Если первое число, рассматриваемое как беззнаковое, больше или равно второму числу, то при вычитании произойдет перенос и будет поднят флаг *C*.

Последующие команды можно выполнять в зависимости от состояния флагов. В названиях команд применяется *мнемоника условий*, описывающая, когда выполнять команду. В **Табл. 1.3** перечислены значения 4-битового поля условия (*cond*), мнемоника и полное название условия и состояние флагов условий, при которых эта команда выполняется (CondEx). Допустим, к примеру, что программа выполняет команду `CMP R4, R5`, а затем `ADDEQ R1, R2, R3`. Команда сравнения установит флаг *Z*, если R4 и R5 равны, а команда `ADDEQ` будет выполнена, только если флаг *Z* установлен. Поле *cond* используется в кодировке машинных команд, как описано в **разделе 1.4**.

Таблица 1.3. Мнемонические обозначения условий

cond	Мнемоника	Название	CondEx
0000	EQ	Равно	Z
0001	NE	Не равно	$\overline{Z}$
0010	CS/HS	Флаг переноса поднят / беззнаковое больше или равно	C
0011	CC/LO	Флаг переноса сброшен / беззнаковое меньше	$\overline{C}$
0100	MI	Минус / отрицательное	N
0101	PL	Плюс / положительное или нуль	$\overline{N}$
0111	VS	Переполнение / флаг переполнения поднят	V
1000	VC	Переполнения нет / флаг переполнения сброшен	$\overline{V}$
1001	HI	Беззнаковое больше	$\overline{Z}C$
1010	LS	Беззнаковое меньше или равно	Z OR $\overline{C}$
1011	GE	Знаковое больше или равно	$\overline{N \oplus V}$
1100	LT	Знаковое меньше	$N \oplus V$
1101	GT	Знаковое больше	$\overline{Z}(\overline{N \oplus V})$
1110	LE	Знаковое меньше или равно	Z OR $(N \oplus V)$
	AL (или пусто)	Всегда / безусловно	Игнорируется

Другие команды обработки данных устанавливают флаги условий, когда мнемоническое обозначение команды сопровождается суффиксом «`S`». Например, команда `SUBS R2, R3, R7` вычитает R7 из R3, помещает результат в R2 и устанавливает флаги условий. В **Табл. B.5** из **прило-**

Мнемонические обозначения условий различаются для сравнения со знаком и без знака. Так, в ARM есть две формы сравнения на больше или равно: HS (CS) для чисел без знака и GE для чисел со знаком. Для чисел без знака вычисление разности `A – B` поднимает флаг переноса (`C`), если `A ≥ B`. Для чисел со знаком в результате вычисления `A – B` флаги `N` и `V` будут одновременно равны 0 или 1, если `A ≥ B`. На Рис. 1.7 показано различие между сравнениями HS и GE на двух примерах 4-битовых чисел (для простоты).

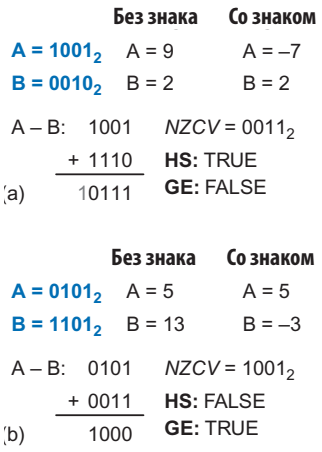


Рис. 1.7. Сравнение чисел со знаком и без знака: HS и GE

жения A описано, на какие флаги условий влияет каждая команда. Все команды обработки данных устанавливают флаги Z и N, если результат соответственно равен нулю или в нем поднят старший бит. Команды `ADDS` и `SUBS` влияют также на флаги `V` и `C`, а команды сдвига – на флаг `C`.

В **примере кода 1.10** иллюстрируется условное выполнение команд. Первая команда, `CMP R2, R3`, выполняется безусловно и устанавливает флаги условий. Остальные команды выполняются условно в зависимости от значений флагов условий. Пусть R2 и R3 содержат значения 0x80000000 и 0x00000001. Команда сравнения вычисляет разность R2 – R3 = 0x80000000 – 0x00000001 = 0x80000000 + 0xFFFFFFFF = 0x7FFFFFFF и устанавливает флаг переноса ($C = 1$). Знаки операндов-источников противоположны, и знак результата отличается от знака первого источника, поэтому имеет место переполнение результата ($V = 1$). Два оставшихся флага (N и Z) равны 0. Команда `ANDHS` выполняется, потому что $C = 1$. Команда `EORLT` выполняется, потому что $N = 0$ и $V = 1$ (см. **Табл. 1.3**). Интуитивно понятно, что `ANDHS` и `EORLT` выполняются, потому что R2 ≥ R3 (без знака) и R2 < R3 (со знаком) соответственно. Команды `ADDEQ` и `ORRMI` не выполняются, потому что результат вычисления R2 – R3 не равен нулю (т. е. R2 ≠ R3) и не отрицателен.

Пример кода 1.10. УСЛОВНОЕ ВЫПОЛНЕНИЕ

Код на языке ассемблера ARM

```
CMP   R2, R3
ADDEQ R4, R5, #78
ANDHS R7, R8, R9
ORRMI R10, R11, R12
EORLT R12, R7, R10
```

1.3.3. Переходы

Преимуществом компьютера над калькулятором является способность принимать решения. Компьютер выполняет разные задачи в зависимости от входных данных. Например, предложения if/else, switch/case, циклы while и for выполняют те или иные части кода в зависимости от результата проверки некоторых условий.

Один из способов принятия решений – воспользоваться условным выполнением, чтобы пропустить некоторые команды. Это годится для простых предложений if, когда нужно пропустить небольшое количество

команд, но расточительно, если в теле предложения if много команд, и недостаточно, если требуется цикл. Поэтому в ARM и в большинстве других архитектур имеются команды переходы для пропуска участков кода или повторения кода.

Обычно программа выполняется последовательно, и после выполнения каждой команды счетчик команд (PC) увеличивается на 4 и указывает на следующую команду (напомним, что длина любой команды равна 4 и что в ARM адресация памяти побайтовая). Команды перехода изменяют счетчик команд. В ARM есть два типа переходов: просто *перейти* (B) и *перейти и связать* (branch and link) (BL). Команда BL применяется для вызова функций и рассматривается в **разделе 1.3.7**. Как и другие команды в ARM, переходы могут быть условными и безусловными. В некоторых других архитектурах команды перехода называются не *branch*, а *jump*.

В **примере кода 1.11** показан безусловный переход с помощью команды B. Когда программа доходит до команды B TARGET, производится переход. Это значит, что следующей выполняется команда SUB после *метки* TARGET.

Пример кода 1.11. БЕЗУСЛОВНЫЙ ПЕРЕХОД

Код на языке ассемблера ARM

```
   ADD R1, R2, #17   ; R1 = R2 + 17
   B   TARGET        ; переход к TARGET
   ORR R1, R1, R3    ; не выполняется
   AND R3, R1, #0xFF ; не выполняется

 TARGET
   SUB R1, R1, #78   ; R1 = R1 - 78
```

В ассемблерном коде меткой обозначается положение команды в программе. В процессе трансляции ассемблерного кода в машинный метки заменяются адресами команд (см. **раздел 1.4.3**). В ассемблере ARM метки не должны совпадать с зарезервированными словами, например с мнемоническими обозначениями команд. Обычно программисты записывают команды с отступом, а метки без отступа, чтобы их было сразу видно. Компилятор ARM явно требует этого: метки должны начинаться в первой позиции строки, а командам должен предшествовать хотя бы один пробел. Некоторые компиляторы, в т. ч. GCC, требуют, чтобы после метки стояло двоеточие.

Команды перехода можно выполнять условно; при написании кода помогают мнемонические обозначения в **Табл. 1.3**. В **примере кода 1.12** демонстрируется команда BEQ, которая осуществляет переход по равенству ($Z = 1$). Когда программа доходит до команды BEQ, флаг Z равен 0 (т. е.

R0 ≠ R1), поэтому переход не производится и, значит, выполняется следующая команда, ORR.

Пример кода 1.12. УСЛОВНЫЙ ПЕРЕХОД

Код на языке ассемблера ARM

```
   MOV R0, #4          ; R0 = 4
   ADD R1, R0, R0      ; R1 = R0 + R0 = 8
   CMP R0, R1          ; установить флаги по результатам выполнения R0-R1 = -4. NZCV = 1000
   BEQ THERE           ; переход не производится (Z != 1)
   ORR R1, R1, #1      ; R1 = R1 OR 1 = 9

 THERE
   ADD R1, R1, #78     ; R1 = R1 + 78 = 87
```

1.3.4. Условные предложения

Условные предложения if, if/else и switch/case часто используются в языках высокого уровня. Они условно выполняют *блок* кода, содержащий одно или несколько предложений. В этом разделе показано, как эти высокоуровневые конструкции транслируются на язык ассемблера ARM.

Предложения if

Предложение if выполняет блок кода, называемый *блоком if*, только если выполнено заданное условие. В **примере кода 1.13** демонстрируется, как предложение if транслируется на язык ассемблера ARM.

Пример кода 1.13. ПРЕДЛОЖЕНИЕ IF

Код на языке высокого уровня

```
if (apples == oranges)
   f = i + 1;


f = f - i;
```

Код на языке ассемблера ARM

```
; R0 = apples, R1 = oranges, R2 = f, R3 = i
  CMP R0, R1      ; apples == oranges ?
  BNE L1          ; если не равно, пропустить блок if
  ADD R2, R3, #1 ; блок if: f = i + 1
L1
  SUB R2, R2, R3 ; f = f - i
```

В ассемблерном коде, соответствующем предложению if, проверяется условие, противоположное тому, что находится в высокоуровневом коде. В **примере кода 1.13** в высокоуровневом коде проверяется, что `apples == oranges`. А в ассемблерном коде проверяется условие `apples != oranges` — с помощью команды BNE, которая пропускает блок if, если условие не

Напомним, что в коде на языке высокого уровня != обозначает сравнение на неравенство, а == — сравнение на равенство.

выполнено. Если же `apples == oranges`, то переход не производится и блок if выполняется.

Поскольку любую команду можно выполнить условно, ассемблерный код в примере 1.13 можно было бы записать более компактно:

```
CMP   R0, R1       ; apples == oranges ?
ADDEQ R2, R3, #1   ; f = i + 1 в случае равенства (т.е. Z = 1)
SUB   R2, R2, R3   ; f = f - i
```

Решение с условным выполнением команд короче и быстрее, потому что в нем на одну команду меньше. Более того, как мы увидим в **разделе 2.5.3**, ветвление иногда приводит к дополнительной задержке, тогда как условное выполнение всегда одинаково быстро. Этот пример демонстрирует эффективность условного выполнения в архитектуре ARM.

Вообще говоря, если блок кода содержит только одну команду, то лучше воспользоваться условным выполнением, чем обходить этот блок с помощью команды перехода. Но если блок длиннее, то ветвление оказывается более полезно, поскольку позволяет не тратить время на выборку команд, которые не будут выполняться.

Предложения if/else

Предложение if/else выполняет один из двух блоков кода в зависимости от условия. Если условие в предложении if удовлетворяется, то выполняется блок if, в противном случае – блок else. В **примере кода 1.17** демонстрируется предложение if/else.

Как и в случае предложения if, ассемблерный код if/else проверяет условие, противоположное тому, что задано в коде на языке высокого уровня. Так, в **примере кода 1.14** в высокоуровневом коде проверяется условие `apples == oranges`, а в ассемблерном коде – условие `apples != oranges`. Если противоположное условие истинно, то команда `BNE` пропускает блок if и выполняет блок else. В противном случае блок if выполняется и завершается командой безусловного перехода (B), которая обходит блок else.

Пример кода 1.14. ПРЕДЛОЖЕНИЕ IF/ELSE

Код на языке высокого уровня

```
if (apples == oranges)
   f = i + 1;



else
   f = f - i;
```

Код на языке ассемблера ARM

```
; R0 = apples, R1 = oranges, R2 = f, R3 = i
  CMP R0, R1     ; apples == oranges ?
  BNE L1         ; если не равно, пропустить блок if
  ADD R2, R3, #1 ; блок if: f = i + 1
  B   L2
L1
  SUB R2, R2, R3 ; f = f - i
L2
```

И на этот раз, поскольку любую команду можно выполнить условно, а команды в блоке if не изменяют флаги условий, то ассемблерный код в **примере 1.14** можно записать гораздо компактнее:

```
CMP R0, R1          ; apples == oranges?
ADDEQ R2, R3, #1    ; f = i + 1 в случае равенства  (т.е. Z = 1)
SUBNE R2, R2, R3    ; f = f - i в случае неравенства  (т.е. Z = 0)
```

Предложения switch/case

Предложение switch/case выполняет один из нескольких блоков кода в зависимости от того, какое из условий удовлетворяется. Если не удовлетворяется ни одно условие, то выполняется *блок default*. Предложение switch/case эквивалентно последовательности *вложенных* предложений if/else. В **примере кода 1.15** показаны два фрагмента на языке высокого уровня с одной и той же функциональностью: они вычисляют, какие купюры – достоинством 20, 50 или и 100 долларов – выдавать в зависимости от нажатой на банкомате кнопки. Реализация на языке ассемблера ARM одинакова в обоих случаях.

Пример кода 1.15. ПРЕДЛОЖЕНИЕ SWITCH/CASE

Код на языке высокого уровня

```
switch (button) {
  case 1: amt = 20; break;


  case 2: amt = 50; break;


  case 3: amt = 100; break;


  default: amt = 0;
}
// эквивалентный код с использованием
// предложений if/else
  if (button == 1)      amt = 20;
  else if (button == 2) amt = 50;
  else if (button == 3) amt = 100;
  else                  amt = 0;
```

Код на языке ассемблера ARM

```
; R0 = button, R1 = amt
  CMP R0, #1     ; кнопка 1 ?
  MOVEQ R1, #20  ; amt = 20, если кнопка 1
  BEQ DONE       ; break

  CMP R0, #2     ; кнопка 2 ?
  MOVEQ R1, #50  ; amt = 50, если кнопка 2
  BEQ DONE       ; break

  CMP R0, #3     ; кнопка 3?
  MOVEQ R1, #100 ; amt = 100, если кнопка 3
  BEQ DONE       ; break

  MOV R1, #0     ; по умолчанию amt = 0
DONE
```

1.3.5. Циклы

Цикл многократно выполняет блок кода в зависимости от условия. Конструкции for и while часто используются для организации циклов в языках высокого уровня. В этом разделе будет показано, как эти конструкции транслируются на язык ассемблера ARM с помощью команд условного перехода.

Циклы while

В языке C тип данных `int` соответствует одному машинному слову, представляющему целое число в дополнительном коде. В ARM слова 32-битовые, поэтому тип `int` представляет число в диапазоне $[-2^{31}, 2^{31} - 1]$.

Цикл while повторно выполняет блок кода, до тех пор пока условие не станет ложным. В **примере кода 1.16** в цикле while ищется значение x — такое, что $2^x = 128$. Цикл выполнится семь раз, прежде чем условие `pow = 128` окажется истинным.

Как и в случае предложения if/else, в ассемблерном коде, соответствующем циклу while, проверяется условие, противоположное тому, что есть в коде на языке высокого уровня. Если это противоположное условие истинно (в данном случае R0 = 128), то цикл while завершается. В противном случае (R0 ≠ 128) переход не производится и выполняется тело цикла.

В **примере кода 1.16** в цикле while значение переменной `pow` сравнивается с величиной 128. В случае равенства цикл завершается, а иначе `pow` удваивается (с помощью сдвига влево), `x` увеличивается на 1, и производится переход на начало цикла while.

Пример кода 1.16. ЦИКЛ WHILE

Код на языке высокого уровня

```
int pow = 1;x
int x = 0;


while (pow != 128) {
  pow = pow * 2;
  x = x + 1;
}
```

Код на языке ассемблера ARM

```
; R0 = pow, R1 = x
  MOV R0, #1     ; pow = 1
  MOV R1, #0     ; x = 0

WHILE
  CMP R0, #128   ; pow != 128 ?
  BEQ DONE       ; если pow == 128, выйти из цикла
  LSL R0, R0, #1 ; pow = pow * 2
  ADD R1, R1, #1 ; x = x + 1
  B   WHILE      ; повторить цикл
DONE
```

Циклы for

Очень часто производится следующая последовательность действий: инициализировать переменную до входа в цикл while, проверять в условии цикла значение переменной и изменять переменную на каждой итерации цикла. Цикл for предлагает удобную и краткую нотацию для такой последовательности. Выглядит он следующим образом:

```
for (инициализация; условие; операция цикла)
  предложение
```

Код инициализации выполняется до начала цикла for. Условие проверяется в начале каждой итерации. Если условие не выполнено, цикл завершается. Операция цикла выполняется в конце каждой итерации.

В **примере кода 1.17** складываются целые числа от 0 до 9. Переменная цикла, в данном случае `i`, инициализируется нулем и увеличивается на единицу в конце каждой итерации. Цикл продолжается, пока `i` меньше 10. Отметим, что в этом примере еще раз иллюстрируется соотношение между сравнениями в двух языках. В цикле проверяется условие продолжения, содержащее оператор <, поэтому в ассемблерном коде проверяется противоположное условие выхода из цикла >=.

Пример кода 1.17. ЦИКЛ FOR

Код на языке высокого уровня

```
int i;
int sum = 0;


for (i = 0; i < 10; i = i + 1) {
   sum = sum + i;
}
```

Код на языке ассемблера ARM

```
; R0 = i, R1 = sum
  MOV R1, #0      ; sum = 0
  MOV R0, #0      ; i = 0 инициализация цикла

FOR
  CMP R0, #10     ; i < 10 ?        проверить условие
  BGE DONE        ; если (i >= 10) выйти из цикла
  ADD R1, R1, R0  ; sum = sum + i   тело цикла
  ADD R0, R0, #1  ; i = i + 1       операция цикла
  B   FOR         ; повторить цикл
DONE
```

Циклы особенно полезны при доступе к большому объему однородных данных в памяти. Этот вопрос обсуждается ниже.

1.3.6. Память

Для удобства хранения и доступа однородные данные можно сгруппировать в *массив*. Массив располагается в ячейках памяти с последовательными адресами и таким образом занимает непрерывный участок памяти. Каждый элемент массива идентифицируется порядковым номером, называемым *индексом*. Количество элементов массива называется его *длиной*.

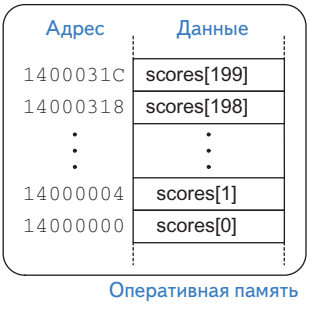


Рис. 1.8. Область памяти, содержащая массив `scores[200]` и начинающаяся с базового адреса 0x14000000

На **Рис. 1.8** показан размещенный в памяти массив оценок scores из 200 элементов. В **примере кода 1.18** приведен алгоритм увеличения оценок, который прибавляет к каждой оценке по 10 баллов. Отметим, что код инициализации массива не показан. Индексом массива является переменная (`i`), а не константа, поэтому мы должны умножать ее на 4, перед тем как прибавлять к базовому адресу.

В ARM есть команда, которая выполняет сразу три действия: *масштабировать* (умножить на коэффициент) индекс, прибавить результат к базовому адресу и загрузить данные из ячейки памяти с этим адресом. Вместо последо-

вательности команд LSL и LDR, как в **примере кода 1.18**, можно написать одну команду:

```
LDR R3, [R0, R1, LSL #2]
```

Регистр R1 масштабируется (сдвигается влево на два бита), а затем прибавляется к базовому адресу в регистре R0. Таким образом, адрес в памяти равен R0 + (R1 × 4).

Пример кода 1.18. ДОСТУП К МАССИВУ В ЦИКЛЕ FOR

Код на языке высокого уровня

```
int i;
int scores[200];
...


for (i = 0; i < 200; i = i + 1)


   scores[i] = scores[i] + 10;
```

Код на языке ассемблера ARM

```
; R0 = базовый адрес массива, R1 = i
; код инициализации...
  MOV R0, #0x14000000 ; R0 = базовый адрес
  MOV R1, #0          ; i = 0

LOOP
  CMP R1, #200        ; i < 200?
  BGE L3              ; если i ≥ 200, выйти из цикла
  LSL R2, R1, #2      ; R2 = i * 4
  LDR R3, [R0, R2]    ; R3 = scores[i]
  ADD R3, R3, #10     ; R3 = scores[i] + 10
  STR R3, [R0, R2]    ; scores[i] = scores[i] + 10
  ADD R1, R1, #1      ; i = i + 1
  B   LOOP            ; повторить цикл
L3
```

В дополнение к масштабированию индексного регистра ARM предлагает адресацию со смещением, а также с предындексацией и постиндексацией. Это позволяет писать лаконичный и эффективный код для доступа к массиву и вызова функций. В **Табл. 1.4** приведены примеры всех трех режимов индексации. Во всех случаях R1 – базовый регистр, а R2 содержит смещение. Смещение можно вычитать, нужно только написать –R2. Смещение также можно задавать в виде непосредственного операнда в диапазоне от 0 до 4095, прибавляемого (например, #20) или вычитаемого (#–20).

В режиме *адресации со смещением* адрес вычисляется как базовый адрес ± смещение; при этом сам базовый регистр не изменяется. В режиме *адресации с предындексацией* адрес тоже вычисляется как базовый адрес ± смещение, а затем этот адрес записывается в базовый регистр. В режиме *адресации с постиндексацией* в качестве адреса берется базовый регистр, а уже после доступа к памяти в базовый регистр записывается новый адрес – базовый ± смещение. Мы уже видели много примеров режима индексации со смещением. В **примере кода 1.19** цикл из **примера 1.18** переписан с использованием адресации с постиндексацией, что позволило отказаться от команды ADD для инкремента i.

Таблица 1.4. Режимы адресации в ARM

Режим	Ассемблер ARM	Адрес	Базовый регистр
Со смещением	`LDR R0, [R1, R2]`	R1 + R2	Без изменения
С предындексацией	`LDR R0, [R1, R2]!`	R1 + R2	R1 = R1 + R2
С постиндексацией	`LDR R0, [R1], R2`	R1	R1 = R1 + R2

Пример кода 1.19. ЦИКЛ FOR С ПОСТИНДЕКСАЦИЕЙ

Код на языке высокого уровня

```
int i;
int scores[200];
...


for (i = 0; i < 200; i = i + 1)
   scores[i] = scores[i] + 10;
```

Код на языке ассемблера ARM

```
; R0 = базовый адрес массива, R1 = i
; код инициализации...
  MOV R0, #0x14000000 ; R0 = базовый адрес
  ADD R1, R0, #800    ; R1 = базовый адрес + (200*4)

LOOP
  CMP R0, R1          ; достигнут конец массива?
  BGE L3              ; если да, выйти из цикла
  LDR R2, [R0]        ; R2 = scores[i]
  ADD R2, R2, #10     ; R2 = scores[i] + 10
  STR R2, [R0], #4    ; scores[i] = scores[i] + 10
                      ; затем R0 = R0 + 4
  B   LOOP            ; повторить цикл
L3
```

Байты и символы

Любое число из диапазона [−128, 127] можно сохранить в одном байте, целое слово для этого не нужно. Так как на англоязычной клавиатуре менее 256 символов, то символы английского языка обычно представляются байтами. В языке C для представления байтов или символов используется тип данных `char`.

В ранних компьютерах отсутствовало стандартное соответствие между байтами и символами английского языка, поэтому текстовый обмен между компьютерами был затруднителен. В 1963 году американская ассоциация по стандартизации опубликовала *Американский стандартный код для обмена информацией* (American Standard Code for Information Interchange, *ASCII*), в котором каждому символу было назначено уникальное значение байта. В **Табл. 1.5** перечислены коды всех печатных символов. Значения ASCII приведены в шестнадцатеричной форме. Буквы верхнего и нижнего регистров отличаются на 0x20 (32).

В других языках программирования, например Java, используются иные кодировки символов, в частности Юникод (Unicode). В Юникоде каждый символ кодируется 16 битами, что позволяет поддерживать диакритические знаки и азиатские языки. Дополнительные сведения см. на сайте www.unicode.org.

Команды `LDRH`, `LDRSH` и `STRH` аналогичны, но осуществляют доступ не к байтам, а к 16-битовым полусловам.

Таблица 1.5. Кодировка ASCII

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

В архитектуре ARM для доступа к отдельным байтам в памяти предусмотрены команды *загрузить байт* (LDRB), *загрузить байт со знаком* (LDRSB) и *сохранить байт* (STRB). Чтобы заполнить 32-битовый регистр целиком, команда LDRB дополняет байт нулями, а команда LDRSB – знаковым битом. Команда STRB сохраняет по указанному адресу в памяти младший байт 32-битового регистра. Все три команды показаны на **Рис. 1.9**, где считается, что базовый адрес в регистре R4 равен 0. LDRB загружает байт из ячейки с адресом 2 в младший байт R1, а остальные биты регистра заполняет нулями. LDRSB загружает тот же байт в R2 и копирует в старшие 24 бита регистра знаковый бит загруженного байта. STRB сохраняет младший байт R3 (0x9B) в ячейке памяти с адресом 3; ее старое содержимое 0xF7 заменяется значением 0x9B. Старшие байты R3 игнорируются.

Кодировка ASCII стала развитием более ранних форм символьных кодировок. В 1838 году на телеграфе начали использовать азбуку Морзе, т. е. последовательность точек и тире, для передачи символов. Например, буквы A, B, C и D представляются как «. –», «– ...», «– . – .» и «– ..» соответственно. Количество и порядок точек и тире отличаются для каждой буквы. Для повышения эффективности часто встречающимся буквам соответствуют более короткие коды.

В 1874 году Жан Морис Эмиль Бодо изобрел 5-битный код, названный кодом Бодо. В нем A, B, C и D были представлены как 00011, 11001, 01110 и 01001.

Однако 32 возможных вариантов этого 5-битного кода было недостаточно для всех английских символов, но 8-битной кодировки уже хватало. Поэтому с развитием электронных средств связи 8-битная кодировка ASCII стала стандартом.

Рис. 1.9. Команды загрузки и сохранения байтов

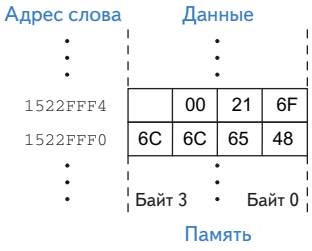


Рис. 1.10. Строка «Hello!» в памяти

Последовательность символов называют *строкой* (string). У строк переменная длина, поэтому язык программирования должен предоставлять какой-нибудь способ определения либо длины, либо конца строки. В языке C конец строки обозначается нулевым символом (0x00). На **Рис. 1.10** показана строка «Hello!» (0x48 65 6C 6C 6F 21 00), хранимая в памяти. Длина строки составляет семь байтов, строка занимает адреса с 0x1522FFF0 по 0x1522FFF6 включительно. Первый символ строки (H = 0x48) хранится по наименьшему адресу (0x1522FFF0).

Пример 1.2. ПРИМЕНЕНИЕ КОМАНД LDRB И STRB ДЛЯ ДОСТУПА К МАССИВУ СИМВОЛОВ

Приведенный ниже код на языке высокого уровня преобразует буквы, находящиеся в массиве символов из 10 элементов, из строчных в прописные путем вычитания 32 из каждого элемента массива. Транслируйте этот код на язык ассемблера ARM. Не забудьте, что адреса соседних элементов массива теперь отличаются на один, а не на четыре байта. Считайте, что регистр R0 уже содержит базовый адрес массива `chararray`.

```
// код на языке высокого уровня
// массив chararray[10] объявлен и инициализирован раньше
int i;

for (i = 0; i < 10; i = i + 1)
  chararray[i] = chararray[i] - 32;
```

Решение.

```
; код на языке ассемблера ARM
; R0 = базовый адрес chararray (инициализирован раньше), R1 = i
       MOV R1, #0          ; i = 0
LOOP   CMP R1, #10         ; i < 10 ?
       BGE DONE            ; если (i >=10), выйти из цикла
       LDRB R2, [R0, R1]   ; R2 = mem[R0+R1] = chararray[i]
       SUB R2, R2, #32     ; R2 = chararray[i] - 32
       STRB R2, [R0, R1]   ; chararray[i] = R2
       ADD R1, R1, #1      ; i = i + 1
       B LOOP              ; повторить цикл
DONE
```

1.3.7. Вызовы функций

В языках высокого уровня поддерживаются *функции* (их также называют *процедурами*, или *подпрограммами*) для повторного использования часто выполняемого кода и для того, чтобы сделать программу модульной и удобочитаемой. У функций есть входные данные, называемые *аргументами*, и выходной результат, называемый *возвращаемым значением*. Функция должна вычислять возвращаемое значение, не вызывая неожиданных побочных эффектов.

Когда одна функция вызывает другую, вызывающая и вызываемая функции должны согласовать, где размещать аргументы и возвращаемое значение. В архитектуре ARM принято соглашение о том, что вызывающая функция размещает до четырех аргументов в регистрах R0–R3, перед тем как произвести вызов, а вызываемая функция помещает возвращаемое значение в регистр R0, перед тем как вернуть управление. Следуя этому соглашению, обе функции знают, где искать аргументы и возвращенное значение, даже если вызывающая и вызываемая функции были написаны разными людьми.

Вызываемая функция не должна вмешиваться в работу вызывающей. Это означает, что вызываемая функция должна знать, куда передать управление после завершения работы, и не должна изменять значения регистров или памяти, необходимых вызывающей функции. Вызывающая функция сохраняет *адрес возврата* в регистре связи LR одновременно с передачей управления вызываемой функции путем выполнения команды *перейти и связать* (branch and link, `BL`). Вызываемая функция не должна изменять архитектурное состояние и содержимое памяти, от которых зависит вызывающая функция. В частности, вызываемая функция должна оставить неизменным содержимое сохраняемых регистров (R4–R11 и LR) и *стека* – участка памяти, используемого для хранения временных переменных.

В этом разделе мы покажем, как вызывать функции и возвращаться из них. Мы увидим, как функции получают доступ к входным аргументам и возвращают значение, а также то, как они используют стек для хранения временных переменных.

Вызов функции и возврат из функции

В архитектуре ARM для вызова функции используется команда *перейти и связать* (`BL`), а для возврата из функции нужно поместить регистр связи в счетчик команд (`MOV PC, LR`). В **примере кода 1.20** показана функция `main`, вызывающая функцию `simple`. Здесь функция `main` является вызывающей, а `simple` – вызываемой. Функция `simple` не получает входных аргументов и ничего не возвращает, она просто передает управление обратно вызывающей функции. В **примере кода 1.20** слева от каждой команды ARM приведен ее адрес в шестнадцатеричном виде.

Команды BL (перейти и связать) и MOV PC, LR необходимы для вызова и возврата из функции. BL выполняет две операции: сохраняет адрес следующей за ней команды (*адрес возврата*) в регистре связи (LR) и производит переход по указанному адресу.

Пример кода 1.20. ВЫЗОВ ФУНКЦИИ simple

Код на языке высокого уровня

```
int main() {

  simple();
  ...
}

// void означает, что функция не
// возвращает значения
void simple() {
}
```

Код на языке ассемблера ARM

```
0x00008000 MAIN   ...
...               ...
0x00008020        BL SIMPLE ; вызвать функцию simple
...



0x0000902C SIMPLE MOV PC, LR ; возврат
```

Напомним, что PC и LR — альтернативные имена регистров R15 и R14 соответственно. Архитектура ARM необычна тем, что PC входит в состав набора регистров, а значит, для возврата из функции достаточно команды MOV. Во многих других наборах команд PC является специальным регистром, а для возврата из функции применяется специальная команда возврата или перехода. В настоящее время компиляторы для ARM производят возврат из функции с помощью команды BX LR. Команда BX (перейти и обменять — branch and exchange) похожа на обычную команду перехода, но может также переключаться между стандартным набором команд ARM и набором команд Thumb, описанным в разделе 1.7.1. В этой главе набор команд Thumb и команда BX не используются, поэтому мы будет придерживаться принятого в версии ARMv4 метода MOV PC, LR.
В главе 2 мы увидим, что трактовка PC как обычного регистра усложняет реализацию процессора.

В **примере кода 1.20** функция main вызывает функцию simple, выполняя команду *перейти и связать* (BL). Команда BL производит переход на метку SIMPLE и сохраняет значение 0x00008024 в регистре LR. Функция simple немедленно завершается, выполняя команду MOV PC, LR, т. е. копирует адрес возврата из LR в PC. После этого функция main продолжает выполнение с этого адреса (0x00008024).

Входные аргументы и возвращаемые значения

В **примере кода 1.20** функция simple не получает входных данных от вызывающей функции (main) и ничего не возвращает. По соглашениям, принятым в архитектуре ARM, функции используют регистры R0–R3 для входных аргументов и регистр R0 для возвращаемого значения. В **примере кода 1.21** функция diffofsums вызывается с четырьмя аргументами и возвращает один результат. Локальную переменную result мы решили поместить в регистр R4.

Следуя соглашениям ARM, вызывающая функция main помещает аргументы функции слева направо во входные регистры R0–R3. Вызываемая функция diffofsums помещает возвращаемое значение в регистр возврата R0. Если у функции больше четырех аргументов, то остальные помещаются в стек, о чем мы поговорим ниже.

Пример кода 1.21. ВЫЗОВ ФУНКЦИИ С АРГУМЕНТАМИ И ВОЗВРАЩАЕМЫМ ЗНАЧЕНИЕМ

Код на языке высокого уровня

```
int main() {
  int y;
  ...
  y = diffofsums(2, 3, 4, 5);
  ...
}



int diffofsums(int f, int g, int h, int i) {

  int result;

  result = (f + g) - (h + i);
  return result;
 }
```

Код на языке ассемблера ARM

```
; R4 = y
MAIN
  ...
  MOV R0, #2      ; аргумент 0 = 2
  MOV R1, #3      ; аргумент 1 = 3
  MOV R2, #4      ; аргумент 2 = 4
  MOV R3, #5      ; аргумент 3 = 5
  BL DIFFOFSUMS   ; вызвать функцию
  MOV R4, R0      ; y = возвращаемое значение
  ...

; R4 = result

DIFFOFSUMS
  ADD R8, R0, R1 ; R8 = f + g
  ADD R9, R2, R3 ; R9 = h + i
  SUB R4, R8, R9 ; result = (f + g) - (h + i)
  MOV R0, R4     ; поместить возвращаемое значение в R0
  MOV PC, LR     ; вернуть управление
```

Стек

Стек – это область памяти для хранения локальных переменных функции. Стек расширяется (занимает больше памяти), если процессору нужно больше места, и сжимается (занимает меньше памяти), если процессору больше не нужны сохраненные там переменные. Прежде чем объяснять, как функции используют стек для хранения временных переменных, расскажем, как стек работает.

Стек является очередью, работающей в режиме «последним пришел – первым ушел» (LIFO, last-in-first-out). Как и в стопке тарелок, последний элемент, помещенный (push) в стек (верхняя тарелка), будет первым элементом, который с него снимут (извлекут, pop). Каждая функция может выделить память в стеке для хранения локальных переменных, и она же должна освободить выделенную память перед возвратом. *Вершина стека* – это элемент, выделенный последним. Если стопка тарелок растет вверх, то стек в архитектуре ARM по мере необходимости расширяется вниз в памяти, т. е. в сторону младших адресов.

На **Рис. 1.11** изображен стек. *Указатель стека* SP (от англ. *stack pointer*) в ARM – это обычный регистр, который по соглашению *указывает* на *вершину стека*. *Указатель* – просто другое название адреса памяти. SP

В примере кода 1.21 есть тонкие ошибки. В примерах кода 1.22–1.25 приведены исправленные версии этой программы.

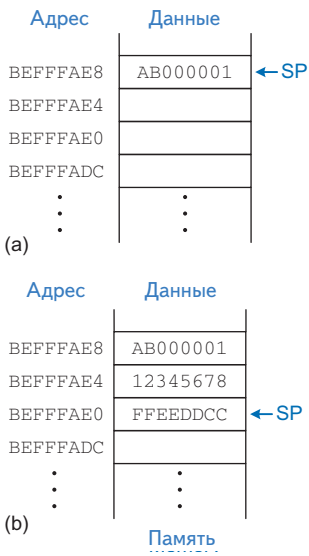


Рис. 1.11. Стек (a) до расширения и (b) после расширения на два слова

Обычно стек размещается в памяти сверху вниз, так что вершина стека имеет наибольший адрес и стек растет вниз в сторону меньших адресов. Такой стек называется *нисходящим*. В ARM разрешены также *восходящие стеки*, которые растут в сторону больших адресов. Указатель стека обычно направлен на самый верхний его элемент; это называется *полным стеком*. В ARM разрешены также *пустые стеки*, когда SP указывает на адрес, расположенный на одно слово после (или перед — в зависимости от направления роста) вершины стека. В *двоичном интерфейсе приложений* ARM (Application Binary Interface, ABI) определен стандартный способ передачи аргументов функциям и порядок использования ими стека, чтобы библиотеки, собранные разными компиляторами, можно было использовать совместно. Стандарт настаивает на *полном нисходящем стеке*, и этого требования мы будем придерживаться в данной главе.

указывает на данные, то есть содержит их адрес. Так, на **Рис. 1.11 (a)** указатель стека SP содержит адрес 0xBEFFFAE8 и указывает на значение 0xAB000001.

Вначале указатель стека (SP) содержит большой адрес памяти и уменьшается, когда программа запрашивает дополнительное место в стеке. На **Рис. 1.11 (b)** изображен стек, расширяющийся, для того чтобы предоставить два дополнительных слова для хранения временных переменных. Для этого значение SP уменьшается на 8 и становится равным 0xBEFFFAE0. В стеке временно размещаются два дополнительных слова данных 0x12345678 и 0xFFEEDDCC.

Одно из важных применений стека – сохранение и восстановление регистров, используемых внутри функции. Напомним, что функция должна вычислять возвращаемое значение и не иметь неожиданных побочных эффектов. В частности, она не должна изменять значения никаких регистров, кроме R0, в который помещается возвращаемое значение. В **примере кода 1.21** функция `diffofsums` нарушает это правило, потому что изменяет регистры R4, R8 и R9. Если бы функция `main` использовала эти регистры до вызова `diffofsums`, то в результате этого вызова их содержимое было бы повреждено.

Чтобы решить эту проблему, функция сохраняет значения регистров в стеке, прежде чем изменить их, и восстанавливает их из стека перед возвратом. Точнее, она выполняет следующие действия:

1) выделяет место в стеке для сохранения значений одного или нескольких регистров;
2) сохраняет значения регистров в стеке;
3) выполняет то, для чего предназначена, используя регистры;
4) восстанавливает исходные значения регистров из стека;
5) освобождает место в стеке.

В **примере кода 1.22** приведена исправленная версия функции `diffofsums`, которая сохраняет и восстанавливает регистры R4, R8 и R9. На **Рис. 1.12** показан стек до, во время и после вызова этой версии `diffofsums`. Стек начинается с адреса 0xBEF0F0FC. Функция `diffofsums` выделяет в стеке место для трех слов, уменьшая указатель стека SP на 12. Затем она сохраняет текущие значения R4, R8 и R9 в только что выделенной области. Дальше выпол-

няется остальная часть функции, в которой значения этих трех регистров изменяются. В самом конце функция `diffofsums` восстанавливает значения регистров из стека, освобождает выделенное в стеке место и возвращает управление. После возврата в регистре R0 будет находиться результат, но больше никаких побочных эффектов нет: в регистрах R4, R8, R9 и SP находятся те же значения, что и до вызова функции.

Пример кода 1.22. ФУНКЦИЯ, СОХРАНЯЮЩАЯ РЕГИСТРЫ В СТЕКЕ

Код на языке ассемблера ARM

```
;R4 = result
DIFFOFSUMS
  SUB SP, SP, #12       ; выделить в стеке место для 3 регистров
  STR R9, [SP, #8]      ; сохранить R9 в стеке
  STR R8, [SP, #4]      ; сохранить R8 в стеке
  STR R4, [SP]          ; сохранить R4 в стеке

  ADD R8, R0, R1        ; R8 = f + g
  ADD R9, R2, R3        ; R9 = h + i
  SUB R4, R8, R9        ; result = (f + g) - (h + i)
  MOV R0, R4            ; записать возвращенное значение в R0

  LDR R4, [SP]          ; восстановить R4 из стека
  LDR R8, [SP, #4]      ; восстановить R8 из стека
  LDR R9, [SP, #8]      ; восстановить R9 из стека
  ADD SP, SP, #12       ; освободить место в стеке

MOV PC, LR              ; вернуть управление
```

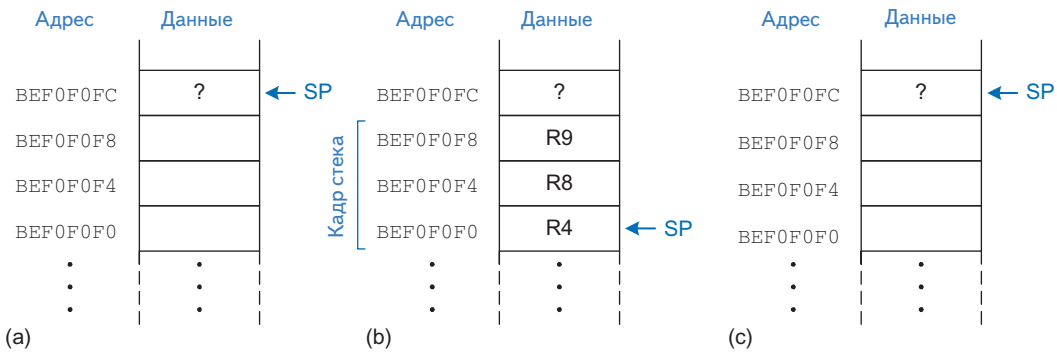


Рис. 1.12. Состояние стека: (a) до, (b) во время и (c) после вызова функции `diffofsums`

Область стека, которую функция выделяет для себя, называется *кадром стека*. В случае функции `diffofsums` кадр стека занимает три слова. Согласно принципу модульности, каждая функция вправе обращаться только к своему кадру стека, не залезая в кадры, принадлежащие другим функциям.

Загрузка и сохранение нескольких регистров

Загрузка и сохранение регистров в стеке – настолько распространенная операция, что в ARM имеются команды *загрузить несколько* (Load Multiple) и *сохранить несколько* (Store Multiple) (`LDM` и `STM`), оптимизированные для этой цели. В **примере кода 1.23** функция `diffofsums` переписана с использованием этих команд. В стеке хранится в точности та же информация, что и раньше, но код гораздо короче.

Пример кода 1.23. СОХРАНЕНИЕ И ВОССТАНОВЛЕНИЕ НЕСКОЛЬКИХ РЕГИСТРОВ

Код на языке ассемблера ARM

```
;R4 = result
DIFFOFSUMS
  STMFD SP!, {R4, R8, R9}       ; поместить R4/8/9 в полный нисходящий стек

  ADD R8, R0, R1                ; R8 = f + g
  ADD R9, R2, R3                ; R9 = h + i
  SUB R4, R8, R9                ; result = (f + g) - (h + i)
  MOV R0, R4                    ; записать возвращенное значение в R0

  LDMFD SP!, {R4, R8, R9}       ; извлечь R4/8/9 из полного нисходящего стека

  MOV PC, LR                    ; вернуть управление
```

Команды `LDM` и `STM` имеют по четыре варианта: для полных и пустых, нисходящих и восходящих стеков (`FD`, `ED`, `FA`, `EA`). Операнд `SP!` в командах означает, что следует сохранять данные относительно указателя стека и обновлять этот указатель после сохранения или загрузки. Команды `PUSH` и `POP`, являющиеся синонимами `STMFD SP!, {regs}` и `LDMFD SP! {regs}` соответственно, – предпочтительный способ сохранения регистров в традиционном полном нисходящем стеке.

Оберегаемые регистры

В **примерах кода 1.22** и **1.23** предполагается, что все используемые регистры (R4, R8 и R9) следует сохранять и восстанавливать. Если вызывающая функция не использует эти регистры, то время на их сохранения и восстановления потрачено впустую. Чтобы избежать этих издержек, в архитектуре ARM регистры разделены на две категории: *оберегаемые* (preserved) и *необерегаемые* (nonpreserved). К оберегаемым относятся регистры R4–R11, а к необерегаемым – R0–R3 и R12. Регистры SP и LR (R13 и R14) также должны оберегаться. Функция должна сохранять и восстанавливать любые оберегаемые регистры, с которыми собирается работать, но может беспрепятственно изменять значения необерегаемых регистров.

В **примере кода 1.24** показана улучшенная версия функции `diffofsums`, которая сохраняет в стеке только регистр R4. Демонстрируется также использование синонимов `PUSH` и `POP`. Для хранения промежуточных сумм используются необеpегаемые регистры R1 и R3, которые сохранять необязательно.

Пример кода 1.24. УМЕНЬШЕНИЕ КОЛИЧЕСТВА СОХРАНЯЕМЫХ РЕГИСТРОВ

Код на языке ассемблера ARM

```
;R4 = result
DIFFOFSUMS
  PUSH {R4}            ; сохранить R4 в стеке
  ADD R1, R0, R1       ; R1 = f + g
  ADD R3, R2, R3       ; R3 = h + i
  SUB R4, R1, R3       ; result = (f + g) - (h + i)
  MOV R0, R4           ; записать возвращенное значение в R0
  POP {R4}             ; извлечь R4 из стека
  MOV PC, LR           ; вернуть управление
```

Напомним, что когда одна функция вызывает другую, первая называется вызывающей функцией, а вторая – вызываемой. Вызываемая функция должна сохранять и восстанавливать любые оберегаемые регистры, которые собирается использовать, но вправе изменять любые необеpегаемые регистры. Следовательно, если вызывающая функция хранит важные данные в необеpегаемых регистрах, то она должна сохранять их перед вызовом другой функции и восстанавливать после возврата. По этой причине оберегаемые регистры также называют *сохраняемыми вызываемой функцией*, а необеpегаемые – *сохраняемыми вызывающей функцией*.

В **Табл. 1.6** перечислены оберегаемые регистры. Регистры R4–R11 обычно используются для хранения локальных переменных внутри функции, поэтому они должны быть сохранены. Регистр LR также следует сохранять, чтобы функция знала, куда возвращаться.

Таблица 1.6. Оберегаемые и необеpегаемые регистры

Оберегаемые	Необеpегаемые
Сохраняемые регистры: R4–R11	Временный регистр: R12
Указатель стека: SP (R13)	Регистры аргументов: R0–R3
Адрес возврата: LR (R14)	Регистр текущего состояния программы
Стек выше указателя стека	Стек ниже указателя стека

Регистры R0–R3 используются для хранения временных результатов вычислений. Такие вычисления обычно завершаются до вызова функции, поэтому эти регистры не оберегаются, а необходимость сохранять их в вызывающей функции возникает крайне редко.

Соглашение о том, какие регистры являются оберегаемыми, а какие нет, определено в «Стандарте вызова процедур для архитектуры ARM», а не в самой архитектуре. Существуют и альтернативные стандарты вызова.

Регистры R0–R3 часто изменяются вызываемой функцией, поэтому вызывающая функция должна сохранять их значения, если они могут понадобиться ей после возврата из вызванной функции. Регистр R0, очевидно, не следует оберегать, потому что в нем вызываемая функция возвращает свой результат. Напомним, что в регистре текущего состояния программы (CPSR) хранятся флаги условий. Он не сохраняется вызываемой функцией.

Стек выше указателя стека автоматически остается в сохранности, если только вызываемая функция не осуществляет запись в память по адресам выше SP. При соблюдении этого условия она не изменяет кадры стека, принадлежащие другим функциям. Сам указатель стека остается в сохранности, потому что вызываемая функция перед возвратом освобождает свой кадр стека, прибавляя к SP то же значение, которое вычла из него вначале.

Внимательный читатель (а равно оптимизирующий компилятор), вероятно, обратил внимание, что локальная переменная `result` просто возвращается и больше нигде не используется. А раз так, то без этой переменной можно обойтись и записывать значение прямо в регистр возврата R0, устранив тем самым необходимость сохранять и восстанавливать регистр R4 и копировать результат из R4 в R0. В **примере кода 1.25** показана эта дополнительная оптимизация функции `diffofsums`.

Пример кода 1.25. ОПТИМИЗИРОВАННАЯ ФУНКЦИЯ `diffofsums`

Код на языке ассемблера ARM

```
DIFFOFSUMS
  ADD R1, R0, R1              ; R1 = f + g
  ADD R3, R2, R3              ; R3 = h + i
  SUB R0, R1, R3              ; return (f + g) - (h + i)
  MOV PC, LR                  ; вернуть управление
```

Вызовы нелистовых функций

Функция, которая не вызывает другие функции, называется *листовой* (leaf function); примером может служить функция `diffofsums`. Функция, которая вызывает другие функции, называется *нелистовой* (nonleaf function). Как уже отмечалось, нелистовые функции устроены более сложно, потому что перед вызовом других функций им приходится сохранять необеpегаемые регистры в стеке и затем восстанавливать их. Точнее:

Правило сохранения для вызывающей функции: перед вызовом другой функции вызывающая функция должна сохранить

все необеpегаемые регистры (R0–R3 и R12), которые понадобятся ей после вызова. После возврата из вызванной функции вызывающая функция должна восстановить эти регистры.

Правило сохранения для вызываемой функции: прежде чем изменять какой-либо оберегаемый регистр, вызываемая функция должна сохранить его. Перед возвратом она должна восстановить эти регистры.

Пример кода 1.26. ВЫЗОВ НЕЛИСТОВОЙ ФУНКЦИИ

Код на языке высокого уровня

```
int f1(int a, int b) {
  int i, x;

  x = (a + b)*(a - b);
  for (i=0; i<a; i++)
    x = x + f2(b+i);
  return x;
}
```

```
int f2(int p) {
  int r;

  r = p + 5;
  return r + p;
}
```

Код на языке ассемблера ARM

```
; R0 = a, R1 = b, R4 = i, R5 = x
F1
  PUSH {R4, R5, LR} ; сохранить оберегаемые
                    ; регистры, используемые f1
  ADD R5, R0, R1    ; x = (a + b)
  SUB R12, R0, R1   ; temp = (a - b)
  MUL R5, R5, R12   ; x = x * temp = (a + b) * (a - b)
  MOV R4, #0        ; i = 0
FOR
  CMP  R4, R0       ; i < a?
  BGE  RETURN       ; нет: выйти из цикла
  PUSH {R0, R1}     ; сохранить необеpегаемые регистры
  ADD  R0, R1, R4   ; аргумент - b + i
  BL   F2           ; call f2(b+i)
  ADD  R5, R5, R0   ; x = x + f2(b+i)
  POP  {R0, R1}     ; восстановить необеpегаемые
                    ; регистры
  ADD  R4, R4, #1   ; i++
  B    FOR          ; продолжить цикл
RETURN
  MOV R0, R5        ; возвращается значение x
  POP {R4, R5, LR}  ; восстановить оберегаемые
                    ; регистры
  MOV PC, LR        ; возврат из f1

; R0 = p, R4 = r
F2
  PUSH {R4}         ; сохранить оберегаемые
                    ; регистры, используемые f2
  ADD R4, R0, 5     ; r = p + 5
  ADD R0, R4, R0    ; возвращается значение r + p
  POP {R4}          ; восстановить оберегаемые
                    ; регистры
  MOV PC, LR        ; возврат из f2
```

В **примере кода 1.26** показаны нелистовая функция `f1` и листовая функция `f2` со всеми необходимыми операциями сохранения и оберегания регистров. Предположим, что `f1` хранит переменную `i` в регистре R4,

Нелистовая функция перезаписывает LR, когда вызывает другую функцию, используя команду BL. Следовательно, нелистовая функция обязана сохранять регистр LR в своем стеке и восстанавливать его перед возвратом.

При внимательном рассмотрении можно заметить, что `f2` не модифицирует регистр R1, поэтому `f1` могла бы обойтись без его сохранения и восстановления. Но компилятору не всегда легко определить, какие необеregаемые регистры могут быть изменены вызванной функцией. Поэтому простой компилятор всегда заставляет вызывающую функцию сохранять и восстанавливать все необеregаемые регистры, которые могут понадобиться ей после вызова.

Оптимизирующий компилятор заметил бы, что `f2` — листовая функция, и отвел бы под `r` необеregаемый регистр, избежав тем самым необходимости сохранять и восстанавливать R4.

а переменную `x` — в R5. Функция `f2` хранит `r` в R4. `f1` использует обеregаемые регистры R4, R5 и LR, поэтому в самом начале работы помещает их в стек, следуя правилу сохранения для вызываемой функции. Она использует регистр R12 для хранения промежуточного результата `(a - b)`, поэтому сберегать еще один регистр для данного вычисления не нужно. Перед вызовом `f2` функция `f1` помещает R0 и R1 в стек, следуя правилу сохранения для вызывающей функции, поскольку это необеregаемые регистры, которые `f2` потенциально может изменить, а они будут нужны `f1` после возврата из `f2`. Хотя R12 — необеregаемый регистр, который `f2` тоже могла бы перезаписать, он больше не нужен `f1`, поэтому она его не сохраняет. Затем `f1` передает `f2` аргумент в регистре R0, вызывает функцию и использует результат, возвращенный в R0. Далее `f1` восстанавливает R0 и R1, поскольку они еще нужны. По завершении работы `f1` помещает возвращаемое значение в R0, восстанавливает обеregаемые регистры R4, R5 и LR и возвращает управление. Функция `f2` сохраняет и восстанавливает R4 в соответствии с правилом сохранения для вызываемой функции.

На **Рис. 1.13** показано состояние стека во время выполнения `f1`. Первоначально указатель стека был равен 0xBEF7FF0C.

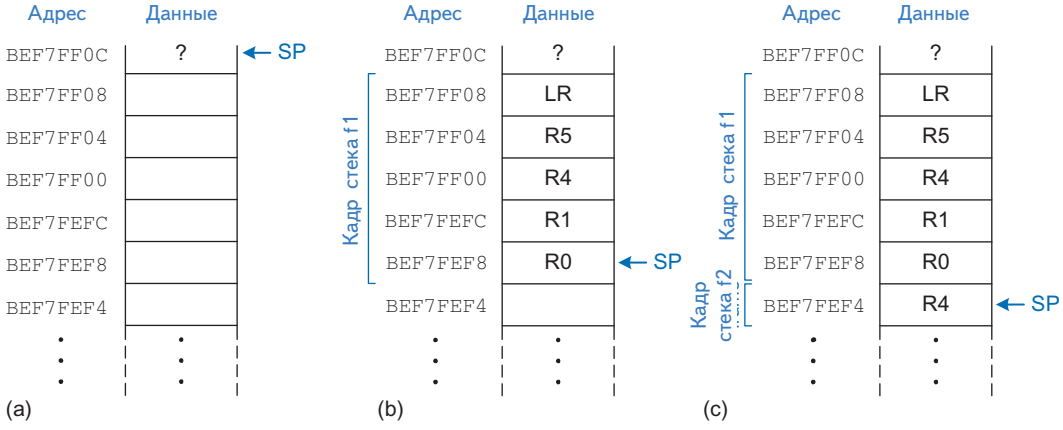


Рис. 1.13. Состояние стека: (a) до вызова обеих функций, (b) во время вызова `f1` и (c) во время вызова `f2`

Рекурсивные вызовы функций

Рекурсивной называется нелистовая функция, вызывающая саму себя. Рекурсивная функция ведет себя одновременно как вызывающая и как вызываемая, поэтому должна сохранять и оберегаемые, и необеregаемые регистры. Например, функция вычисления факториала может быть написана рекурсивно. Напомним, что $factorial(n) = n \times (n-1) \times (n-2) \times \ldots \times 2 \times 1$. Эту функцию можно переписать в рекурсивной форме: $factorial(n) = n \times factorial(n-1)$, как показано в **примере кода 1.27**. Факториал 1 равен 1. Для удобства будем предполагать, что программа начинается с адреса 0x8500.

Пример кода 1.27. ВЫЗОВ РЕКУРСИВНОЙ ФУНКЦИИ `factorial`

Код на языке высокого уровня

```
int factorial(int n) {
  if (n <= 1)
    return 1;




  else
    return (n * factorial(n - 1));
}
```

Код на языке ассемблера ARM

```
0x8500  FACTORIAL PUSH {R0, LR}  ; поместить n и LR в стек
0x8504         CMP R0, #1        ; R0 <= 1?
0x8508         BGT ELSE          ; нет: перейти к else
0x850C         MOV R0, #1        ; иначе вернуть 1
0x8510         ADD SP, SP, #8    ; восстановить SP
0x8514         MOV PC, LR        ; возврат
0x8518 ELSE    SUB R0, R0, #1    ; n = n - 1
0x851C         BL FACTORIAL      ; рекурсивный вызов
0x8520         POP {R1, LR}    ; восстановить n (в R1) и LR
0x8524         MUL R0, R1, R0 ; R0 = n * factorial(n - 1)
0x8528         MOV PC, LR        ; возврат
```

Согласно правилу сохранения для вызываемой функции, `factorial` – нелистовая функция и должна сохранять регистр LR. Поскольку `factorial` будет использовать `n` после вызова себя, то, согласно правилу сохранения для вызывающей функции, она должна сохранить R0. Поэтому в начале работы оба регистра помещаются в стек. Затем функция проверяет условие `n ≤ 1`. Если оно выполнено, то в регистр R0 записывается 1, после чего функция восстанавливает указатель стека и возвращает управление вызывающей функции. В этом случае нет необходимости перезагружать LR и R0, поскольку они не были изменены. Если `n > 1`, то функция рекурсивно вызывает `factorial(n - 1)`. Затем она восстанавливает значение `n` и регистр связи (LR) из стека, выполняет умножение и возвращает результат. Отметим, что функция предусмотрительно восстанавливает `n` в R1, чтобы не перезаписать возвращенное значение. Команда `MUL R0, R1, R0` умножает `n` (R1) на возвращенное значение (R0) и помещает результат в R0.

Для простоты мы всегда будем сохранять регистры в начале вызова функции. Оптимизирующий компилятор мог бы заметить, что при `n ≤ 1` сохранять R0 и LR необязательно, т. е. помещать регистры в стек нужно только в части `ELSE`.

На **Рис. 1.14** показан стек в процессе выполнения функции `factorial(3)`. Мы предполагаем, что первоначально SP равен 0xBEFF0FF0, как показано на **Рис. 1.14 (a)**. Функция создает кадр стека длиной два слова для хранения `n` (R0) и LR. При первом вызове `factorial` сохраняет регистр R0 (содержащий `n` = 3) по адресу 0xBEFF0FE8 и LR по адресу 0xBEFF0FEC, как показано на **Рис. 1.14 (b)**. Затем функция изменяет `n` на 2 и рекурсивно вызывает `factorial(2)`, при этом значение LR становится равным 0x8520. При втором вызове функция сохраняет R0 (содержащий `n` = 2) по адресу 0xBEFF0FE0 и LR по адресу 0xBEFF0FE4. В этот раз мы знаем, что LR содержит 0x8520. Затем функция присваивает `n` значение 1 и рекурсивно вызывает `factorial(1)`. При третьем вызове она сохраняет R0 (содержащий `n` = 1) по адресу 0xBEFF0FD8 и LR по адресу 0xBEFF0FDC. На этот раз LR снова содержит 0x8520. Третий вызов `factorial` возвращает значение 1 в регистре R0 и освобождает кадр стека, перед тем как вернуться ко второму вызову. Второй вызов восстанавливает значение `n` = 2 (в регистр R1), восстанавливает 0x8520 в LR (так получилось, что он уже содержит это значение), освобождает кадр стека и возвращает R0 = 2 × 1 = 2 первому вызову. Первый вызов восстанавливает `n` = 3 (в R1), восстанавливает адрес возврата к вызывающей функции в LR, освобождает кадр стека и возвращает R0 = 3 × 2 = 6. На **Рис. 1.14 (c)** показано состояние стека после возврата из рекурсивных вызовов функций. Когда функция `factorial` возвращает управление вызвавшей ее функции, указатель стека оказывается равен исходному значению (0xBEFF0FF0), содержимое стека выше указателя стека не изменилось, и все оберегаемые регистры содержат значения, которые были в них до вызова. Регистр R0 содержит результат вычисления, 6.

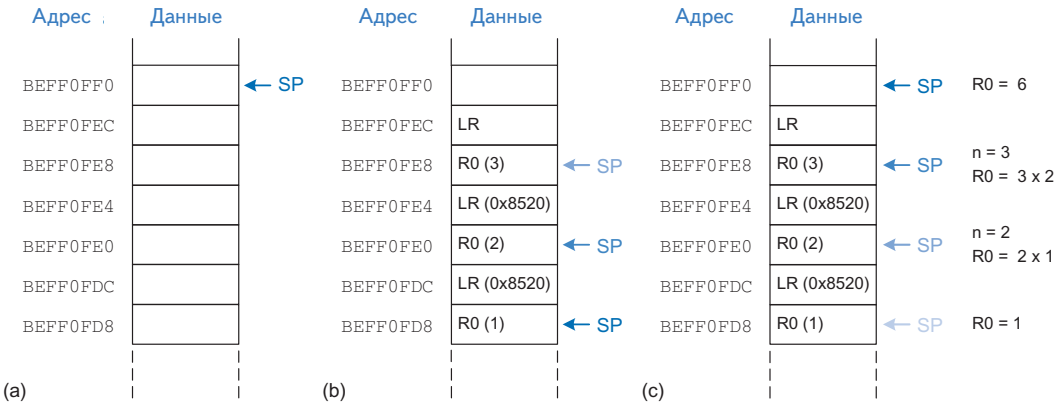


Рис. 1.14. Состояние стека: (a) до, (b) во время и (c) после вызова функции `factorial` при `n` = 3

Дополнительные аргументы и локальные переменные

У функций может быть более четырех аргументов, а также настолько много локальных переменных, что для всех не хватит оберегаемых регистров. Для хранения этой информации используют стек. По соглашениям, принятым в архитектуре ARM, если у функции больше четырех аргументов, то первые четыре, как обычно, передаются в регистрах аргументов. Дополнительные аргументы передаются в стеке и размещаются непосредственно перед указателем стека SP. Вызывающая функция должна расширить стек, выделив место для дополнительных аргументов. На **Рис. 1.15 (a)** показан стек функции, вызывающей функцию, которая принимает более четырех аргументов.

Функция также может объявлять локальные переменные или массивы. Локальные переменные объявляются внутри функции и доступны только ей самой. Локальные переменные хранятся в регистрах R4–R11; если локальных переменных слишком много, их можно хранить в кадре стека функции. В частности, локальные массивы хранятся в стеке.

На **Рис. 1.15 (b)** показана структура стека вызываемой функции. Кадр стека содержит временные регистры и регистр связи (если их необходимо сохранять из-за последующего вызова функции), а также все оберегаемые регистры, которые функция может изменять. Он также содержит локальные массивы и локальные переменные, которым не хватило регистров. Если у вызываемой функции более четырех аргументов, она находит их в кадре стека вызывающей функции. Доступ к дополнительным аргументам – единственный случай, когда функции позволено читать данные из стека за пределами собственного кадра.

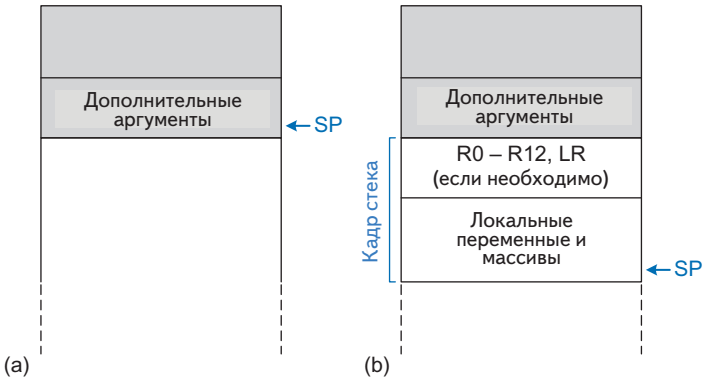


Рис. 1.15. Состояние стека: (a) до, (b) после вызова

1.4. Машинный язык

Язык ассемблера удобен человеку, но цифровые схемы понимают только нули и единицы. Поэтому программу, написанную на языке ассемблера, необходимо преобразовать из последовательности мнемонических команд в последовательность нулей и единиц, которую называют *машинным языком*. В этом разделе описывается машинный язык ARM и кропотливый процесс преобразования языка ассемблера в машинный язык.

В архитектуре ARM используются 32-битовые команды. Для простоты следует придерживаться единообразия, и наиболее единообразным представлением команд в машинном языке было бы такое, где каждая команда занимает ровно одно слово памяти. Для некоторых команд все 32 бита не нужны, но если бы длина команд была переменной, то архитектура оказалась бы сложнее. Для простоты можно было бы также определить единый формат для всех команд, но такой подход обернулся бы серьезными ограничениями. Тем не менее размышления в этом направлении позволяют сформулировать последний принцип проектирования:

> **Четвертый принцип проектирования:** хороший проект требует хороших компромиссов.

В архитектуре ARM в качестве компромисса используются три основных формата команд: **обработки данных**, **доступа к памяти** и **перехода**. Благодаря небольшому количеству форматов удается обеспечить определенное единообразие команд и, как следствие, более простую аппаратную реализацию декодера. При этом разные форматы позволяют учитывать различные потребности. В командах обработки данных имеется первый регистр-источник, второй источник, который может быть непосредственным операндом или регистром, возможно сдвинутым, и регистр-приемник. В командах доступа к памяти три операнда: базовый регистр, смещение, представленное непосредственным операндом или регистром, возможно сдвинутым, и регистр, играющий роль приемника в команде `LDR` или еще одного источника в команде `STR`. В командах перехода имеется один операнд: 24-битовое непосредственное смещение. В этом разделе мы обсудим все форматы команд ARM и покажем, как они кодируются в двоичном виде. В **приложении А** приведен краткий справочник по командам ARMv4.

1.4.1. Команды обработки данных

Формат команд обработки данных наиболее распространенный. Первым операндом-источником является регистр. Второй операнд-источник может быть непосредственным операндом или регистром, возможно, сдви-

нутым. Третий регистр является приемником. На **Рис. 1.16** показан формат команды обработки данных. 32-битовая команда состоит из шести полей: *cond*, *op*, *funct*, *Rn*, *Rd* и *Src2*.

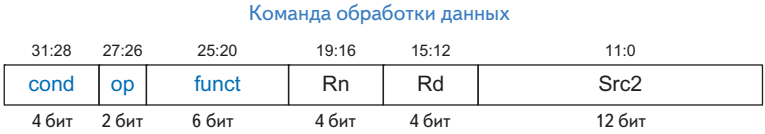


Рис. 1.16. Формат команды обработки данных

Операция, выполняемая командой, закодирована полями, выделенными синим цветом: полем *op* (также называемым *opcode*, или кодом операции) и полем *funct* (кодом функции); в поле *cond* закодирована информация об условном выполнении в зависимости от флагов (см. **раздел 1.3.2**). Напомним, что *cond* = 1110_2 для безусловных команд.

Rd – сокращение от «register destination» (регистр-приемник). *Rn* и *Rm* обозначают первый и второй регистр-источник, хотя интуитивно это неочевидно.

Операнды кодируются тремя полями: *Rn*, *Rd* и *Src2*. *Rn* – первый регистр-источник, *Src2* – второй источник, а *Rd* – регистр-приемник.

На **Рис. 1.17** показаны формат поля *funct* и три варианта поля *Src2* для команд обработки данных. Поле *funct* состоит из трех подполей: *I*, *cmd* и *S*. *I*-бит равен 1, если *Src2* – непосредственный операнд. *S*-бит равен 1, если команда устанавливает флаги условий. Например, для команды `SUBS R1, R9, #11` бит *S* = 1. В *cmd* указана конкретная команда обработки данных (см. **табл. А.1** в **приложении А**). Например, для команды `ADD` *cmd* равно 4 (0100_2), а для `SUB` – 2 (0010_2).

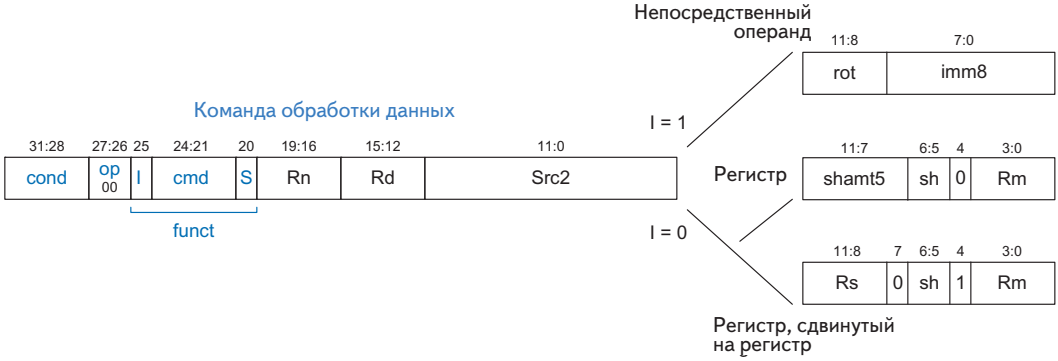


Рис. 1.17. Кодирование поля *funct* и вариантов *Src2* в формате команды обработки данных

Три варианта кодирования *Src2* описывают второй операнд-источник, который может быть: (1) непосредственным операндом, (2) регистром (*Rm*), возможно, сдвинутым на константу (*shamt5*), или (3) регистром (*Rm*), сдвинутым на число бит, хранящееся в другом ре-

Если непосредственный операнд можно закодировать несколькими способами, то выбирается представление с минимальной величиной *rot*. Например, #12 следует представить в виде (*rot*, *imm8*) = (0000, 00001100), а не (0001, 00110000).

гистре (*Rs*). В последних двух случаях *sh* кодирует тип сдвига в соответствии с **Табл. 1.8**.

У команд обработки данных с непосредственным операндом довольно необычное представление, включающее 8-битовое число без знака *imm8* и 4-битовую величину циклического сдвига *rot*: *imm8* циклически сдвигается вправо на 2 × *rot* бит, в результате чего получается 32-битовая константа.

В **Табл. 1.7** приведены примеры циклического сдвига и получающихся 32-битовых констант для 8-битового непосредственного операнда 0xFF. Это представление ценно тем, что позволяет упаковать в небольшое количество бит много полезных констант, в т. ч. небольших кратных любой степени двойки. В **разделе 1.6.1** описано, как генерируются произвольные 32-битовые константы.

Таблица 1.7. Циклические сдвиги и получающиеся 32-битовые константы для *imm8* = 0xFF

rot	32-битовая константа
0000	0000 0000 0000 0000 0000 0000 1111 1111
0001	1100 0000 0000 0000 0000 0000 0011 1111
0010	1111 0000 0000 0000 0000 0000 0000 1111
...	...
1111	0000 0000 0000 0000 0000 0011 1111 1100

Таблица 1.8. Кодирование поля *sh*

Команда	sh	Операция
LSL	00_2	Логический сдвиг влево
LSR	01_2	Логический сдвиг вправо
ASR	10_2	Арифметический сдвиг вправо
ROR	11_2	Циклический сдвиг вправо

На **Рис. 1.18** показан машинный код команд ADD и SUB, когда *Src2* является регистром. Самый простой способ транслировать ассемблерный код в машинный состоит в том, чтобы выписать значения каждого поля, а затем перевести их в двоичную систему счисления. Чтобы получить более компактное машинное представление, разбейте биты на группы по четыре и замените каждую группу шестнадцатеричной цифрой. Не забывайте, что регистр-приемник на ассемблере является первым операндом, а в машинной команде – вторым регистровым полем (*Rd*). *Rn* и *Rm* – соответственно первый и второй операнды-источ-

ники. Например, для ассемблерной команды `ADD R5, R6, R7` имеем *Rn* = 6, *Rd* = 5, *Rm* = 7.

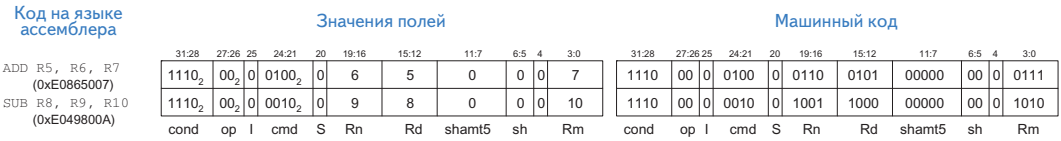


Рис. 1.18. Команды обработки данных с тремя регистровыми операндами

На **Рис. 1.19** показан машинный код команд `ADD` и `SUB` с одним непосредственным и двумя регистровыми операндами. И снова регистр-приемник – первый операнд на языке ассемблера, но второе регистровое поле (*Rd*) в машинной команде. Непосредственный операнд в команде `ADD` (42) можно закодировать 8 битами, поэтому циклический сдвиг не нужен (*imm8* = 42, *rot* = 0). Но в команде `SUB R2, R3, 0xFF0` непосредственный операнд не помещается в 8 бит *imm8*. Вместо этого мы полагаем *imm8* равным 255 (0xFF) и циклически сдвигаем вправо на 28 бит (*rot* = 14). Интерпретировать будет проще, если вспомнить, что циклический сдвиг вправо на 28 бит эквивалентен циклическому сдвигу влево на 32 – 28 = 4 бита.

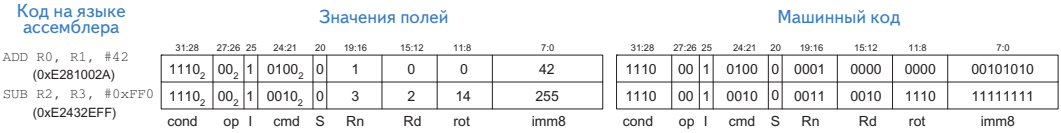


Рис. 1.19. Команды обработки данных с одним непосредственным и двумя регистровыми операндами

Сдвиги тоже считаются командами обработки данных. Напомним (см. **раздел 1.3.1**), что величину сдвига можно закодировать 5 битами в непосредственном операнде или в регистре.

На **Рис. 1.20** показан машинный код команд логического сдвига влево (`LSL`) и циклического сдвига вправо (`ROR`), в которых величина сдвига задана непосредственно. Для всех команд сдвига поле *cmd* равно 13 (1101_2), а в поле *sh* кодируется тип сдвига в соответствии с **Табл. 1.8**. В поле *Rm* (т. е. R5) находится 32-битовое значение, подлежащее сдвигу, а в поле *shamt5* – величина сдвига в битах. Результат сдвига помещается в регистр *Rd*. Поле *Rn* не используется и должно быть равно 0.

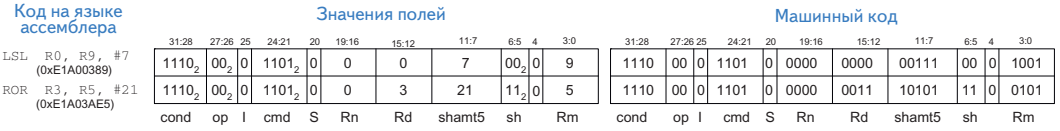


Рис. 1.20. Команды сдвига с непосредственно заданной величиной сдвига

На **Рис. 1.21** показан машинный код команд `LSR` и `ASR`, когда величина сдвига закодирована в 8 младших битах *Rs* (R6 и R12). Как и раньше, *cmd* равно 13 (1101_2), *sh* кодирует тип сдвига, *Rm* содержит значение, подлежащее сдвигу, а результат сдвига помещается в *Rd*. В этой команде используется режим адресации *регистр – сдвиговый регистр*, когда один регистр (*Rm*) сдвигается на величину, хранящуюся во втором регистре (*Rs*). Поскольку используются 8 младших бит *Rs*, регистр *Rm* можно сдвинуть не более чем на 255 позиций. Например, если в *Rs* находится значение 0xF001001C, то величина сдвига будет равна 0x1C (28). В случае логического сдвига более чем на 31 бит все биты «выталкиваются», и остаются одни нули. Циклический сдвиг на 50 бит эквивалентен сдвигу на 18 бит.

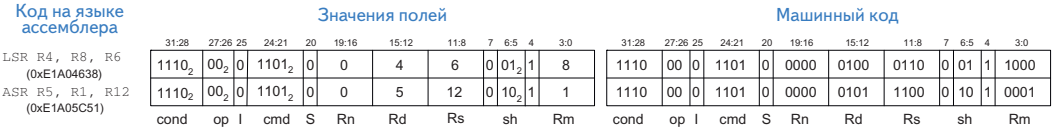


Рис. 1.21. Команды сдвига с величиной сдвига, заданной в регистре

1.4.2. Команды доступа к памяти

Формат команд доступа к памяти похож на формат команд обработки данных, в нем те же шесть полей: *cond*, *op*, *funct*, *Rn*, *Rd* и *Src2*, показанных на **Рис. 1.22**. Однако поле *funct* кодируется по-другому, у поля *Src2* два варианта, а код операции *op* равен 01_2. *Rn* – регистр, содержащий базовый адрес, в *Src2* находится смещение, а *Rd* – регистр-приемник в командах загрузки и регистр-источник в командах сохранения. Смещение может задаваться либо 12-битовым непосредственным операндом (*imm12*), либо регистром (*Rm*), возможно, сдвинутым на константу (*shamt5*). Поле *funct* содержит шесть управляющих битов: $\overline{I}$, *P*, *U*, *B*, *W* и *L*. Биты *I* (непосредственно) и *U* (сложение) определяют, как задано смещение: непосредственно или в регистре, – и что с ним делать: прибавлять к базе или вычитать (см. **Табл. 1.9**). Биты *P* (предындексация) и *W* (обратная запись) определяют режим индексации в соответствии с **Табл. 1.10**. Биты *L* (загрузка) и *B* (байт) определяют тип операции доступа к памяти в соответствии с **Табл. 1.11**.

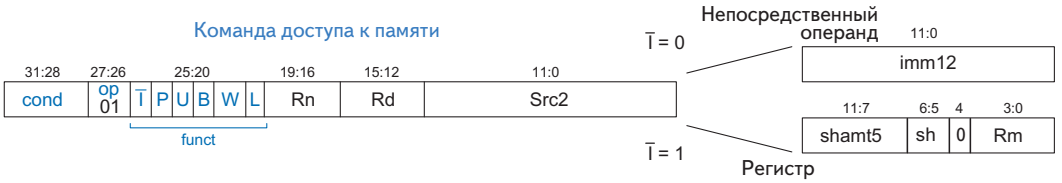


Рис. 1.22. Формат команды доступа к памяти (`LDR`, `STR`, `LDRB` и `STRB`)

Таблица 1.9. Биты, управляющие типом смещения в командах доступа к памяти

Бит	Назначение	
	$\overline{I}$	U
0	Непосредственное смещение в Src2	Вычесть смещение из базы
1	Регистровое смещение в Src2	Прибавить смещение к базе

Таблица 1.10. Биты, управляющие типом смещения в командах доступа к памяти

P	W	Режим индексации
0	0	Постиндексация
0	1	Не поддерживается
1	0	Смещение
1	1	Предындексация

Таблица 1.11. Биты, управляющие типом операции в командах доступа к памяти

L	B	Команда
0	0	`STR`
0	1	`STRB`
1	0	`LDR`
1	1	`LDRB`

Пример 1.3. ТРАНСЛЯЦИЯ КОМАНД ДОСТУПА К ПАМЯТИ В МАШИННЫЙ КОД

Транслируйте на машинный язык следующую команду на языке ассемблера:

```
STR R11, [R5], #-26
```

Решение. `STR` – команда доступа к памяти, поэтому `op` равно 01_2. Согласно **Табл. 1.11**, для команды `STR` $L = 0$ и $B = 0$. В команде используется режим постиндексации, поэтому, согласно **Табл. 1.10**, $P = 0$ и $W = 0$. Непосредственное смещение вычитается из базы, поэтому $\overline{I} = 0$ и $U = 0$. На **Рис. 1.23** показаны все поля и машинный код: 0xE405B01A.

Обратите внимание на противоречащее интуиции кодирование режима постиндексации.

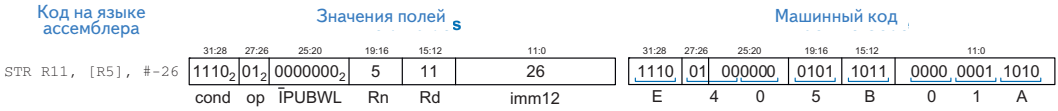


Рис. 1.23. Машинный код команды доступа к памяти из примера 1.3

1.4.3. Команды перехода

В командах перехода есть только один 4-битовый непосредственный операнд со знаком, *imm24*, как показано на **Рис. 1.24**. Как и команды обработки данных и доступа к памяти, команды перехода начинаются

4-битовым полем условия и 2-битовым кодом операции *op*, равным 10_2. Поле `funct` состоит всего из 2 бит. Старший бит *funct* для команд перехода всегда равен 1, а младший, *L*, обозначает тип перехода: 1 для команды `BL` и 0 для команды `B`. Остальные 24 бита содержат поле *imm24* в дополнительном коде, оно используется для задания конечного адреса относительно PC + 8.

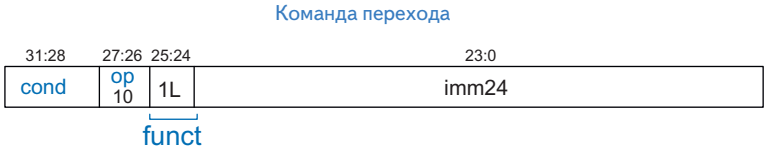


Рис. 1.24. Формат команды перехода

В **примере кода 1.28** показана команда *перейти, если меньше* (`BLT`), а на **Рис. 1.25** – ее машинный код. *Конечный адрес перехода* (branch target address, BTA) – это адрес следующей команды, в случае если производится переход. В команде `BLT` на **Рис. 1.25** конечный адрес равен 0x80B4, т. е. адресу метки `THERE`.

Пример кода 1.28. ВЫЧИСЛЕНИЕ КОНЕЧНОГО АДРЕСА ПЕРЕХОДА

Код на языке ассемблера ARM

```
0x80A0 BLT THERE
0x80A4 ADD R0, R1, R2
0x80A8 SUB R0, R0, R9
0x80AC ADD SP, SP, #8
0x80B0 MOV PC, LR
0x80B4 THERE   SUB R0, R0, #1
0x80B8 ADD R3, R3, #0x5
```

Рис. 1.25. Машинный код команды `BLT`

24-битовый непосредственный операнд задает число команд между BTA и PC + 8 (две команды после команды перехода). В данном случае непосредственное значение в поле *imm24* команды `BLT` равно 3, потому что конечный адрес (0x80B4) находится на три команды дальше PC + 8 (0x80A8).

Для вычисления адреса BTA процессор выполняет следующие действия: расширить 24-битовый непосредственный операнд со знаком, сдвинуть результат влево на 2 (для преобразования слов в байты) и прибавить к PC + 8.

Пример 1.4. ВЫЧИСЛЕНИЕ НЕПОСРЕДСТВЕННОГО ПОЛЯ ДЛЯ АДРЕСАЦИИ ОТНОСИТЕЛЬНО PC

Вычислите поле imm24 и машинный код команды перехода в следующей ассемблерной программе:

```
0x8040 TEST      LDRB R5, [R0, R3]
0x8044           STRB R5, [R1, R3]
0x8048           ADD  R3, R3, #1
0x8044           MOV  PC, LR
0x8050           BL   TEST
0x8054           LDR  R3, [R1], #4
0x8058           SUB  R4, R3, #9
```

Решение. На **Рис. 1.26** показан машинный код команды перейти и связать (BL). В ней конечный адрес перехода (0x8040) на шесть команд раньше PC + 8 (0x8058), поэтому поле *imm24* равно −6.

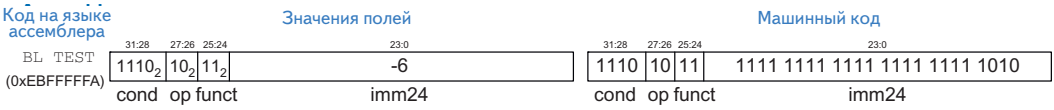


Рис. 1.26. Машинный код команды BL

1.4.4. Режимы адресации

В этом разделе описаны режимы адресации, применяемые в различных командах. В ARM есть четыре основных режима: регистровый, непосредственный, базовый и относительно счетчика команд (PC-relative). В большинстве других архитектур имеются аналогичные режимы адресации, поэтому, разобравшись в них, вы упростите себе изучение других языков ассемблера. В регистровом и базовом режимах имеется несколько подрежимов, все они описаны ниже. Первые три режима (регистровый, непосредственный и базовый) определяют способы чтения и записи операндов, а последний (относительно счетчика команд) – режим записи счетчика команд (PC). В **Табл. 1.12** приведены примеры всех режимов адресации.

В командах обработки данных используется регистровая или непосредственная адресация, когда первый операнд-источник является регистром, а второй – регистром либо непосредственным операндом соответственно. В ARM второй регистр можно дополнительно сдвигать на величину, заданную непосредственно или в третьем регистре. В командах доступа к памяти используется базовая адресация, причем базовый адрес задается в регистре, а смещение может быть зада-

ARM отличается от других RISC-архитектур тем, что допускает сдвиг второго операнда-источника в регистровом и базовом режимах адресации. Это означает, что в аппаратной реализации должен присутствовать сдвиговый регистр, включенный последовательно с АЛУ, зато при этом существенно уменьшается длина кода в типичных программах, особенно тех, где производится доступ к массивам. Например, если массив содержит 32-битовые элементы, то для вычисления байтового смещения от начала массива нужно сдвинуть индекс массива влево на 2. Допускаются сдвиги любого типа, но чаще всего используется сдвиг влево с целью умножения.

но непосредственно в регистре или в регистре, сдвинутом на непосредственно заданную величину. В командах перехода используется адресация относительно счетчика команд, когда конечный адрес перехода вычисляется путем сложения смещения с PC + 8.

Таблица 1.12. Режимы адресации в архитектуре ARM

Режим адресации	Пример	Описание
Регистровая		
Только регистр	`ADD R3, R2, R1`	`R3 ← R2 + R1`
Регистр, сдвинутый на константу	`SUB R4, R5, R9, LSR #2`	`R4 ← R5 – (R9 >> 2)`
Регистр, сдвинутый на регистр	`ORR R0, R10, R2, ROR R7`	`R0 ← R10 | (R2 ROR R7)`
Непосредственная	`SUB R3, R2, #25`	`R3 ← R2 – 25`
Базовая		
Непосредственное смещение	`STR R6, [R11, #77]`	`mem[R11+77] ← R6`
Смещение в регистре	`LDR R12, [R1, –R5]`	`R12 ← mem[R1 – R5]`
Смещение в сдвинутом регистре	`LDR R8, [R9, R2, LSL #2]`	`R8 ← mem[R9 + (R2 << 2)]`
Относительно счетчика команд	`B LABEL1`	Перейти на `LABEL1`

1.4.5. Интерпретация кода на машинном языке

Чтобы интерпретировать код на машинном языке, необходимо декодировать поля каждого 32-битового слова команды. Форматы команд различаются, но любая команда начинается 4-битовым полем *cond* и 2-битовым полем *op*. Начинать лучше всего с анализа поля *op*. Если оно равно 00_2, то мы имеем команду обработки данных; если 01_2 – то команду доступа к памяти, а если 10_2 – то команду перехода. После этого можно интерпретировать все остальные поля.

Пример 1.5. ТРАНСЛЯЦИЯ МАШИННОГО ЯЗЫКА НА ЯЗЫК АССЕМБЛЕРА

Транслируйте следующий код на машинном языке в код на языке ассемблера.

```
0xE0475001
0xE5949010
```

Решение. Сначала представим каждую команду в двоичном виде и выделим биты 27:26, содержащие код операции *op* (см. **Рис. 1.27**). Поле *op* равно 00_2 и 01_2, т. е. мы имеем команду обработки данных и команду доступа к памяти соответственно. Затем проанализируем поле *funct* в каждой команде.

Поле *cmd* в команде обработки данных равно 2 (0010_2), а *I*-бит (бит 25) равен 0. Это означает, что мы имеем дело с командой SUB с регистром в качестве *Src2*. *Rd* равно 5, *Rn* – 7, *Rm* – 1.

Поле *funct* в команде доступа к памяти содержит 011001_2. $B = 0$ и $L = 1$, т. е. это команда LDR. $P = 1$ и $W = 0$, значит, это адресация со смещением. $\overline{I} = 0$, т. е. смещение задано непосредственно. $U = 1$, т. е. смещение прибавляется к базе. Таким образом, мы имеем команду загрузки регистра с непосредственным смещением, прибавляемым к базовому регистру. *Rd* равно 9, *Rn* – 4, *imm12* – 16. На **Рис. 1.27** показан ассемблерный код, эквивалентный этим двум машинным командам.

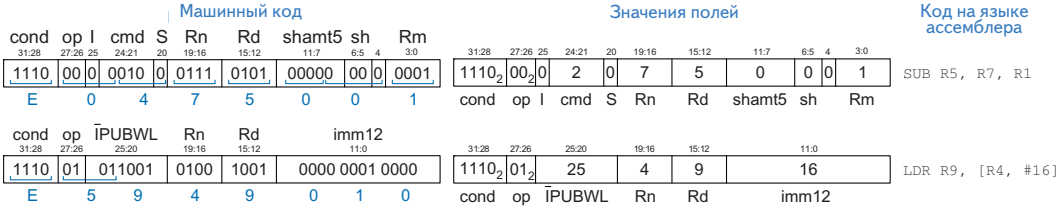


Рис. 1.27. Трансляция машинного кода в код на языке ассемблера

1.4.6. Могущество хранимой программы

Программа, написанная на машинном языке, – это последовательность 32-битовых чисел, представляющих команды. Как и любые двоичные числа, эти команды можно хранить в памяти. Эта концепция называется *хранимой программой*, именно в ней заключается главная причина могущества компьютеров. Для запуска новой программы не нужно тратить много времени и усилий на изменение или переконфигурацию аппаратного обеспечения, необходимо лишь записать новую программу в память. Хранимые программы, в отличие от специализированного аппаратного обеспечения, способны выполнять *вычисления общего характера*. Таким образом, компьютер может выполнять любые приложения, начиная от простого калькулятора и заканчивая текстовыми процессорами и проигрывателями видео, – нужно лишь изменить хранимую программу.

В хранимой программе команды считываются, или *выбираются* из памяти, и выполняются процессором. Даже большие и сложные программы представляют собой просто последовательность операций чтения из памяти и выполнения команд.

Ада Лавлейс
1815–1852

Британская женщина-математик, написавшая первую компьютерную программу. Программа предназначалась для вычисления чисел Бернулли на аналитической машине Чарльза Бэббиджа. Ада была дочерью поэта лорда Байрона.

На **Рис. 1.28** показано, как машинные команды хранятся в памяти. В программах для ARM команды обычно хранятся, начиная с младших адресов, в данном случае 0x00008000. Напомним, что адресация памяти в архитектуре ARM побайтовая, поэтому 32-битовые (4-байтовые) адреса команд кратны четырем байтам, а не одному.

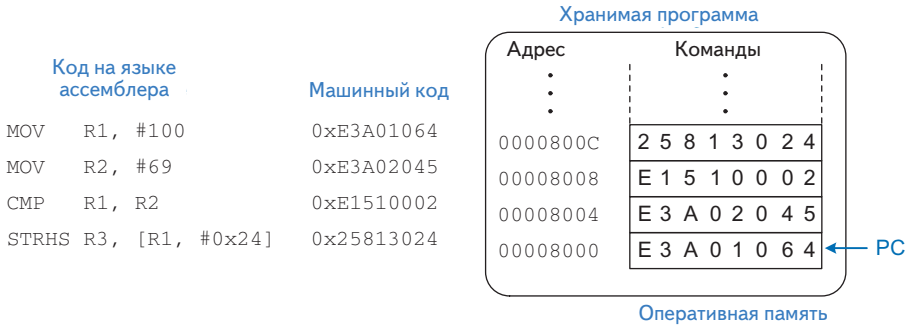


Рис. 1.28. Хранимая программа

Чтобы запустить, или выполнить, хранимую программу, процессор последовательно выбирает ее команды из памяти. Далее выбранные команды декодируются и выполняются оборудованием. Адрес текущей команды хранится в 32-битовом регистре, называемом счетчиком команд (program counter, PC), в роли которого выступает регистр R15. В силу исторических причин операция чтения счетчика команд возвращает адрес текущей команды плюс 8.

Для того чтобы выполнить код, показанный на **Рис. 1.28**, счетчик команд инициализируется значением 0x00008000. Процессор читает из памяти по этому адресу команду 0xE3A01064 (`MOV R1, #100`) и выполняет ее. Затем процессор увеличивает значение счетчика команд на 4 (оно становится равным 0x00008004), выбирает из памяти и выполняет новую команду, после чего процесс повторяется.

Под *архитектурным состоянием* микропроцессора понимается состояние программы. В случае ARM архитектурное состояние включает содержимое регистрового файла и регистры состояния. Если операционная система сохранит архитектурное состояние в какой-то точке программы, то она сможет эту программу прервать, заняться чем-то другим, а потом восстановить архитектурное состояние, после чего прерванная программа продолжит выполняться, даже не узнав, что ее вообще прерывали. Архитектурное состояние будет играть важную роль, когда мы приступим к созданию микропроцессора в **главе 2**.

1.5. Свет, камера, мотор! Компилируем, ассемблируем и загружаем

До сих пор мы видели, как небольшие фрагменты кода, написанного на языке высокого уровня, транслируются в ассемблерный и машинный код. В этом разделе мы опишем, как происходят компиляция и ассемблирование целой программы на языке высокого уровня и как эта программа загружается в память компьютера для выполнения. Начнем с рассмотрения *карты памяти* ARM, в которой определяется расположение кода, данных и стека в памяти.

На **Рис. 1.29** показаны шаги трансляции программы с языка высокого уровня на машинный язык и ее последующего запуска. Прежде всего *компилятор* транслирует программу с языка высокого уровня на язык ассемблера. *Ассемблер* транслирует ассемблерный код в машинный и записывает результат в объектный файл. *Компоновщик* объединяет машинный код с кодом из библиотек и других файлов и вычисляет адреса переходов и переменных, формируя файл исполняемой программы. На практике большинство компиляторов выполняют все три шага: компиляцию, ассемблирование и компоновку. Наконец, *загрузчик* загружает программу в памяти и начинает ее выполнение. Далее в этом разделе мы разберем все эти шаги для простой программы.

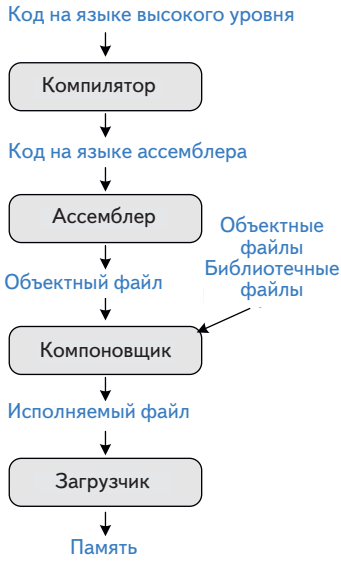


Рис. 1.29. Шаги трансляции и запуска программы

1.5.1. Карта памяти

Так как в архитектуре ARM используются 32-битовые адреса, размер адресного пространства составляет 2^{32} байта = 4 гигабайта (ГБ). Адреса слов кратны 4 и располагаются в промежутке от 0 до 0xFFFFFFFC. На **Рис. 1.30** изображена карта памяти. Адресное пространство разделено на пять частей, или сегментов: сегмент кода, сегмент глобальных данных, сегмент динамических данных и сегменты для обработчиков исключений, операционной системы (ОС) и ввода-вывода. Эти сегменты рассматриваются в следующих разделах.

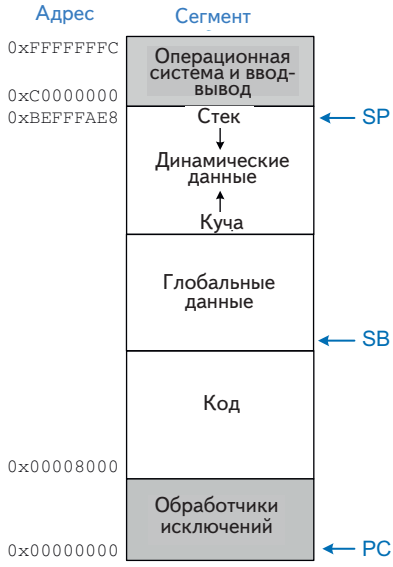


Рис. 1.30. Примеры карты памяти в архитектуре ARM

Грейс Хоппер
1906-1992

Окончила Йельский университет со степенью доктора философии по математике. Во время работы в компании Remington Rand Corporation разработала первый компилятор. Сыграла важную роль в разработке языка программирования COBOL. Будучи офицером ВМФ, получила множество наград, в том числе медаль за победу во Второй мировой войне и медаль «За примерную действительную службу в ВС США».

Сегмент кода

Сегмент кода (text segment) содержит машинные команды исполняемой программы. В ARM его называют также *постоянным сегментом* (read-only segment). Помимо кода, он может содержать литералы (константы) и данные, предназначенные только для чтения.

Сегмент глобальных данных

Сегмент глобальных данных содержит глобальные переменные, которые, в отличие от локальных переменных, находятся в области видимости всех функций программы. Глобальные переменные инициализируются при загрузке программы, но до начала ее выполнения. В ARM этот сегмент называют также *сегментом чтения-записи*. Доступ к глобальным переменным обычно осуществляется с помощью *статического базового* регистра (SB), который указывает на начала глобального сегмента. В ARM принято соглашение об использовании в этом качестве регистра R9.

Сегмент динамических данных

Сегмент динамических данных содержит стек и кучу. В момент запуска программы этот сегмент не содержит данных – они динамически выделяются и освобождаются в процессе выполнения.

В момент запуска операционная система устанавливает указатель стека (SP), так чтобы он указывал на вершину стека. Обычно стек растет вниз, как показано на рисунке выше. В стеке хранятся временные данные и локальные переменные, например массивы, для которых не хватило регистров. Как обсуждалось в **разделе 1.3.7**, функции также используют стек для сохранения и восстановления регистров. Доступ к кадрам стека осуществляется в порядке последним пришел – первым ушел.

В *куче* (heap) хранятся данные, динамически выделяемые программой во время работы. В языке C выделение памяти осуществляется функцией `malloc`; в C++ и Java для этого служит оператор `new`. По аналогии с кучей одежды, разбросанной по полу в спальне, данные, находящиеся в куче, можно использовать и отбрасывать в произвольном порядке. Куча обычно растет вверх от нижней границы сегмента динамических данных.

Если стек и куча прорастут друг в друга, данные программы могут быть повреждены. Распределитель памяти стремится избежать этой ситуации. Он возвращает ошибку нехватки памяти (out-of-memory error), если памяти недостаточно для размещения новых динамических данных.

Сегменты обработчиков исключений, ОС и ввода-вывода

Нижняя часть карты памяти в ARM, начиная с адреса 0x0, зарезервирована для таблицы векторов исключений и обработчиков исключений (см. **раздел 1.6.3**), а верхняя часть – для операционной системы и ввода-вывода, отображенного на память (см. **раздел 4.2** (книга 1)).

1.5.2. Компиляция

Компилятор транслирует код высокого уровня в код на языке ассемблера. Примеры в этом разделе ориентированы на GCC, популярный и широко используемый свободный компилятор, работающий в том числе на одноплатном компьютере Raspberry Pi (см. **раздел 4.3** (книга 1)). В **примере кода 1.29** показана простая программа на языке высокого уровня, содержащая три глобальные переменные и две функции, а также ассемблерный код, сгенерированный GCC.

В **примере кода 1.29** для доступа к глобальным переменным применяются две команды: одна загружает адрес переменной, а вторая читает или записывает переменную по этому адресу. Адреса глобальных переменных размещаются после кода, начиная с метки `.L3`. Команда `LDR R3, .L3` загружает *адрес* `f` в регистр `R3`, а команда `STR R0, [R3, #0]` производит запись в `f`; команда `LDR R3, .L3+4` загружает адрес `g` в `R3`, а `STR R1, [R3, #0]` производит запись в `g` и т. д. В **разделе 1.6.1** мы продолжим обсуждение этой ассемблерной конструкции.

Пример кода 1.29. КОМПИЛЯЦИЯ ПРОГРАММЫ НА ЯЗЫКЕ ВЫСОКОГО УРОВНЯ

Код на языке высокого уровня

```
int f, g, y; // глобальные переменные


int sum(int a, int b) {
  return (a + b);
}


int main(void)
{
  f = 2;
  g = 3;
  y = sum(f, g);
  return y;
}
```

Код на языке ассемблера ARM

```
   .text
   .global sum
   .type sum, %function
sum:
   add     r0, r0, r1
   bx      lr
   .global main
   .type   main, %function
main:
   push    {r3, lr}
   mov     r0, #2
   ldr     r3, .L3
   str     r0, [r3, #0]
   mov     r1, #3
   ldr     r3, .L3+4
   str     r1, [r3, #0]
   bl      sum
   ldr     r3, .L3+8
   str     r0, [r3, #0]
   pop     {r3, pc}
.L3:
   .word   f
   .word   g
   .word   y
```

Чтобы откомпилировать, ассемблировать и скомпоновать программу `prog.c` на языке C с помощью GCC, введите команду

```
gcc -O1 -g prog.c -o prog
```

Эта команда порождает выходной исполняемый файл `prog`. Флаг `-O1` означает, что компилятор должен произвести минимальную оптимизацию, а не оставлять совсем уж неэффективный код. Флаг `-g` просит компилятор включить в файл отладочную информацию.

Чтобы видеть промежуточные шаги, можно задать флаг `-S`, означающий, что GCC должен откомпилировать программу, но не выполнять ассемблирование и компоновку.

```
gcc -O1 -S prog.c -o prog.s
```

Выходной файл `prog.s` довольно длинный, но его интересные части показаны в **примере кода 1.29**. Отметим, что GCC генерирует метки, заканчивающиеся двоеточием. Весь сгенерированный код записан строчными буквами, и в нем встречаются ассемблерные директивы, которые мы не обсуждали. Обратите внимание, что возврат из функции `sum` производится командой `BX`, а не `MOV PC, LR`. Кроме того, отметим, что GCC решил сохранять и восстанавливать регистр R3, хотя он не относится к оберегаемым. Адреса глобальных переменных хранятся в таблице, начинающейся меткой .L3.

1.5.3. Ассемблирование

Ассемблер преобразует код на языке ассемблера в *объектный файл*, содержащий код на машинном языке. GCC может создать объектный файл как из файла `prog.s`, так и непосредственно из файла `prog.c`. Для этого служат соответственно команды

```
gcc -c prog.s -o prog.o
```

или

```
gcc -O1 -g -c prog.c -o prog.o
```

Ассемблер выполняет два прохода по файлу. На первом проходе он назначает адреса командам и находит все символы, например метки и имена глобальных переменных. Имена и адреса символов хранятся в *таблице символов*. На втором проходе ассемблер генерирует машинный код. Адреса меток берутся из таблицы символов. Код на машинном языке и таблица символов сохраняются в объектном файле.

Объектный файл можно *дизассемблировать* командой `objdump` и узнать, какой ассемблерный код соответствует машинному. Если код

был откомпилирован с флагом `-g`, то дизассемблер покажет заодно и соответствующие строки на C:

```
objdump -S prog.o
```

Ниже показан результат дизассемблирования секции `.text`:

```
00000000 <sum>:
int sum(int a, int b) {
             return (a + b);
}
  0:  e0800001 add r0, r0, r1
  4:  e12fff1e bx  lr

00000008 <main>:

int f, g, y; // глобальные переменные

int sum(int a, int b);

int main(void) {
  8:  e92d4008 push {r3, lr}
      f = 2;
  c:  e3a00002 mov r0, #2
 10:  e59f301c ldr r3, [pc, #28] ; 34 <main+0x2c>
 14:  e5830000 str r0, [r3]
      g = 3;
 18:  e3a01003 mov r1, #3
 1c:  e59f3014 ldr r3, [pc, #20] ; 38 <main+0x30>
 20:  e5831000 str r1, [r3]
      y = sum(f,g);
 24:  ebffffe bl 0 <sum>
 28:  e59f300c ldr r3, [pc, #12] ; 3c <main+0x34>
 2c:  e5830000 str r0, [r3]
      return y;
}
 30: e8bd8008 pop {r3, pc}
 ...
```

Напомним (см. **раздел 1.4.6**), что чтение счетчика команд возвращает адрес текущей команды плюс 8. Следовательно, команда `LDR R3, [PC, #28]` загружает адрес переменной `f`, которая размещена сразу после кода: (PC + 8) + 28 = (0x10 + 0x8) + 0x1C = 0x34.

Команда `objdump` с флагом `-t` выводит таблицу символов, хранящуюся в объектном файле. Ниже показаны ее наиболее интересные части. Отметим, что функция `sum` начинается по адресу 0, а ее длина равна 8. Функция `main` начинается по адресу 8, ее длина равна 0x38. Перечислены также символы глобальных переменных `f`, `g` и `h`, каждый длиной 4 байта, но адреса им еще не назначены.

```
objdump -t prog.o

SYMBOL TABLE:
```

```
00000000 l d .text 00000000 .text
00000000 l d .data 00000000 .data
00000000 g F .text 00000008 sum
00000008 g F .text 00000038 main
00000004   O *COM* 00000004 f
00000004   O *COM* 00000004 g
00000004   O *COM* 00000004 y
```

1.5.4. Компоновка

Большие программы обычно содержат несколько файлов. Если программист изменяет только один из этих файлов, то заново компилировать и ассемблировать все остальные файлы – лишняя трата времени. В частности, программы нередко вызывают функции из библиотечных файлов, которые почти никогда не меняются. А если код на языке высокого уровня не изменился, то соответствующий ему объектный файл не нуждается в обновлении. Кроме того, в программе обычно имеется начальный код, который инициализирует стек, кучу и прочее и должен выполняться до вызова функции `main`.

Работа компоновщика заключается в том, чтобы объединить все объектные файлы и начальный код в один *исполняемый* файл, содержащий машинный код. Компоновщик перемещает данные и команды в объектных файлах так, чтобы они не наслаивались друг на друга. Он использует информацию из таблиц символов для коррекции адресов перемещаемых глобальных переменных и меток. Для компоновки объектного файла вызовите GCC следующим образом:

```
gcc prog.o -o prog
```

Исполняемый файл также можно дизассемблировать командой:

```
objdump -S -t prog
```

Начальный код слишком длинный, показывать его мы не будем, но код нашей программы начинается по адресу 0x8390, а глобальным переменным назначены адреса в глобальном сегменте, начиная с 0x10570. Обратите внимание на ассемблерные директивы `.word`, которые определяют адреса глобальных переменных `f`, `g` и `y`.

```
00008390 <sum>:

int sum(int a, int b) {
      return (a + b);
}
  8390:  e0800001 add r0, r0, r1
  8394:  e12fff1e bx lr
```

```
00008398 <main>:

int f, g, y; // глобальные переменные

int sum(int a, int b);

int main(void) {
  8398:  e92d4008 push {r3, lr}
  f = 2;
  839c:  e3a00002 mov r0, #2
  83a0:  e59f301c ldr r3, [pc, #28] ; 83c4 <main+0x2c>
  83a4:  e5830000 str r0, [r3]
  g = 3;
  83a8:  e3a01003 mov r1, #3
  83ac:  e59f3014 ldr r3, [pc, #20] ; 83c8 <main+0x30>
  83b0:  e5831000 str r1, [r3]
  y = sum(f,g);
  83b4:  ebfffff5 bl 8390 <sum>
  83b8:  e59f300c ldr r3, [pc, #12] ; 83cc <main+0x34>
  83bc:  e5830000 str r0, [r3]
  return y;
}
  83c0:  e8bd8008 pop {r3, pc}
  83c4:  00010570 .word 0x00010570
  83c8:  00010574 .word 0x00010574
  83cc:  00010578 .word 0x00010578
```

Команда `LDR R3, [PC, #28]` в исполняемом файле загружает слово по адресу (PC + 8) + 28 = (0x83A0 + 0x8) + 0x1C = 0x83C4. По этому адресу находится значение 0x10570 — адрес глобальной переменной `f`.

Исполняемый файл содержит также обновленную таблицу символов с перемещенными адресами функций и глобальных переменных.

```
SYMBOL TABLE:
000082e4 l    d  .text 00000000    .text
00010564 l    d  .data 00000000    .data
00008390 g     F .text 00000008    sum
00008398 g     F .text 00000038    main
00010570 g     O .bss  00000004    f
00010574 g     O .bss  00000004    g
00010578 g     O .bss  00000004    y
```

1.5.5. Загрузка

Операционная система загружает программу, считывая сегмент кода из исполняемого файла на запоминающем устройстве (обычно это жесткий диск) в сегмент кода в памяти. Операционная система выполняет переход в начало программы, начиная тем самым ее выполнение. На **Рис. 1.31** показана карта памяти в начале выполнения программы.

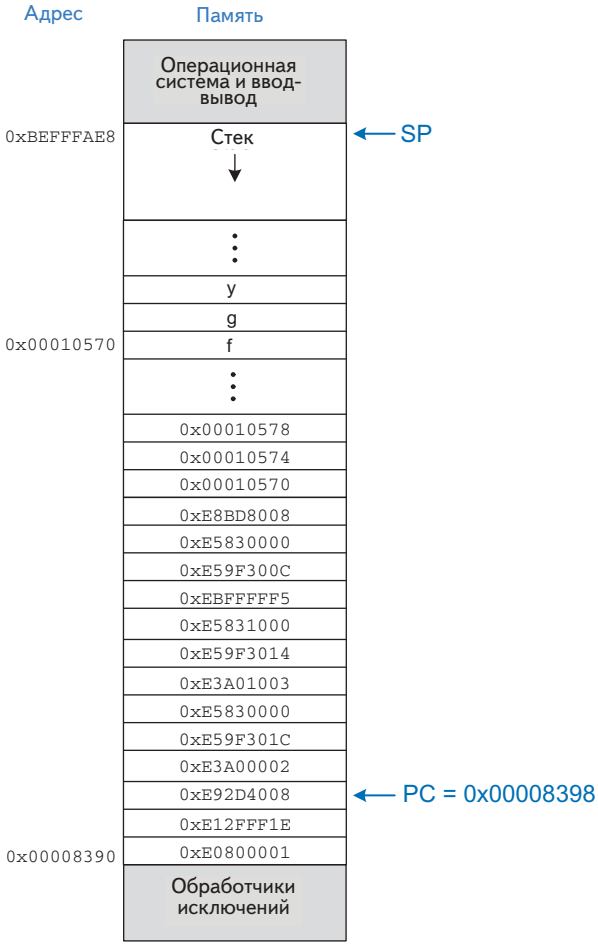


Рис. 1.31. Исполняемый файл, загруженный в память

1.6. Дополнительные сведения

В этом разделе рассматривается несколько дополнительных тем, не нашедших отражения в других местах этой главы. Речь пойдет о загрузке 32-битовых литералов, командах NOP и исключениях.

1.6.1. Загрузка литералов

Во многих программах необходимо загружать 32-битовые литералы, например константы и адреса. Команда `MOV` принимает только 12-битовый источник, поэтому для загрузки таких чисел из *пула литералов* в сег-

менте кода применяется команда LDR. Ассемблеры ARM понимают команды загрузки вида:

```
LDR Rd, =literal
LDR Rd, =label
```

Первая команда загружает 32-битовую константу literal, вторая – адрес переменной или указатель, определяемый меткой label. В обоих случаях значение, подлежащее загрузке, находится в *пуле литералов* – части сегмента кода, отведенной для литералов. Пул литералов должен отстоять от команды LDR меньше, чем на 4096 байтов, чтобы загрузку можно было выполнить командой LDR Rd, [PC, #offset_to_literal]. Программа должна обходить пул литералов, поскольку попытка выполнить литерал как команду бессмысленна, если не хуже.

В **примере кода 1.30** иллюстрируется загрузка литерала. Предположим (см. **Рис. 1.32**), что команда LDR находится по адресу 0x8110, а литерал – по адресу 0x815C. Напомним, что чтение счетчика команд (PC) возвращает адрес текущей команды плюс 8 байтов. Поэтому при выполнении команды LDR чтение PC вернет 0x8118. Следовательно, для поиска пула литералов в LDR используется смещение 0x44: LDR R1, [PC, #0x44].

Пример кода 1.30. ЗАГРУЗКА С ИСПОЛЬЗОВАНИЕМ ПУЛА ЛИТЕРАЛОВ

Код на языке высокого уровня

```
int a = 0x2B9056F;
```

Код на языке ассемблера ARM

```
; R1 = a
  LDR R1, =0x2B9056F
  ...
```

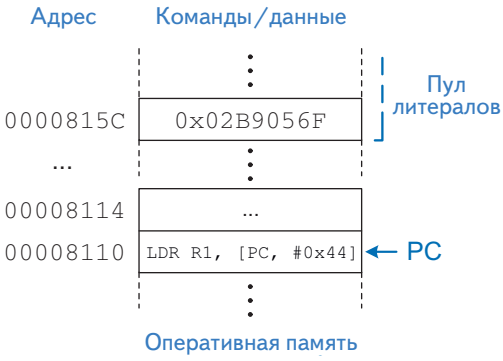


Рис. 1.32. Пример пула литералов

1.6.2. NOP

Мнемоника NOP означает «no operation» (нет операции). Это *псевдокоманда*, которая ничего не делает. Ассемблер заменяет ее командой `MOV R0, R0 (0xE1A00000)`. Команды NOP полезны, в частности, для реализации задержки или выравнивания команд на некоторую границу в памяти.

1.6.3. Исключения

Исключение можно уподобить неожиданному вызову функции с переходом на новый адрес. Причиной исключения может быть как оборудование, так и программа. Например, процессор может получить уведомление о том, что пользователь нажал клавишу на клавиатуре. В этом случае процессор может приостановить выполнение программы, определить, какая клавиша нажата, сохранить информацию об этом, после чего возобновить выполнение прерванной программы. Такие аппаратные исключения, вызванные устройствами ввода-вывода, например клавиатурой, часто называют *прерываниями* (interrupt). С другой стороны, исключение может быть вызвано ошибкой в программе, например из-за использования неопределенной команды. В этом случае программа передает управление операционной системе, которая может либо эмулировать нереализованную команду, либо завершить выполнение программы-нарушителя. Программные исключения иногда называют *ловушками* (trap). Особенно важной формой ловушки является *системный вызов*, в результате которого программа вызывает функцию ОС, работающую с повышенными привилегиями. Из других причин исключений отметим попытку чтения из несуществующей памяти.

Псевдокоманды не являются частью набора команд, а используются как сокращения команд или последовательностей команд, часто употребляемых программистами или компилятором. Ассемблер транслирует псевдокоманду в одну или несколько настоящих команд.

Как и любой вызов функции, обработчик исключения должен сохранить адрес возврата, перейти на другой адрес, выполнить свою работу, почистить после себя и вернуть управление в ту точку, откуда был вызван. Для обработки исключений используются таблица *векторов*, определяющая, на какой *обработчик исключения* перейти, и *банк регистров*, в котором хранятся дополнительные копии основных регистров, чтобы не повредить регистры активной программы. На время обработки исключения изменяется *уровень привилегий* программы, что дает обработчику возможность обращаться к защищенным областям памяти.

Режимы выполнения и уровни привилегий

Процессор ARM может функционировать в одном из нескольких режимов выполнения с разными уровнями привилегий. Благодаря различ-

ным режимам исключение, возникшее в обработчике исключения, не приводит к повреждению состояния; например, прерывание может произойти, когда процессор исполняет код операционной системы в режиме супервизора, а впоследствии могло бы возникнуть исключение Abort, если бы обработчик прерывания попытался обратиться к недействительному адресу памяти. В конечном итоге обработчики вернут управление, и возобновится выполнение супервизора. Режим задается в младших битах регистра текущего состояния программы (CPSR), показанного на **Рис. 1.6**. В **Табл. 1.13** перечислены все режимы управления и соответствующие им коды. В режиме пользователя процессор работает с уровнем привилегий PL0, запрещающим доступ к защищенным областям памяти, в т. ч. к коду операционной системы. В других режимах устанавливается уровень привилегий PL1, дающий доступ ко всем системным ресурсам. Уровни привилегий важны, потому что не дают некорректному или вредоносному пользовательскому коду повредить другие программы или привести к аварийному останову либо заражению системы.

Таблица 1.13. Режимы выполнения процессора ARM

Режим	$CPSR_{4:0}$
Пользователя	10000
Супервизора	10011
Аварийный	10111
Неопределенный	11011
Прерывания (IRQ)	10010
Быстрого прерывания (FIQ)	10001

Таблица векторов исключений

Когда происходит исключение, процессор переходит по адресу, указанному в *таблице векторов исключений* и зависящему от причины исключения. В **Табл. 1.14** описана таблица векторов, которая обычно находится по адресу 0x00000000 в памяти. Например, при возникновении прерывания процессор переходит по адресу 0x00000018, а при включении питания – по адресу 0x00000000. В каждом элементе вектора исключений обычно находится команда перехода на обработчик исключения, который обрабатывает исключение и либо завершает работу системы, либо возвращает управление пользовательскому коду.

ARM поддерживает также режим High Vectors, в котором таблица векторов исключений размещена по адресу 0xFFFF0000. Например, на этапе начальной загрузки системы может использоваться таблица векторов в постоянной памяти по адресу 0x00000000. А после того как система загрузится, она может записать обновленную таблицу векторов в ОЗУ по адресу 0xFFFF0000 и перейти в режим High Vectors.

Таблица 1.14. Таблица векторов исключений

Исключение	Адрес	Режим
Сброс	0x00	Супервизора
Неопределенная команда	0x04	Неопределенный
Вызов супервизора	0x08	Супервизора
Ошибка предвыборки (ошибка выборки команды)	0x0C	Аварийный
Ошибка данных (ошибка при загрузке или сохранении данных)	0x10	Аварийный
Зарезервирован	0x14	–
Прерывание	0x18	IRQ
Быстрое прерывание	0x1C	FIQ

Банк регистров

Прежде чем изменять счетчик команд в ответ на возникновение исключения, процессор должен сохранить адрес возврата в регистре LR, чтобы обработчик исключения знал, куда возвращаться. Однако ни в коем случае нельзя изменить значение, которое уже находится в LR, потому что оно еще понадобится программе. Поэтому процессор поддерживает банк альтернативных регистров, которые можно использовать в качестве LR в различных режимах выполнения. Точно так же обработчик исключения не должен изменять биты регистра состояния. Поэтому существует *банк сохраненных регистров состояния программы* (saved program status registers, SPSR), в котором хранятся копии CPSR на время обработки исключений.

Если исключение происходит в момент, когда программа манипулирует своим кадром стека, то кадр может оказаться в нестабильном состоянии (например, данные уже помещены в стек, но указатель стека еще не направлен на его вершину). Поэтому в каждом режиме выполнения есть свой собственный стек и копия регистра SP в банке, указывающая на его вершину. На этапе запуска система должна зарезервировать память для стека в каждом режиме выполнения и инициализировать указатели этих стеков.

Первым делом обработчик исключения должен поместить в стек все регистры, которые могут быть изменены. Это занимает некоторое время. В ARM предусмотрен режим *быстрого прерывания* (FIQ), в котором копии регистров R8–R12 также хранятся в банке. Поэтому обработчик исключения может начать работу немедленно, не сохраняя этих регистров.

Обработка исключения

Поняв, что такое режимы выполнения, векторы исключений и банки регистров, мы можем описать, что происходит во время обработки исключения. Обнаружив исключение, процессор выполняет следующие действия:

1) сохраняет CPSR в банке SPSR;
2) устанавливает режим выполнения и уровень привилегий в соответствии с типом исключения;
3) устанавливает в регистре CPSR биты *маски прерываний*, чтобы обработчик исключения не прерывался;
4) сохраняет адрес возврата в экземпляре LR в банке;
5) находит в таблице векторов исключений элемент, соответствующий типу возникшего исключения.

Затем процессор выполняет команду, хранящуюся в таблице векторов исключений, обычно это команда перехода на обработчик исключения. Обработчик, как правило, помещает в стек прочие регистры, обрабатывает исключение и восстанавливает регистры из стека. Обработчик возвращает управление с помощью команды `MOVS PC, LR` – специальной разновидности команды `MOV`, которая производит очистку:

1) восстанавливает регистр состояния, копируя значение из банка SPSR в CPSR;
2) копирует из банка регистр LR (для некоторых исключений он корректируется) в счетчик команд PC, чтобы вернуться в то место программы, где произошло исключение;
3) восстанавливает режимы выполнения и уровень привилегий.

Команды, относящиеся к обработке исключений

Пользовательские программы работают с низким уровнем привилегий, а операционная система – с высоким. Для контролируемого переключения уровня программа помещает аргументы в регистры и выполняет команду *вызова супервизора* (`SVC`), которая генерирует исключение и поднимает уровень привилегий. ОС анализирует аргументы, выполняет запрошенную функцию и возвращает управление программе.

ОС и другой код, работающий с уровнем привилегий PL1, может получить доступ к банку регистров в различных режимах выполнения с помощью команд `MRS` (поместить в регистр из специального регистра) и `MSR` (поместить в специальный регистр из регистра). Так, на этапе начальной загрузки ОС использует эти команды, чтобы инициализировать стеки для обработчиков исключений.

Запуск системы

На этапе запуска процессор переходит по адресу, хранящемуся в векторе сброса, и начинает выполнять код *начального загрузчика* в режиме супервизора. Типичный начальный загрузчик конфигурирует подсистему памяти, инициализирует указатель стека и читает код ОС с диска; затем он запускает гораздо более длительный процесс загрузки ОС. В конечном итоге ОС загружает некую программу, меняет режим на непривилегированный и переходит в начало этой программы.

1.7. Эволюция архитектуры ARM

Процессор ARM1 был разработан британской компанией Acorn Computer для компьютеров BBC Micro в 1985 году как модернизация микропроцессора 6502, которые в то время широко использовались во многих персональных компьютерах. Не прошло и года, как за ним последовал процессор ARM2, который промышленно изготавливался для компьютера Acorn Archimedes. ARM – это акроним *Acorn RISC Machine*. В изделии была реализована версия 2 набора команд ARM (ARMv2). Адресная шина была 26-битовой, а старшие 6 бит 32-битового счетчика команд использовались для хранения битов состояния. В архитектуре уже присутствовали почти все команды, описанные в этой главе, в т. ч. команды обработки данных, большинство команд загрузки и сохранения, перехода и умножения.

Вскоре адресная шина была расширена до полных 32 бит, а биты состояния переместились в специальный регистр текущего состояния программы (CPSR). В версии ARMv4, появившейся в 1993 году, были добавлены команды загрузки и сохранения полуслова, а также команды загрузки полуслов со знаком и без знака и байтов. С этим дополнением сформировалась базовая часть современного набора команд ARM, который мы рассматривали в этой главе.

В версии ARMv7 регистр CPSR называется регистром состояния прикладной программы (Application Program Status Register, APSR).

Набор команд ARM подвергался многочисленным усовершенствованиям, которые описываются ниже. Оказавшийся чрезвычайно успешным процессор ARM7TDMI, выпущенный в 1995 году, привнес 16-битовый набор команд Thumb в виде версии ARMv4T с целью повысить плотность кода. В версии ARMv5TE были добавлены команды для цифровой обработки сигналов (digital signal processing, DSP) и необязательные команды для чисел с плавающей точкой. В версии ARMv6 добавились мультимедийные команды и был расширен набор команд Thumb. В версии ARMv7 были усовершенствованы команды для работы с плавающей точкой и мультимедиа, которые получили обще название Advanced SIMD. Версия ARMv8 ознаменовалась совершенно новой

64-битовой архитектурой. По мере эволюции архитектуры добавлялись и другие команды для программирования системы.

1.7.1. Набор команд Thumb

Команды Thumb 16-битовые, что позволяет увеличить плотность кода; они повторяют обычные команды ARM, но с рядом ограничений:

- могут обращаться только к восьми младшим регистрам;
- один и тот же регистр играет роль источника и приемника;
- поддерживаются более короткие непосредственные операнды;
- отсутствует условное выполнение;
- всегда устанавливаются флаги состояния.

Почти у всех команд ARM есть эквивалентные команды Thumb. Поскольку команды Thumb скованы ограничениями, для написания эквивалентной программы их понадобится больше. Однако длина этих команд в два раза меньше, поэтому общий размер написанного с их помощью кода составляет примерно 65% от эквивалентного кода с использованием полной системы команд ARM. Набор команд Thumb полезен не только для уменьшения размера кода и стоимости требуемой для его хранения памяти. Он еще позволяет использовать дешевую 16-битовую шину для памяти команд и сокращает энергопотребление при выборке команд из памяти.

Иррегулярность кодирования команд из набора Thumb и переменная длина команд (1 или 2 полуслова) характерны для 16-битовых архитектур процессора, поскольку требуется упаковать много информации в короткую команду. Такая иррегулярность усложняет декодирование команд.

В процессорах ARM имеется регистр состояния набора команд, ISETSTATE, в котором есть бит T, определяющий, в каком режиме работает процессор: нормальном (T = 0) или Thumb (T = 1). От этого режима зависит способ выборки и интерпретации команд. Команды перехода `BX` и `BLX` переключают бит T – осуществляют соответственно вход в режим Thumb и выход из него.

Кодирование команд в режиме Thumb сложнее и не так единообразно, как в режиме ARM, поскольку требуется упаковать как можно больше полезной информации в 16-битовое полуслово. На **Рис. 1.33** показано, как кодируются некоторые часто встречающиеся команды Thumb. В старших битах задается тип команды. В командах обработки данных обычно указываются два регистра, один из которых играет роль источника и приемника. Любая команда всегда устанавливает флаги состояния. В командах сложения, вычитания и сдвига можно задавать короткий непосредственный операнд. В командах условного перехода задается 4-битовый код условия и короткое смещение, а в командах безусловного перехода допустимо более длинное смещение. Отметим, что команда `BX` принимает 4-битовый идентификатор регистра, поэтому может получить доступ к регистру связи LR. Существуют специальные разновидности ко-

манд LDR, STR, ADD и SUB, работающие относительно указателя стека SP (чтобы можно было получить доступ к кадру стека при обработке вызова функции). Еще одна разновидность команды LDR загружает данные относительно счетчика команд PC (для доступа к пулу литералов). Варианты команд ADD и MOV позволяют обращаться ко всем 16 регистрам. В команде BL необходимо всегда задавать два полуслова, определяющих 22-битовый конечный адрес.

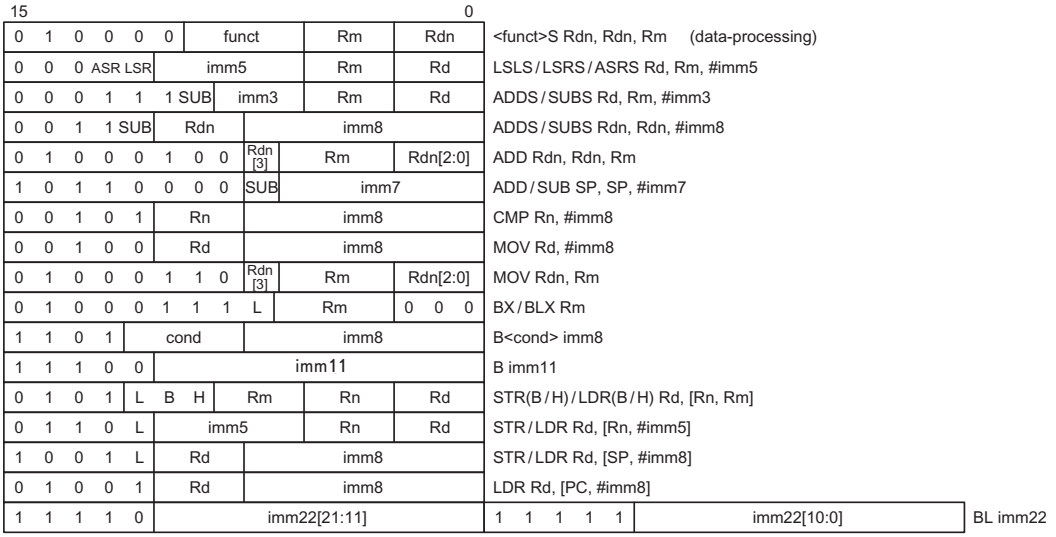


Рис. 1.33. Примеры кодирования команд из набора Thumb

Впоследствии набор команд Thumb был усовершенствован. В набор Thumb-2 добавили ряд 32-битовых команд для повышения производительности типичных операций и для того, чтобы в режиме Thumb можно было написать любую программу. Команда принадлежит набору Thumb-2, если 5 старших бит принимают одно из значений: 11101, 11110, 11111. В этом случае процессор выбирает из памяти второе полуслово, содержащее конец команды. Процессоры серии Cortex-M работают исключительно в режиме Thumb.

1.7.2. Команды для цифровой обработки сигналов

Цифровые сигнальные процессоры (digital signal processor, DSP) предназначены для эффективной реализации алгоритмов обработки сигналов, в т. ч. быстрого преобразования Фурье (БПФ) и фильтров с конечной или бесконечной импульсной характеристикой (КИХ- и БИХ-фильтров). К типичным приложениям относятся кодирование и декодирование аудио и видео, управление приводом и распознавание речи. ARM предоставля-

ет ряд команд DSP для этих целей, в т. ч. умножение, сложение и умножение с накоплением (multiply-accumulate, MAC) – умножение с последующим прибавлением к текущей сумме: sum = sum + src1 × src2. Именно команда MAC отличает наборы команд DSP от обычных наборов команд. Она очень часто используется в алгоритмах DSP и в два раза повышает производительность, по сравнению с раздельными командами умножения и сложения. Однако в MAC необходимо указывать дополнительный регистр для хранения накопительной суммы.

Быстрое преобразование Фурье (БПФ), самый распространенный алгоритм цифровой обработки сигналов, с одной стороны, сложен, а с другой – должен работать быстро. В компьютерных архитектурах предусмотрены специальные команды DSP для эффективного БПФ, особенно в случае 16-битовых дробных данных.

Базовые команды умножения, перечисленные в **приложении А**, входят в состав набора команд ARMv4. В версии ARMv5TE для поддержки алгоритмов DSP были добавлены арифметические команды с насыщением, а также команды умножения упакованных и дробных чисел.

Команды DSP часто применяются к коротким (16-битовым) данным, представляющим отсчеты, полученные от датчика с помощью аналого-цифрового преобразователя (АЦП). Однако промежуточные результаты хранятся с большей точностью (32 или 64 бита) или обрезаются для предотвращения переполнения. В *арифметике с насыщением* результат, больший наибольшего представимого положительного числа, заменяется этим наибольшим положительным числом, а результат, меньший наименьшего представимого отрицательного числа, заменяется этим наименьшим отрицательным числом. Например, в 32-битовой арифметике с насыщением результаты, большие $2^{31} - 1$, заменяются на $2^{31} - 1$, а меньшие -2^{31} заменяются на -2^{31}. Общеупотребительные в DSP типы данных приведены в **Табл. 1.15**. Числа в дополнительном коде опознаются по наличию одного знакового бита. 16-, 32- и 64-битовые типы называются соответственно типами с *половинной*, *одинарной* и *двойной* точностью, их не следует путать с числами с плавающей точкой одинарной и двойной точности. Ради эффективности два числа с половинной точностью упаковываются в одно 32-битовое слово.

Таблица 1.15. Типы данных для DSP

Тип	Знаковый бит	Бит целой части	Бит дробной части
short	1	15	0
unsigned short	0	16	0
long	1	31	0
unsigned long	0	32	0
long long	1	63	0
unsigned long long	0	64	0
Q15	1	0	15
Q31	1	0	31

Арифметика с насыщением — важный способ постепенного уменьшения точности алгоритмов DSP. Для большинства входных данных достаточно арифметики с одинарной точностью, но бывают патологические случаи, когда диапазона одинарной точности не хватает. Из-за переполнения внезапно выдается совершенно неверный ответ, который может выглядеть как щелчок в звуковом потоке или пиксель странного цвета в видеопотоке. Переход к арифметике с двойной точностью предотвращает переполнение, но снижает производительность и, как правило, увеличивает энергопотребление. Арифметика с насыщением заменяет переполнение обрезанием результатов, обычно при этом результаты получаются лучше, и потеря точности оказывается меньше.

Тип *integer* бывает знаковым и беззнаковым, знаковым является старший бит. *Дробные* типы (Q15 и Q31) служат для представления дробных чисел со знаком; например, тип Q31 охватывает диапазон $[-1, 1 - 2^{-31}]$ с шагом между соседними числами 2^{-31}. Эти типы не определены в стандарте языка C, но поддерживаются некоторыми библиотеками. Тип Q31 можно преобразовать в Q15 путем усечения или округления. В случае усечения остается просто старшее полуслово. В случае округления к значению типа Q31 прибавляется 0x00008000, а затем результат усекается. Если вычисление состоит из большого числа шагов, то округление полезно, т. к. предотвращает накопление малых ошибок усечения, которые в сумме выливаются в большую погрешность.

В архитектуре ARM в регистры состояния добавлен флаг *Q*, указывающий, что в командах DSP имело место переполнение или насыщение. В приложениях, где точность критична, программа может сбросить флаг *Q* до начала вычисления, произвести вычисление в режиме одинарной точности, а затем проверить флаг *Q*. Если он поднят, значит, произошло переполнение, и при необходимости вычисление можно повторить с двойной точностью.

Сложение и вычитание выполняются одинаково вне зависимости от формата. Но умножение зависит от типа. Например, в случае 16-битовых чисел 0xFFFF интерпретируется как 65 535 для типа unsigned short, как −1 для типа short и как -2^{-15} для типа Q15. Следовательно, произведение 0xFFFF × 0xFFFF зависит от представления (соответственно 4 294 836 225, 1 и 2^{-30}). Поэтому для умножения со знаком и без знака нужны разные команды.

Число *A* типа Q15 можно рассматривать как $a \times 2^{-15}$, где a — его интерпретация в диапазоне $[-2^{15}, 2^{15} - 1]$ как 16-битового числа со знаком. Поэтому произведение двух чисел типа Q15 равно:

$$A \times B = a \times b \times 2^{-30} = 2 \times a \times b \times 2^{-31}.$$

Это означает, что для умножения двух чисел типа Q15 и получения результата типа Q31 нужно произвести обыкновенное умножение со знаком, а затем удвоить произведение. Затем при необходимости это произведение можно усечь или округлить, чтобы привести его снова к формату Q15.

Богатый ассортимент команд умножения и умножения с накоплением приведен в **Табл. 1.16**. Команды умножения с накоплением требуют задания до четырех регистров: *RdHi*, *RdLo*, *Rn* и *Rm*. В случае операций с двойной точностью в *RdHi* и *RdLo* размещаются старшие и младшие

32 бита соответственно. Например, команда `UMLAL RdLo, RdHi, Rn, Rm` вычисляет $\{RdHi, RdLo\} = \{RdHi, RdLo\} + Rn \times Rm$. Команды умножения с половинной точностью имеют по нескольку вариантов, перечисленных в фигурных скобках, в зависимости от того, берутся операнды из старшей или младшей половины слова, а в *дуальных* формах перемножаются старшие и младшие половины. Команды умножения с накоплением, применяемые к входным данным половинной точности и аккумулятору одинарной точности (`SMLA*`, `SMLAW*`, `SMUAD`, `SMUSD`, `SMLAD`, `SMLSD`), устанавливают флаг Q в случае переполнения аккумулятора. Команды умножения старшего слова (MSW) также имеют варианты с суффиксом `R`, которые производят округление, а не усечение.

Таблица 1.16. Команды умножения и умножения с накоплением

Команда	Функция	Описание
Обыкновенное 32-битовое умножение работает для чисел со знаком и без знака		
`MUL`	`32 = 32 × 32`	Умножение
`MLA`	`32 = 32 + 32 × 32`	Умножение с накоплением
`MLS`	`32 = 32 – 32 × 32`	Умножение с вычитанием
unsigned long long = unsigned long × unsigned long		
`UMULL`	`64 = 32 × 32`	Длинное умножение без знака
`UMLAL`	`64 = 64 + 32 × 32`	Длинное умножение с накоплением без знака
`UMAAL`	`64 = 32 + 32 × 32 + 32`	Длинное умножение с накоплением и сложением без знака
long long = long × long		
`SMULL`	`64 = 32 × 32`	Длинное умножение со знаком
`SMLAL`	`64 = 64 + 32 × 32`	Длинное умножение с накоплением со знаком
Арифметика упакованных чисел: short × short		
`SMUL{BB/BT/TB/TT}`	`64 = 32 × 32`	Умножение со знаком {младшее/старшее}
`SMLA{BB/BT/TB/TT}`	`64 = 64 + 32 × 32`	Умножение с накоплением со знаком {младшее/старшее}
`SMLAL{BB/BT/TB/TT}`		Длинное умножение с накоплением со знаком {младшее/старшее}

Табл. 6.16. (окончание)

Команда	Функция	Описание
	Дробное умножение (Q31 / Q15)	
SMULW{B/T}	32 = (32 × 16) >> 16	Умножение слова на полуслово со знаком {младшее/старшее}
SMLAW{B/T}	32 = 32 + (32 × 16) >> 16	Умножение со сложением слова на полуслово со знаком {младшее/старшее}
SMMUL{R}	32 = (32 × 32) >> 32	Умножение старших слов со знаком {с округлением}
SMMLA{R}	32 = 32 + (32 × 32) >> 32	Умножение с накоплением старших слов со знаком {с округлением}
SMMLS{R}	32 = 32 – (32 × 32) >> 32	Умножение с вычитанием старших слов со знаком {с округлением}
	long или long long = short × short + short × short	
SMUAD	32 = 16 × 16 + 16 × 16	Дуальное умножение со сложением со знаком
SMUSD	32 = 16 × 16 – 16 × 16	Дуальное умножение с вычитанием со знаком
SMLAD	32 = 32 + 16 × 16 + 16 × 16	Дуальное умножение с накоплением со знаком
SMLSD	32 = 32 + 16 × 16 – 16 × 16	Дуальное умножение с вычитанием со знаком
SMLALD	64 = 64 + 16 × 16 + 16 × 16	Длинное дуальное умножение с накоплением со знаком
SMLSLD	64 = 64 + 16 × 16 – 16 × 16	Длинное дуальное умножение с вычитанием со знаком

Среди команд DSP есть также сложение с насыщением (QADD) и вычитание с насыщением (QSUB) 32-битовых слов, при которых вместо переполнения производится насыщение результата. Команды QDADD и QDSUB умножают второй операнд на 2 перед сложением с первым или вычитанием из первого (с насыщением); скоро мы покажем, насколько они полезны при умножении с накоплением дробных чисел. В случае насыщения эти команды поднимают флаг *Q*.

Наконец, к командам DSP относятся также LDRD и STRD, которые загружают и сохраняют пару четный/нечетный регистр в 64-битовое двойное слово. Эти команды повышают эффективность обмена значений с двойной точностью между регистрами и памятью.

В **Табл. 1.17** показано, как использовать команды DSP для умножения или умножения с накоплением данных разных типов. В примерах предполагается, что полуслово находится в младшей половине регистра, а старшая половина равна 0. Если данные находятся в старшей половине регистра, то следует использовать вариант команды `SMUL` с суффиксом `T`. Результат сохраняется в регистре R2 или в паре регистров {R3, R2} в случае операций с двойной точностью. Операции с дробными типами (Q15/Q31) удваивают результат, применяя сложение с насыщением, чтобы предотвратить переполнение при умножении -1×-1.

Таблица 1.17. Примеры команд умножения и умножения с накоплением для разных типов данных

Первый операнд (R0)	Второй операнд (R1)	Произведение (R3/R2)	Умножение	Умножение с накоплением

short	short	short	`SMULBB R2, R0, R1`
			`LDR R3, =0x0000FFFF`
			`AND R2, R3, R2`
				`SMLABB R2, R0, R1`
				`LDR R3, =0x0000FFFF`
				`AND R2, R3, R2`

short	short	long	`SMULBB R2, R0, R1`	`SMLABB R2, R0, R1, R2`

short	short	long long	`MOV R2, #0`
			`MOV R3, #0`
			`SMLALBB R2, R3, R0, R1`
				`SMLALBB R2, R3, R0, R1`

long	short	long	`SMULWB R2, R0, R1`	`SMLAWB R2, R0, R1, R2`
long	long	long	`MUL R2, R0, R1`	`MLA R2, R0, R1, R2`
long	long	long long	`SMULL R2, R3, R0, R1`	`SMLAL R2, R3, R0, R1`

unsigned short	unsigned short	unsigned short	`MUL R2, R0, R1`
			`LDR R3, =0x0000FFFF`
			`AND R2, R3, R2`
				`MLA R2, R0, R1, R2`
				`LDR R3, =0x0000FFFF`
				`AND R2, R3, R2`

unsigned short	unsigned short	unsigned long	`MUL R2, R0, R1`	`MLA R2, R0, R1, R2`
unsigned long	unsigned short	unsigned long	`MUL R2, R0, R1`	`MLA R2, R0, R1, R2`
unsigned long	unsigned long	unsigned long	`MUL R2, R0, R1`	`MLA R2, R0, R1, R2`
unsigned long	unsigned long	unsigned long long	`UMULL R2, R3, R0, R1`	`UMLAL R2, R3, R0, R1`

Q15	Q15	Q15	`SMULBB R2, R0, R1`
			`QADD R2, R2, R2`
			`LSR R2, R2, #16`
				`SMLABB R2, R0, R1, R2`
				`SSAT R2, 16, R2`

Q15	Q15	Q31	`SMULBB R2, R0, R1` `QADD R2, R2, R2`	`SMULBB R3, R0, R1` `QDADD R2, R2, R3`
Q31	Q15	Q31	`SMULWB R2, R0, R1` `QADD R2, R2, R2`	`SMULWB R3, R0, R1` `QDADD R2, R2, R3`
Q31	Q31	Q31	`SMMUL R2, R0, R1` `QADD R2, R2, R2`	`SMMUL R3, R0, R1` `QDADD R2, R2, R3`

1.7.3. Команды арифметики с плавающей точкой

Операции с плавающей точкой обладают большей гибкостью, чем операции с фиксированной точкой, применяемые для цифровой обработки сигналов, поэтому программирование упрощается. Операции с плавающей точкой находят широкое применение в машинной графике, научных приложениях и алгоритмах управления. Арифметику с плавающей точкой можно реализовать с помощью последовательности обычных команд обработки данных, но специализированное оборудование и команды быстрее и потребляют меньше энергии.

В набор команд ARMv5 входят факультативные команды операций с плавающей точкой. Они обращаются как минимум к 16-ти 64-битовым регистрам двойной точности, независимым от обыкновенных регистров. Эти регистры можно также рассматривать как пары 32-битовых регистров одинарной точности. Регистры двойной точности называются D0–D15, а они же, трактуемые как регистры одинарной точности, – S0–S31. Например, команды `VADD.F32 S2, S0, S1` и `VADD.F64 D2, D0, D1` выполняют сложение с плавающей точкой одинарной и двойной точности соответственно. К командам операций с плавающей точкой, перечисленным в **Табл. 1.18**, добавляется суффикс `.F32` или `.F64`, обозначающий соответственно одинарную или двойную точность.

Таблица 1.18. Команды операций с плавающей точкой в ARM

Команда	Назначение
`VABS Rd, Rm`	Rd = |Rm|
`VADD Rd, Rn, Rm`	Rd = Rn + Rm
`VCMP Rd, Rm`	Сравнить и установить флаги состояния операций с плавающей точкой
`VCVT Rd, Rm`	Сравнить int и float
`VDIV Rd, Rn, Rm`	Rd = Rn / Rm
`VMLA Rd, Rn, Rm`	Rd = Rd + Rn * Rm
`VMLS Rd, Rn, Rm`	Rd = Rd – Rn * Rm
`VMOV Rd, Rm или #const`	Rd = Rm или константа
`VMUL Rd, Rn, Rm`	Rd = Rn * Rm
`VNEG Rd, Rm`	Rd = –Rm
`VNMLA Rd, Rn, Rm`	Rd = –(Rd + Rn * Rm)
`VNMLS Rd, Rn, Rm`	Rd = –(Rd – Rn * Rm)
`VNMUL Rd, Rn, Rm`	Rd = –Rn * Rm
`VSQRT Rd, Rm`	Rd = sqrt(Rm)
`VSUB Rd, Rn, Rm`	Rd = Rn – Rm

Команды MRC и MCR служат для копирования данных между обыкновенными регистрами и регистрами сопроцессора для операций с плавающей точкой.

В ARM определен регистр управления и состояния операций с плавающей точкой (Floating-Point Status and Control Register, FPSCR). Как и обыкновенный регистр состояния, он содержит флаги *N*, *Z*, *C* и *V*, но только для операций с плавающей точкой. В нем же задается режим округления, исключения и такие специальные условия, как переполнение, потеря значимости и деление на нуль. Команды VMRS и VMSR служат для копирования данных между обычным регистром и FPSCR.

1.7.4. Команды энергосбережения и безопасности

Устройства, работающие от аккумулятора, снижают энергопотребление, проводя большую часть времени в спящем режиме. В версии ARMv6K добавлены команды для поддержки энергосбережения. Команда *ожидать прерывания* (wait for interrupt – WFI) переводит процессор в состояние пониженного энергопотребления до возникновения прерывания. Система может генерировать прерывания в ответ на события, инициированные пользователем (например, касание экрана), или по периодически срабатывающему таймеру. Команда *ожидать события* (wait for event – WFE) аналогична, но применяется в многопроцессорных системах (см. **раздел 2.7.8**), чтобы перевести процессор в спящий режим до уведомления от другого процессора. Он просыпается при возникновении прерывания или когда другой процессор посылает событие командой SEV.

В версии ARMv7 механизм обработки исключений дополнен поддержкой виртуализации и безопасности. Под *виртуализацией* понимается одновременная работа на одном и том же процессоре нескольких операционных систем, не подозревающих о существовании друг друга. Переключение между операционными системами осуществляет гипервизор, работающий с уровнем привилегий PL2. Гипервизор вызывается с помощью исключения ловушки гипервизора. Благодаря расширениям *безопасности* процессор определяет безопасное состояние, в котором средства ввода ограничены и для доступа к защищенным областям нужны специальные права. Даже если противнику удастся скомпрометировать операционную систему, безопасное ядро устоит против попыток манипуляции. Например, безопасное ядро можно использовать, чтобы заблокировать украденный телефон или активировать управление цифровыми правами, запрещающее пользователю дублировать материалы, защищенные авторским правом.

1.7.5. Команды SIMD

Акроним SIMD (произносится «сим-ди») означает «single instruction multiple data» (один поток команд, несколько потоков данных), когда одна команда воздействует на несколько элементов данных параллельно. Типичное применение SIMD – выполнение большого количества коротких арифметических операций, особенно для обработки графики. Иногда можно услышать термин «арифметика упакованных чисел».

При обработке графики короткие элементы данных возникают часто. Например, один пиксель в цифровой фотографии может потребовать 8 бит для хранения каждого из трех цветовых каналов: красного, зеленого и синего. Если использовать целое 32-битовое слово для хранения каждого канала, то 24 старших бита пропадают без пользы. Более того, если каналы 16 соседних пикселей упаковать в 128-битовое четверное слово, то обработку можно ускорить в 16 раз. Аналогично координаты в трехмерном пространстве обычно представляют 32-битовым числом с плавающей точкой (одинарной точности). В 128-битовое четверное слово можно упаковать четыре такие координаты.

В большинстве современных архитектур имеются арифметические SIMD-операции с широкими SIMD-регистрами, в которые упаковываются несколько более узких операндов. Например, команды из набора ARMv7 Advanced SIMD используют общие регистры с устройством плавающей арифметики. Кроме того, эти регистры можно объединить в пары, образующие восемь 128-битовых четверных слов Q0–Q7. В регистры упаковывается несколько 8-, 16-, 32- или 64-битовых чисел, целых или с плавающей точкой. Мнемоническое обозначение команды дополняется суффиксом `.I8`, `.I16`, `.I32`, `.I64`, `.F32` или `.F64`, показывающим, как следует интерпретировать регистры.

На **Рис. 1.34** показана команда векторного сложения `VADD.I8 D2, D1, D0` в применении к восьми парам 8-битовых целых чисел, упакованных в 64-битовые двойные слова. Аналогично команда `VADD.I32 Q2, Q1, Q0` складывает четыре пары 32-битовых целых чисел, упакованных в 128-битовые четверные слова, а команда `VADD.F32, D2, D1, D0` складывает две пары 32-битовых чисел с плавающей точкой одинарной точности, упакованные в 64-битовые двойные слова. Для выполнения арифметических операций над упакованными числами необходимо модифицировать АЛУ, чтобы исключить переносы между составляющими элементами данных. Например, перенос при сложении $a_0 + b_0$ не должен влиять на результат сложения $a_1 + b_1$.

Названия команд из набора Advanced SIMD начинаются буквой `V`. Их можно отнести к следующим категориям:

- базовые арифметические операции, определенные также для чисел с плавающей точкой;

- команды загрузки и сохранения нескольких элементов, включая чередование и устранение чередования;
- поразрядные логические операции;
- операции сравнения;
- разновидности операций сдвига, сложения и вычитания с насыщением и без;
- разновидности операций умножения и умножения с накоплением;
- прочие команды.

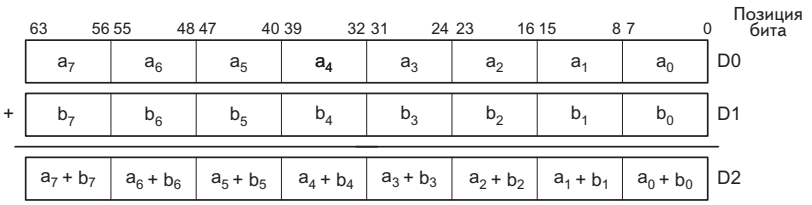


Рис. 1.34. Арифметика упакованных чисел: восемь одновременных операций сложения 8-битовых целых чисел

В ARMv6 определен также ограниченный набор SIMD-команд, оперирующих обыкновенными 32-разрядными регистрами. Он включает 8- и 16-битовые команды сложения и вычитания, а также команды эффективной упаковки и распаковки байтов и полуслов в слова. Эти команды полезны для операций с 16-битовыми данными в программах цифровой обработки сигналов.

1.7.6. 64-битовая архитектура

В 32-битовых архитектурах программа может напрямую адресовать не более $2^{32} = 4$ ГБ памяти. Появление больших серверов повлекло за собой переход к 64-битовым архитектурам, в которых объем адресуемой памяти гораздо больше. За ними последовали персональные компьютеры, а затем и мобильные устройства. Иногда 64-битовая архитектура оказывается еще и быстрее, потому что позволяет перемещать больше информации с помощью одной команды.

Во многих архитектурах обошлись просто расширением регистров общего назначения с 32 до 64 битов, но в версии ARMv8 появился также новый набор команд, устраняющий прошлые огрехи. В классическом наборе команд не хватает регистров общего назначения для сложных программ, что вынуждает идти на дорогостоящие перемещения данных между регистрами и памятью. Хранение счетчика команд (PC) в регистре R15, а указателя стека (SP) в R13 также усложняет реализацию процессора. А еще программам часто нужен регистр, содержащий значение 0.

В версии ARMv8 команды по-прежнему имеют длину 32 бита, а набор команд очень похож на тот, что был в версии ARMv7, но некоторые проблемы разрешены. Файл регистров расширен и теперь включает 31 64-битовый регистр (X0–X30), а PC и SP перестали быть регистрами общего назначения. X30 играет роль регистра связи. Отметим, что регистра X31 не существует, он называется нулевым регистром (ZR) и постоянно содержит 0. Команды обработки данных могут работать с 32- или 64-битовыми операндами, но в командах сохранения и загрузки всегда используются 64-битовые адреса. Чтобы освободить место для дополнительных битов, необходимых для задания регистра-источника и регистра-приемника, из большинства команд убрано поле условия. Однако команды перехода по-прежнему могут быть условными. Кроме того, в ARMv8 упрощена обработка исключений, удвоено количество дополнительных SIMD-регистров и добавлены команды для реализации криптографических алгоритмов AES и SHA. Система кодирования команд довольно сложна и не укладывается в полдесятка категорий.

После сброса процессоры версии ARMv8 загружаются в 64-битовом режиме. Чтобы перейти в 32-битовый режим, процессор должен установить бит в системном регистре и вызвать исключение. После возврата из обработчика исключения восстанавливается 64-битовый режим.

1.8. Живой пример: архитектура x86

Практически во всех современных персональных компьютерах используются процессоры с архитектурой x86. Архитектура x86, также называемая IA-32, – это 32-битовая архитектура, изначально разработанная компанией Intel. Компания AMD также продает x86-совместимые микропроцессоры.

Архитектура x86 имеет долгую и запутанную историю, которая берет начало в 1978 году, когда Intel объявила о разработке 16-разрядного микропроцессора 8086. Компания IBM выбрала 8086 и его собрата 8088 для своих первых персональных компьютеров (ПК). В 1985 году Intel представила 32-разрядный микропроцессор 80386, который был обратно совместим с 8086 и мог исполнять программы, разработанные для более ранних ПК. Процессорные архитектуры, совместимые с 80386, называют x86-совместимыми архитектурами. Pentium, Core и Athlon – наиболее известные x86-совместимые процессоры.

Различные группы разработчиков в Intel и AMD на протяжении многих лет добавляли все новые команды и возможности в архаичную архитектуру. В результате она выглядит гораздо менее элегантно, чем ARM. Тем не менее совместимость программного обеспечения гораздо важнее технической элегантности, так что x86 был стандартом де-факто для ПК на протяжении более чем двух десятилетий. Каждый год продается свы-

ше 100 миллионов x86-совместимых микропроцессоров. Это огромный рынок, оправдывающий ежегодные затраты на улучшение этих процессоров, превышающие пять миллиардов долларов.

Архитектура x86 является примером CISC-архитектуры (Complex Instruction Set Computer – компьютер со сложным набором команд). В отличие от RISC-архитектур, таких как ARM, каждая CISC-команда способна произвести больше полезной работы. Из-за этого программы для CISC-архитектур обычно содержат меньше команд. Кодирование команд выбрано так, чтобы обеспечить компактность кода – это было важно в те времена, когда стоимость оперативной памяти была гораздо выше, чем сейчас. Команды имеют переменную длину и зачастую короче 32 бит. Недостаток же состоит в том, что сложные команды трудно декодировать, к тому же они, как правило, работают медленнее.

В этом разделе мы ознакомимся с архитектурой x86. Наша цель состоит не в том, чтобы сделать вас программистом на языке ассемблера x86, а, скорее, в том, чтобы проиллюстрировать некоторые сходства и различия между x86 и ARM. Нам кажется интересным взглянуть, как работает архитектура x86. Впрочем, материал этого раздела не обязателен для понимания оставшейся части книги. Основные отличия между x86 и ARM приведены в **Табл. 1.19**.

Таблица 1.19. Основные отличия между ARM и x86

Характеристика	ARM	x86
Количество регистров	15, общего назначения	8, некоторые ограничения по использованию
Количество операндов	3–4 (2–3 источника, 1 приемник)	2 (1 источник, 1 источник/приемник)
Расположение операндов	Регистры или непосредственные операнды	Регистры, непосредственные операнды или память
Размер операнда	32 бита	8, 16 или 32 бита
Флаги условий	Есть	Есть
Типы команд	Простые	Простые и сложные
Размер команд	Фиксированный, 4 байта	Переменный, 1–15 байтов

1.8.1. Регистры x86

У микропроцессора 8086 было восемь 16-битовых регистров. Для некоторых был возможен доступ отдельно к старшим и младшим восьми битам. В 32-битовой архитектуре 80386 регистры были просто расширены до 32 бит. Они называются EAX, ECX, EDX, EBX, ESP, EBP, ESI и EDI. Для обеспечения обратной совместимости была оставлена возможность получить отдельный доступ к их младшим 16 битам, а для некоторых регистров – и к младшим 8 битам, как показано на **Рис. 1.35**.

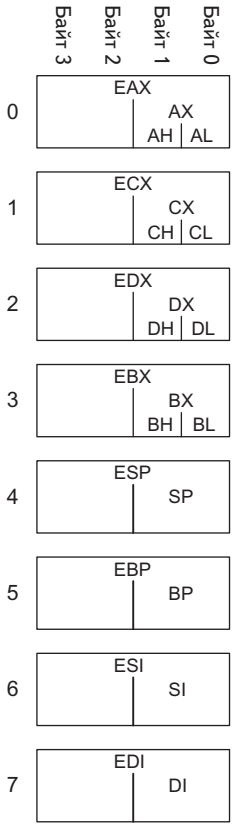


Рис. 1.35. Регистры x86

Эти восемь регистров можно, за некоторыми исключениями, считать регистрами общего назначения. Не все команды могут использовать все регистры. Бывает и так, что команда записывает результат в строго определенные регистры. Так же как регистр SP в ARM, регистр ESP, как правило, играет роль указателя стека.

Счетчик команд в архитектуре x86 называется EIP (extended instruction pointer). Аналогично счетчику команд в архитектуре ARM, он увеличивается при переходе от одной команды к другой, а также может быть изменен командами перехода и вызова функций.

1.8.2. Операнды x86

Команды ARM всегда применяются к регистрам или к непосредственным операндам. Для перемещения данных между памятью и регистрами необходимы явные команды загрузки и сохранения. Команды x86, напротив, могут работать не только с регистрами и непосредственными операндами, но и с памятью. Отчасти это компенсирует небольшой набор регистров.

Команды ARM обычно содержат три операнда: два источника и один приемник. Команды x86 содержат только два операнда. Первый является источником, а второй – одновременно источником и приемником. Следовательно, команда x86 всегда записывает результат на место одного из операндов. В **Табл. 1.20** перечислены поддерживаемые комбинации расположения операндов в командах x86. Как видно, возможны любые комбинации, исключая память-память.

Таблица 1.20. Расположение операндов

Источник/приемник	Источник	Пример	Операция
Регистр	Регистр	`add EAX, EBX`	`EAX <- EAX + EBX`
Регистр	Непосредственный операнд	`add EAX, 42`	`EAX <- EAX + 42`
Регистр	Память	`add EAX, [20]`	`EAX <- EAX + Mem[20]`
Память	Регистр	`add [20], EAX`	`Mem[20] <- Mem[20] + EAX`
Память	Непосредственный операнд	`add [20], 42`	`Mem[20] <- Mem[20] + 42`

Как и в ARM, адресное пространство в архитектуре x86 32-битовое с побайтовой адресацией. Однако x86 поддерживает больше различных режимов индексирования памяти. Расположение ячейки памяти задает-

ся при помощи комбинации *базового регистра*, *смещения* и масштабируемого *индексного регистра* (см. **Табл. 1.21**). Смещение может быть 8-, 16- или 32-битовым. Масштабируемый индексный регистр можно умножать на 1, 2, 4 или 8. Режим базовой адресации со смещением аналогичен режиму базовой адресации в ARM, используемому в командах загрузки и сохранения. В x86 масштабируемый индекс обеспечивает простой способ доступа к массивам и структурам с 2-, 4- или 8-байтовыми элементами без необходимости использовать последовательность команд для явного вычисления адреса.

Таблица 1.21. Режимы адресации памяти

Пример	Операция	Комментарий
`add EAX, [20]`	`EAX <- EAX + Mem[20]`	Смещение
`add EAX, [ESP]`	`EAX <- EAX + Mem[ESP]`	Базовая адресация
`add EAX, [EDX+40]`	`EAX <- EAX + Mem[EDX+40]`	Базовая адресация + смещение
`add EAX, [60+EDI*4]`	`EAX <- EAX + Mem[60+EDI*4]`	Смещение + масштабируемый индекс
`add EAX, [EDX+80+EDI*2]`	`EAX <- EAX + Mem[EDX+80+EDI*2]`	Базовая адресация + смещение + масштабируемый индекс

Если в ARM команды оперируют только 32-битовыми словами, то в x86 можно использовать 8-, 16- или 32-битовые данные. Это иллюстрируется в **Табл. 1.22**.

Таблица 1.22. Команды, применяемые к 8-, 16- или 32-битовым операндам

Пример	Операция	Размер операндов
`add AH, BL`	`AH <- AH + BL`	8 бит
`add AX, -1`	`AX <- AX + 0xFFFF`	16 бит
`add EAX, EDX`	`EAX <- EAX + EDX`	32 бита

1.8.3. Флаги состояния

Как и в большинстве CISC-архитектур, в x86 используются флаги условий (называемые также *флагами состояния*) для принятия решений о ветвлении и отслеживания переносов и арифметических переполнений. В архитектуре x86 флаги состояния хранятся в 32-битовом регистре EFLAGS. Назначение некоторых битов из регистра EFLAGS приведено в **Табл. 1.23**. Остальные биты используются операционной системой.

Характер использования флагов условий в ARM отличает ее от других RISC-архитектур.

Таблица 1.23. Некоторые биты регистра EFLAGS

Название	Назначение
`CF` (Carry Flag, флаг переноса)	При выполнении последней арифметической операции имел место перенос из старшего разряда. Указывает на то, что произошло переполнение при операциях над числами без знака. Также используется как флаг переноса между словами при выполнении операций над числами кратной точности
`ZF` (Zero Flag, флаг нуля)	Показывает, что результат последней операции равен нулю
`SF` (Sign Flag, флаг знака)	Показывает, что результат последней операции был отрицательным (старший бит результата равен 1)
`OF` (Overflow Flag, флаг переполнения)	Показывает, что произошло переполнение при операциях над числами со знаком в дополнительном коде

Архитектурное состояние процессора x86 включает в себя EFLAGS, а также восемь регистров и EIP.

1.8.4. Команды x86

В архитектуре x86 команд больше, чем в ARM. В **Табл. 1.24** показаны некоторые команды общего назначения. Система команд x86 также включает операции над числами с плавающей точкой и короткими векторами упакованных данных. Операнд-приемник (регистр или ячейка памяти) обозначается буквой `D`, а операнд-источник (регистр, непосредственный операнд или ячейка памяти) – буквой `S`.

Обратите внимание, что некоторые команды всегда применяются только к определенным регистрам. Например, при умножении двух 32-битовых чисел одним источником всегда является регистр EAX, 64-битовый результат записывается в EDX и EAX. В команде `LOOP` счетчик итераций цикла всегда находится в ECX, а в командах `PUSH`, `POP`, `CALL` и `RET` используется указатель стека ESP.

Команды условного перехода проверяют значения флагов и, если выполнено соответствующее условие, осуществляют переход. Эти команды имеют много разновидностей. Например, команда `JZ` производит переход в том случае, когда флаг нуля (*ZF*) равен 1, а команда `JNZ` – когда *ZF* равен 0. Как и в ARM, команды перехода обычно следуют за командами, которые устанавливают флаги, например `CMP` (сравнение). В **Табл. 1.25** перечислены некоторые команды условного перехода с указанием флагов, от которых они зависят.

1.8.5. Кодирование команд x86

Система кодирования команд x86 очень запутана и обременена наследием постепенных изменений, производившихся на протяжении десятилетий. В отличие от ARMv4, где команды всегда имеют длину 32 бита, в

x86 длина команды может составлять от 1 до 15 байтов, как показано на **Рис. 1.36**.[1]

Таблица 1.24. Некоторые команды x86

Команда	Назначение	Операция
ADD/SUB	Сложение/вычитание	D = D + S / D = D – S
ADDC	Сложение с переносом	D = D + S + CF
INC/DEC	Инкремент/декремент	D = D + 1 / D = D – 1
CMP	Сравнение	Установить флаги по результатам D – S
NEG	Инверсия	D = –D
AND/OR/XOR	Логическое И/ИЛИ/Исключающее ИЛИ	D = D операция S
NOT	Логическое НЕ	D = $\overline{D}$
IMUL/MUL	Знаковое/беззнаковое Умножение	EDX:EAX = EAX × D
IDIV/DIV	Знаковое/беззнаковое Деление	EDX:EAX/D EAX = частное; EDX = остаток
SAR/SHR	Арифметический/логический сдвиг вправо	D = D >>> S / D = D >> S
SAL/SHL	Сдвиг влево	D = D << S
ROR/ROL	Циклический сдвиг вправо/влево	Циклически сдвинуть D на S разрядов
RCR/RCL	Циклический сдвиг вправо/влево через бит переноса	Циклически сдвинуть CF и D на S разрядов
BT	Проверка бита	CF = D[S] (бит номер S из D)
BTR/BTS	Проверить бит и сбросить/установить его	CF = D[S]; D[S] = 0 / 1
TEST	Установить флаги по результатам проверки бит	Установить флаги по результатам D AND S
MOV	Скопировать операнд	D = S
PUSH	Поместить в стек	ESP = ESP – 4; Mem[ESP] = S
POP	Прочитать из стека	D = MEM[ESP]; ESP = ESP + 4
CLC, STC	Сбросить/установить флаг переноса	CF = 0 / 1
JMP	Безусловный переход	Переход по относительному адресу: EIP = EIP + S Переход по абсолютному адресу: EIP = S
Jcc	Ветвление (условный переход)	Если установлен флаг, то EIP = EIP + S
LOOP	Проверка условия цикла	ECX = ECX – 1 Если ECX ≠ 0, то EIP = EIP + imm
CALL	Вызов функции	ESP = ESP – 4; MEM[ESP] = EIP; EIP = S
RET	Возврат из функции	EIP = MEM[ESP]; ESP = ESP + 4

[1] Если использовать все необязательные поля, то можно собрать команду длиной 17 байтов, но в x86 действует ограничение на длину допустимой команды – 15 байт.

Таблица 1.25. Некоторые условия ветвлений

Команда	Назначение	Операция
JZ/JE	Перейти, если ZF = 1	Перейти, если D = S
JNZ/JNE	Перейти, если ZF = 0	Перейти, если D ≠ S
JGE	Перейти, если SF = OF	Перейти, если D ≥ S
JG	Перейти, если SF = OF и ZF = 0	Перейти, если D > S
JLE	Перейти, если SF ≠ OF и ZF = 1	Перейти, если D ≤ S
JL	Перейти, если SF ≠ OF	Перейти, если D <S
JC/JB	Перейти, если CF = 1	
JNC	Перейти, если CF = 0	
JO	Перейти, если OF = 1	
JNO	Перейти, если OF = 0	
JS	Перейти, если SF = 1	
JNS	Перейти, если SF = 0	

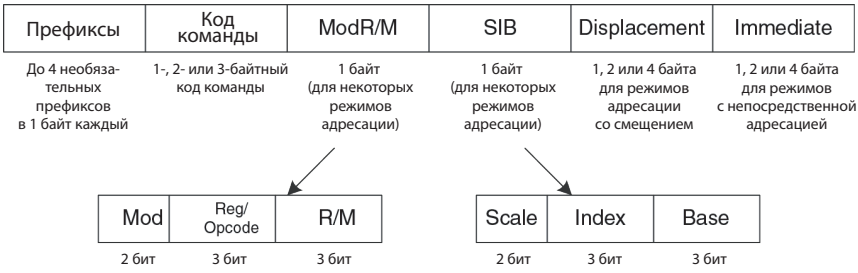


Рис. 1.36. Кодирование команд x86

Код операции может занимать 1, 2 или 3 байта. Далее следуют четыре дополнительных поля: *ModR/M*, *SIB*, *Displacement* и *Immediate*. Поле *ModR/M* определяет режим адресации. Поле *SIB* определяет коэффициент масштабирования, индексный и базовый регистры в некоторых режимах адресации. Поле *Displacement* содержит 1-, 2- или 4-байтовое смещение, используемое в соответствующих режимах адресации. Поле *Immediate* содержит 1-, 2- или 4-байтовую константу для команд, в которых присутствует непосредственный операнд. Кроме того, команда может иметь до четырех однобайтовых префиксов, изменяющих ее поведение.

Однобайтовое поле ModR/M состоит из 2-битового поля режима *Mod* и 3-битового поля *R/M* для задания режима адресации одного из

операндов. Операнд может находиться в одном из восьми регистров, либо его можно указать при помощи одного из 24 режимов адресации памяти. Исторически сложилось так, что регистры ESP и EBP нельзя использовать в качестве базового или индексного регистра в некоторых режимах адресации. В поле *Reg* указывается регистр, используемый в качестве второго операнда. Для некоторых команд, не имеющих второго операнда, поле *Reg* используется для хранения трех дополнительных бит кода операции.

В режимах адресации с масштабируемым индексным регистром байт *SIB* определяет индексный регистр и коэффициент масштабирования (1, 2, 4 или 8). Если используются и базовый адрес, и индекс, то *SIB* определяет также регистр базового адреса.

Архитектура ARM позволяет точно определить тип команды по полям *cond*, *op* и *funct*. В архитектуре x86 количество бит, определяющих команду, различно. Часто используемые команды имеют меньший размер, что уменьшает среднюю длину команд в программе. У некоторых команд может быть даже несколько кодов операций. Например, команда `add AL, imm8` выполняет 8-битовое сложение регистра AL и непосредственного операнда. Эта команда представляется в виде однобайтового кода операции (0x04) и однобайтового непосредственного операнда. Регистр `A` (`AL`, `AX` или `EAX`) называется *аккумулятором*. С другой стороны, команда `add D, imm8` производит 8-битовое сложение непосредственного операнда с произвольным операндом *D* (ячейкой памяти или регистром) и записывает результат в *D*. Эта команда кодируется однобайтовым кодом операции (0x08), за которым следует один или несколько байтов, определяющих местонахождение *D*, и однобайтовый непосредственный операнд *imm8*. Многие команды имеют более короткое представление в случае, если приемником является аккумулятор.

В оригинальном процессоре 8086 в коде операции указывалась разрядность операндов (8 или 16 бит). Когда в процессор 80386 добавили 32-битовые операнды, свободных кодов операции, которые позволили бы добавить новый размер операндов, уже не осталось, поэтому один и тот же код операции используется как для 16-, так и для 32-битовых вариантов команды. ОС использует дополнительный бит в *дескрипторе сегмента кода*, который определяет, какую форму команды должен выбрать процессор. Для обратной совместимости с программами, написанными для 8086, этот бит устанавливается в нуль, в результате чего все операнды по умолчанию считаются 16-битовыми. Если же этот бит равен единице, то используются 32-битовые операнды. Более того, программист может изменить форму конкретной команды при помощи префикса: если перед кодом операции добавить *префикс* 0x66, то будет использоваться альтернативный размер операндов (16 битов в 32-битовом режиме или 32 бита в 16-битовом режиме).

1.8.6. Другие особенности x86

В процессор 80286 был добавлен механизм *сегментации* для разделения памяти на сегменты размером до 64 КБ. Когда операционная система включала сегментацию, все адреса вычислялись относительно начала сегмента. Процессор проверял адреса и при выходе за пределы сегмента формировал сигнал ошибки, тем самым предотвращая доступ программ за пределы своего сегмента. Сегментация оказалась источником массы проблем для программистов и в современных версиях Windows не используется.

В середине 1990-х годов Intel и Hewlett-Packard совместно разработали новую 64-битовую архитектуру под названием IA-64. Она была спроектирована с чистого листа с использованием результатов исследований в области компьютерной архитектуры, полученных за двадцать лет, прошедших с момента появления 8086. Адресное пространство в ней было 64-битовым. Однако IA-64 так и не получила признания на рынке. В большинстве компьютеров, которым необходимо большое адресное пространство, используют 64-битовые расширения x86.

Архитектура x86 поддерживает команды, работающие со строками байтов или слов. К ним относятся команды копирования, сравнения и поиска заданного значения. В современных процессорах такие команды, как правило, работают медленнее, чем последовательность простых команд, делающих то же самое, поэтому их лучше избегать.

Как уже отмечалось, *префикс* 0x66 используется для выбора 16- или 32-битовых операндов. Есть и другие префиксы: для захвата внешней шины (это необходимо для обеспечения атомарного доступа к переменным в общей памяти в многопроцессорных системах), для предсказания ветвлений и для повторения команды при копировании строк.

Кошмар любой архитектуры – нехватка памяти. При 32-битовых адресах процессор x86 может адресовать 4 ГБ памяти. Это гораздо больше, чем было доступно самым большим компьютерам в 1985 году, но к началу 2000-х стало серьезным ограничением. В 2003 году компания AMD расширила адресное пространство и разрядность регистров до 64 бит и назвала новую архитектуру AMD64. У этой архитектуры был режим совместимости, в котором можно было запускать 32-битовые программы без какой-либо модификации, но при этом у ОС имелся доступ к большему адресному пространству. В 2004 году Intel сдалась и тоже включила 64-битовые расширения, назвав свою технологию Extended Memory 64 Technology (EM64T). При 64-битовой адресации компьютеру стало доступно 16 экзабайт (16 миллиардов ГБ) памяти.

Для интересующихся деталями архитектуры x86 на сайте Intel выложено бесплатное «Руководство разработчика программного обеспечения по архитектуре Intel» (Intel Architecture Software Developer's Manual).

1.8.7. Общая картина

В этом разделе мы рассмотрели основные отличия RISC-архитектуры ARM от CISC-архитектуры x86. Архитектура x86 позволяет писать бо-

лее короткие программы, потому что ее сложные команды эквивалентны нескольким простым командам ARM и вдобавок кодируются так, чтобы занимать минимум места в памяти. Однако архитектура x86 – это эклектичная смесь разнородных решений, накопленных за годы разработки. Некоторые из них давно уже бесполезны, но приходится сохранять их ради обратной совместимости со старыми программами. В этой архитектуре слишком мало регистров, а команды трудно декодировать. Даже просто объяснить, как устроен набор команд, нелегко. Но, несмотря на все недостатки, x86 остается доминирующей архитектурой персональных компьютеров, потому что совместимость программного обеспечения критически важна и потому что огромный рынок оправдывает затраты на разработку все более быстрых микропроцессоров, совместимых с x86.

ARM выдерживает баланс между простыми командами и плотным кодом путем включения таких возможностей, как флаги условий и сдвиг регистровых операндов. Благодаря им код, написанный для ARM, оказывается более компактным, чем в других RISC-архитектурах.

1.9. Резюме

Чтобы командовать компьютером, нужно разговаривать на его языке. Архитектура компьютера определяет, как именно нужно это делать. На сегодняшний день в мире существует много коммерчески используемых архитектур, но если вы хорошо понимаете одну из них, то изучить другие гораздо проще. При изучении новой архитектуры следует задать себе следующие главные вопросы:

- Какова длина слова данных?
- Какие есть регистры?
- Как организована память?
- Какие есть команды?

Архитектура ARM является 32-битовой, потому что работает с 32-битовыми данными. В архитектуре ARM определено 16 регистров: 15 регистров общего назначения и счетчик команд. В принципе, любой регистр общего назначения можно использовать для любой цели. Однако существуют соглашения, по которым определенные регистры зарезервированы для конкретных целей. Это сделано для того, чтобы облегчить процесс программирования и чтобы функции, написанные разными программистами, могли легко между собой взаимодействовать. Например, регистр R14 (регистр связи LR) всегда содержит адрес возврата после выполнения команды `BL`, а в регистрах R0–R3 хранятся аргументы функции. В архитектуре ARM применяются побайтовая адресация памяти и 32-битовые адреса. Длина каждой команды 32 бита, и все команды выровнены по границе 4-байтового слова. В этой главе мы обсудили наиболее часто используемые команды ARM.

Важность определения компьютерной архитектуры заключается в том, что программа, написанная для выбранной архитектуры, будет работать на совершенно разных реализациях этой архитектуры. Например, программы, написанные для процессора Intel Pentium в 1993 году, вообще говоря, будут работать (причем работать значительно быстрее) и на процессорах Intel Xeon или AMD Phenom в 2015 году.

В первой части этой книги мы узнали о схемных и логических уровнях абстракции. В этой главе мы занялись архитектурным уровнем, а в следующей изучим микроархитектуру – способ организации цифровых строительных блоков, из которых создается архитектура процессора. Микроархитектура – это мост между конструированием оборудования и программированием. По нашему мнению, изучение микроархитектуры является одним из наиболее захватывающих занятий для инженера: вы узнаете, как создать собственный микропроцессор!

Упражнения

Упражнение 1.1. Приведите по три примера каждого из принципов хорошей разработки в контексте архитектуры ARM: (1) единообразие способствует простоте, (2) типичный сценарий должен быть быстрым, (3) чем меньше, тем быстрее, (4) хороший проект требует хороших компромиссов. Поясните, как каждый пример иллюстрирует соответствующий принцип.

Упражнение 1.2. В архитектуре ARM имеется 16 32-битовых регистров. Можно ли спроектировать компьютерную архитектуру без регистров? Если можно, то кратко опишите такую архитектуру и ее систему команд. Каковы преимущества и недостатки такой архитектуры, по сравнению с архитектурой ARM?

Упражнение 1.3. Представьте себе 32-битовое слово, хранящееся в памяти с побайтовой адресацией и имеющее порядковый номер 42.

a) Каков байтовый адрес у слова памяти с порядковым номером 42?
b) На какие байтовые адреса распространяется это слово?
c) Предположим, что в этом слове хранится значение 0xFF223344. Изобразите графически, как это число хранится в байтах слова в случаях прямого и обратного порядка следования байтов. Ваш рисунок должен быть похож на **Рис. 1.4**. Выпишите байтовые адреса всех байтов данных.

Упражнение 1.4. Повторите **упражнение 1.3** для 32-битового слова, имеющего порядковый номер 15 в памяти.

Упражнение 1.5. Объясните, как использовать следующую программу, чтобы определить, является ли порядок следования байтов в данном компьютере прямым или обратным:

```
MOV  R0, #100
LDR  R1, =0xABCD876          ; R1 = 0xABCD876
STR  R1, [R0]
LDRB R2, [R0, #1]
```

Упражнение 1.6. Используя кодировку ASCII, запишите следующие строки в виде последовательностей шестнадцатеричных значений символов:

a) SOS
b) Cool!
c) boo!

Упражнение 1.7. Повторите **упражнение 1.6** для следующих строк:

a) howdy
b) ions
c) To the rescue!

Упражнение 1.8. Покажите, как строки из **упражнения 1.6** хранятся в памяти с побайтовой адресацией на машине с прямым порядком байтов, начиная с адреса 0x1000100C. Выпишите адрес каждого байта.

Упражнение 1.9. Повторите **упражнение 1.8** для строк из **упражнения 1.7**.

Упражнение 1.10. Преобразуйте следующий код из языка ассемблера ARM в машинный язык. Запишите команды в шестнадцатеричном виде.

```
MOV R10, #63488
LSL R9, R6, #7
STR R4, [R11, R8]
ASR R6, R7, R3
```

Упражнение 1.11. Повторите **упражнение 1.10** для следующего кода:

```
ADD R8, R0, R1
LDR R11, [R3, #4]
SUB R5, R7, #0x58
LSL R3, R2, #14
```

Упражнение 1.12. Рассмотрим команды обработки данных с непосредственным операндом в качестве *Src2*.

a) Какие команды из **упражнения 1.10** имеют такой формат?
b) Для каждой команды из части (a) выпишите 12-битовое поле *imm12*, а затем запишите их в виде 32-битовых непосредственных операндов.

Упражнение 1.13. Повторите **упражнение 1.12** для команд из **упражнения 1.11**.

Упражнение 1.14. Переведите следующую программу с машинного языка на язык ассемблера ARM. Числа слева – адреса команд в памяти. Числа справа – команды по соответствующим адресам. Затем напишите программу на языке высокого уровня, результатом компиляции которой будет эта программа на языке ассемблера. Объясните словами, что делает эта программа. Считайте, что входными данными являются регистры R0 и R1, которые первоначально содержат положительные числа `a` и `b`. После завершения программы результат помещается в R0.

```
0x00008008 0xE3A02000
0x0000800C 0xE1A03001
0x00008010 0xE1510000
0x00008014 0x8A000002
0x00008018 0xE2822001
0x0000801C 0xE0811003
0x00008020 0xEAFFFFFA
0x00008024 0xE1A00002
```

Упражнение 1.15. Повторите **упражнение 1.14** для приведенного ниже машинного кода. Считайте, что входные данные находятся в регистрах R0 и R1. R0 содержит некоторое 32-битовое число, а R1 – адрес некоторого массива из 32 символов (типа char).

```
0x00008104 0xE3A0201F
0x00008108 0xE1A03230
0x0000810C 0xE2033001
0x00008110 0xE4C13001
0x00008114 0xE2522001
0x00008118 0x5AFFFFFA
0x0000811C 0xE1A0F00E
```

Упражнение 1.16. В наборе команд ARM нет команды `NOR`, потому что ее функциональность можно реализовать с помощью существующих команд. Напишите короткий ассемблерный код, реализующий функциональность R0 = R1 NOR R2. Используйте как можно меньше команд.

Упражнение 1.17. В наборе команд ARM нет команды `NAND`, потому что ее функциональность можно реализовать с помощью существующих команд. Напишите короткий ассемблерный код, реализующий функциональность R0 = R1 NAND R2. Используйте как можно меньше команд.

Упражнение 1.18. Ниже приведено два фрагмента кода на языке высокого уровня. Предположим, что целые переменные со знаком `g` и `h` находятся в регистрах R0 и R1 соответственно.

i)
```
if (g > h)
   g = g + h;
else
   g = g - h;
```

ii)
```
if (g < h)
   h = h + 1;
else
   h = h * 2;
```

a) Перепишите этот код на языке ассемблера ARM в предположении, что условное выполнение разрешено только в командах перехода. Используйте как можно меньше команд (в рамках этого предположения).
b) Перепишите этот код на языке ассемблера ARM в предположении, что условное выполнение разрешено во всех командах. Используйте как можно меньше команд.
c) Сравните плотность кода (количество команд) в случаях a) и b) и обсудите плюсы и минусы.

Упражнение 1.19. Повторите **упражнение 1.18** для следующих фрагментов кода:

i)
```
if (g > h)
   g = g + 1;
else
   h = h - 1;
```

ii)
```
if (g <= h)
   g = 0;
else
   h = 0;
```

Упражнение 1.20. Ниже приведен фрагмент кода на языке высокого уровня. Предположим, что базовые адреса массивов `array1` и `array2` находятся в регистрах R1 и R2 и что `array2` предварительно инициализирован.

```
int i;
int array1[100];
int array2[100];
...
for (i=0; i<100; i=i+1)
    array1[i] = array2[i];
```

a) Перепишите этот код на языке ассемблера ARM, не используя ни пред-, ни постиндексирования, ни масштабируемых регистров. Используйте как можно меньше команд (в рамках этого ограничения).
b) Перепишите этот код на языке ассемблера ARM, пользуясь имеющимися возможностями пред- и постиндексирования и масштабирования регистров. Используйте как можно меньше команд.
c) Сравните плотность кода (количество команд) в случаях a) и b) и обсудите плюсы и минусы.

Упражнение 1.21. Повторите **упражнение 1.20** для следующего фрагмента кода. Предполагается, что массив `temp` предварительно инициализирован и что в регистре R3 находится базовый адрес `temp`.

```
int i;
int temp[100];
...
for (i=0; i<100; i=i+1)
   temp[i] = temp[i] * 128;
```

Упражнение 1.22. Ниже приведено два фрагмента кода. Предположим, что в R1 находится `i`, а в `R0` базовый адрес массива `vals`.

i)
```
int i;
int vals[200];

for (i=0; i < 200; i=i+1)
   vals[i] = i;
```

ii)
```
int i;
int vals[200];

for (i=199; i >= 0; i = i-1)
   vals[i] = i;
```

a) Верно ли, что эти фрагменты функционально эквивалентны?
b) Перепишите оба фрагмента на языке ассемблера ARM. Используйте как можно меньше команд.
c) Обсудите плюсы и минусы каждой конструкции.

Упражнение 1.23. Повторите **упражнение 1.22** для следующих фрагментов кода. Предполагается, что в R1 находится `i`, в `R0` базовый адрес массива `nums` и что массив предварительно инициализирован.

i)
```
int i;
int nums[10];
...
for (i=0; i < 10; i=i+1)
  nums[i] = nums[i]/2;
```

ii)
```
int i;
int nums[10];
...
for (i=9; i >= 0; i = i-1)
  nums[i] = nums[i]/2;
```

Упражнение 1.24. Напишите на языке высокого уровня функцию с прототипом `int find42(int array[], int size)`. Здесь `array` – базовый адрес массива, а `size` – число элементов этого массива. Функция должна возвращать порядковый индекс первого элемента массива, со-

держащего значение 42. Если в массиве нет числа 42, то функция должна вернуть −1.

Упражнение 1.25. Функция `strcpy` копирует строку символов, расположенную в памяти по адресу `src`, в новое место с адресом `dst`.

```
// Код на C
void strcpy(char dst[], char src[]) {
  int i = 0;
  do {
    dst[i] = src[i];
  } while (src[i++]);
}
```

a) Реализуйте функцию `strcpy` на языке ассемблера ARM. Используйте для `i` регистр R4.
b) Изобразите стек до, во время и после вызова функции `strcpy`. Считайте, что перед вызовом `strcpy` SP = 0xBEFFF000.

Эта простая функция копирования строки имеет один весьма серьезный изъян: она не знает, зарезервировано ли по адресу `dst` достаточно места для копии строки. Если хакер сумеет заставить программу выполнить функцию `strcpy` в ситуации, когда по адресу `src` находится длинная строка, то `strcpy` сможет изменить данные и даже команды в области программы, находящейся после зарезервированного участка памяти. При некотором навыке модифицированный код может перехватить управление компьютером. Это так называемая *атака с переполнением буфера*. Она используется несколькими вредоносными программами, в частности печально известным «червем» Blaster, который причинил ущерб приблизительно на 525 млн долларов в 2003 году.

Упражнение 1.26. Реализуйте функцию из **упражнения 1.24** на языке ассемблера ARM.

Упражнение 1.27. Рассмотрим приведенный ниже код на языке ассемблера ARM. Функции `func1`, `func2` и `func3` – нелистовые, а `func4` – листовая. Полный код функций не показан, но в комментариях указаны регистры, используемые в каждой из них.

```
0x00091000 func1 ... ; func1 использует R4–R10
0x00091020 BL func2
. . .
0x00091100 func2 ... ; func2 использует R0–R5
0x0009117C BL func3
. . .
0x00091400 func3 ... ; func3 использует R3, R7–R9
0x00091704 BL func4
. . .
0x00093008 func4 ... ; func4 использует R11–R12
0x00093118 MOV PC, LR
```

a) Сколько слов занимает кадр стека каждой из этих функций?
b) Изобразите стек после вызова `func4`. Укажите, какие регистры хранятся в стеке и где именно. Отметьте каждый кадр стека. Там, где возможно, приведите значения, находящиеся в стеке.

Упражнение 1.28. Каждое число последовательности Фибоначчи является суммой двух предыдущих чисел. В **Табл. 1.26** приведены первые числа последовательности, $fib(n)$.

Таблица 1.26. Последовательность чисел Фибоначчи

n	1	2	3	4	5	6	7	8	9	10	11	...
fib(n)	1	1	2	3	5	8	13	21	34	55	89	...

a) Чему равны значения $fib(n)$ для $n = 0$ и $n = -1$?
b) Напишите на языке высокого уровня функцию *fib*, которая возвращает число Фибоначчи с заданным неотрицательным номером *n*. *Подсказка*: используйте цикл. Прокомментируйте свой код.
c) Преобразуйте функцию, написанную в части b), в код на языке ассемблера ARM. Для каждой строки кода добавьте комментарий, поясняющий, что в этой строке делается. Протестируйте код для $fib(9)$, воспользовавшись симулятором Keil MDK-ARM. (О том, как установить этот симулятор, написано в предисловии.)

Упражнение 1.29. Обратимся к **примеру кода 1.27**. В этом упражнении предположим, что функция $factorial(n)$ вызывается с аргументом $n = 5$.

a) Чему будет равен регистр R0, когда функция *factorial* вернет управление вызвавшей ее функции?
b) Предположим, что команды, расположенные по адресам 0x8500 и 0x8520, заменены командами `PUSH {R0, R1}` и `POP {R1, R2}` соответственно. Что произойдет с программой:
   1) войдет в бесконечный цикл, но не завершится аварийно;
   2) завершится аварийно (стек выйдет за пределы динамического сегмента данных или счетчик команд станет указывать на адрес, не принадлежащий программе);
   3) вернет неправильное значение в R0, когда программа вернется в цикл (если да, то какое?);
   4) продолжит работать правильно, несмотря на изменения?
c) Повторите часть (b), произведя следующие модификации:
   i) команды, расположенные по адресам 0x8500 и 0x8520, заменены командами `PUSH {R3, LR}` и `POP {R3, LR}` соответственно;
   ii) команды, расположенные по адресам 0x8500 и 0x8520, заменены командами `PUSH {LR}` и `POP {LR}` соответственно;
   iii) команда, расположенная по адресу 0x8510, удалена.

Упражнение 1.30. Бен Битдидл пытался вычислить функцию $f(a,b) = 2a + 3b$ для неотрицательного значения *b*, но переусердствовал с вызовами функций и рекурсией и написал вот такой код:

```
// код функций f и g на языке высокого уровня
int f(int a, int b) {
```

```
  int j;

  j = a;

  return j + a + g(b);
}
int g(int x) {
  int k;
  k = 3;

  if (x == 0) return 0;
  else return k + g(x - 1);
}
```

После этого Бен транслировал обе функции на язык ассемблера. Он также написал функцию `test`, которая вызывает функцию `f(5,3)`.

```
; код на языке ассемблера ARM
; f: R0 = a, R1 = b, R4 = j;
; g: R0 = x, R4 = k

0x00008000 test     MOV R0, #5          ; a = 5
0x00008004          MOV R1, #3          ; b = 3
0x00008008          BL  f               ; вызвать f(5, 3)
0x0000800C loop     B   loop            ; и циклиться бесконечно
0x00008010 f        PUSH {R1,R0,LR,R4}  ; сохранить регистры в стеке
0x00008014          MOV R4, R0          ; j = a
0x00008018          MOV R0, R1          ; поместить b как аргумент g
0x0000801C          BL  g               ; call g(b)
0x00008020          MOV R2, R0          ; поместить возвращаемое значение в R2
0x00008024          POP {R1,R0}         ; восстановить a и b после вызова
0x00008028          ADD R0, R2, R0      ; R0 = g(b) +  a
0x0000802C          ADD R0, R0, R4      ; R0 = (g(b) + a) + j
0x00008030          POP {R4,LR}         ; восстановить R4, LR
0x00008034          MOV PC, LR          ; возврат
0x00008038 g        PUSH {R4,LR}        ; сохранить регистры в стеке
0x0000803C          MOV R4, #3          ; k = 3
0x00008040          CMP R0, #0          ; x == 0?
0x00008044          BNE else            ; перейти, если не равно
0x00008048          MOV R0, #0          ; если равно, вернуть  0
0x0000804C          B   done            ; и прибраться за собой
0x00008050 else     SUB R0, R0, #1      ; x =  x - 1
0x00008054          BL  g               ; call g(x - 1)
0x00008058          ADD R0, R0, R4      ; R0 =  g(x - 1) + k
0x0000805C done     POP {R4,LR}         ; восстановить R0,R4,LR из стека
0x00008060          MOV PC, LR          ; возврат
```

Возможно, будет полезно изобразить стек, как на **Рис. 1.14**. Это поможет ответить на следующие вопросы:

а) Если код начнет выполнение с метки `test`, то какое значение окажется в регистре R0, когда программа дойдет до метки `loop`? Правильно ли программа вычислит $2a + 3b$?

b) Предположим, что Бен заменил команды по адресам 0x00008010 и 0x00008030 командами `PUSH {R1,R0,R4}` и `POP {R4}` соответственно. Что произойдет с программой:
   1) войдет в бесконечный цикл, но не завершится аварийно;
   2) завершится аварийно (стек выйдет за пределы динамического сегмента данных или счетчик команд станет указывать на адрес, не принадлежащий программе);
   3) вернет неправильное значение в R0, когда программа вернется к метке `loop` (если да, то какое?);
   4) продолжит работать правильно, несмотря на изменения?

c) Повторите часть b) при следующих изменениях. Обратите внимание, что изменяются только команды, но не метки:
   i) команды по адресам 0x00008010 и 0x00008024 заменяются на `PUSH {R1,LR,R4}` и `POP {R1}` соответственно;
   ii) команды по адресам 0x00008010 и 0x00008024 заменяются на `PUSH {R0,LR,R4}` и `POP {R0}` соответственно;
   iii) команды по адресам 0x00008010 и 0x00008030 заменяются на `PUSH {R1,R0,LR}` и `POP {LR}` соответственно;
   iv) команды по адресам 0x00008010, 0x00008024 и 0x00008030 удаляются;
   v) команды по адресам 0x00008038 и 0x0000805C заменяются на `PUSH {R4}` и `POP {R4}` соответственно;
   vi) команды по адресам 0x00008038 и 0x0000805C заменяются на `PUSH {LR}` и `POP {LR}` соответственно;
   vii) команды по адресам 0x00008038 и 0x0000805C удаляются.

Упражнение 1.31. Переведите приведенные ниже команды перехода в машинный код. Адреса команд указаны слева:

a)
```
0x0000A000         BEQ LOOP
0x0000A004         ...
0x0000A008         ...
0x0000A00C LOOP    ...
```

b)
```
0x00801000         BGE DONE
...
0x00802040 DONE    ...
```

c)
```
0x0000B10C BACK    ...
...                ...
0x0000D000         BHI BACK
```

d)
```
0x00103000         BL FUNC
...                ...
0x0011147C FUNC    ...
```

e)
```
0x00008004 L1      ...
...                ...
0x0000F00C         B L1
```

Упражнение 1.32. Рассмотрим следующий фрагмент кода на языке ассемблера ARM. Слева от каждой команды указан ее адрес:

```
0x000A0028 FUNC1        MOV R4, R1
0x000A002C              ADD R5, R3, R5, LSR #2
0x000A0030              SUB R4, R0, R3, ROR R4
0x000A0034              BL FUNC2
...                     ...
0x000A0038 FUNC2        LDR R2, [R0, #4]
0x000A003C              STR R2, [R1, -R2]
0x000A0040 CMP          R3, #0
0x000A0044              BNE ELSE
0x000A0048              MOV PC, LR
0x000A004C ELSE         SUB R3, R3, #1
0x000A0050              B FUNC2
```

a) Транслируйте эту последовательность команд в машинный код, представленный в шестнадцатеричном виде.
b) В каждой строке кода укажите использованный в ней режим адресации.

Упражнение 1.33. Рассмотрим следующий фрагмент кода на C:

```
// Код на C
void setArray(int num) {
  int i;
  int array[10];

  for (i = 0; i < 10; i = i + 1)
    array[i] = compare(num, i);
}
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;
}
```

a) Перепишите этот код на языке ассемблера ARM. Используйте регистр R4 для хранения переменной `i`. Следите за правильностью работы с указателем стека. Массив хранится в стеке функции `setArray` (см. конец **раздела 1.3.7**).
b) Предположим, что первой вызвана функция `setArray`. Изобразите состояние стека перед вызовом `setArray` и в процессе каждого последующего вызова функции. Укажите имена регистров и переменных, находящихся в стеке. Отметьте, куда указывает SP, и обозначьте все кадры стека.

c) Как бы работал ваш код, если бы вы забыли сохранить в стеке регистр LR?

Упражнение 1.34. Рассмотрим следующую функцию на языке высокого уровня:

```
// Код на C
int f(int n, int k) {
  int b;

  b = k + 2;
  if (n == 0) b = 10;
  else b = b + (n * n) + f(n - 1, k + 1);
  return b * k;
}
```

a) Транслируйте функцию `f` на язык ассемблера ARM. Обратите особое внимание на правильность сохранения и восстановления регистров между вызовами функций, а также на использование соглашений, касающихся оберегаемых регистров. Тщательно прокомментируйте код. Можете использовать команду `MUL`. Функция начинается с адреса 0x00008100. Локальную переменную `b` храните в регистре R4.

b) Вручную пошагово выполните функцию из пункта a) для случая `f(2, 4)`. Изобразите стек, как на **Рис. 1.14**, в предположении, что в момент вызова `f` SP содержит 0xBFF00100. Выпишите имена и значения регистров, хранящихся в каждом слове стека, и проследите за изменением указателя стека (SP). Обозначьте каждый кадр стека. Возможно, вам покажется полезным проследить также значения регистров R0, R1 и R4 в процессе выполнения программы. Предполагается, что в момент вызова `f` значение R4 = 0xABCD, а LR = 0x400004. Каким будет конечный результат в регистре R0?

Упражнение 1.35. Приведите пример худшего случая для перехода вперед (т. е. на команду с большим адресом). Худшим считается случай, когда нельзя перейти далеко. Выпишите сами команды и их адреса.

Упражнение 1.36. Приведенные ниже вопросы помогут вам лучше понять работу команды безусловного перехода, `B`. Ответы должны содержать количество команд относительно команды перехода.

a) Как далеко вперед (то есть по направлению к большим адресам) может перейти команда `B` в худшем случае? Худшим считается случай, когда нельзя перейти далеко. Объясните словами, используя по необходимости примеры.

b) Как далеко вперед может перейти команда `B` в лучшем случае? Лучшим считается случай, когда можно перейти максимально далеко. Поясните ответ.

c) Как далеко назад (то есть по направлению к меньшим адресам) может перейти команда `B` в худшем случае? Поясните ответ.
d) Как далеко назад может перейти команда `B` в лучшем случае? Поясните ответ.

Упражнение 1.37. Объясните, почему выгодно иметь большое поле непосредственного адреса *imm24* в машинном формате команд безусловного перехода `B` и `BL`.

Упражнение 1.38. Напишите на языке ассемблера код, который переходит к команде, отстоящей на 32 мегакоманды от начала этого кода. Напомним, что 1 мегакоманда = 2^{20} команд = 1 048 576 команд. Предполагается, что код начинается с адреса 0x00008000. Используйте минимальное количество команд.

Упражнение 1.39. Напишите на языке высокого уровня функцию, которая принимает массив, содержащий десять 32-разрядных целых чисел в прямом порядке следования байтов (от младшего к старшему) и преобразует его в формат с обратным порядком (от старшего к младшему). Перепишите код на языке ассемблера ARM. Прокомментируйте свой код и используйте минимальное количество команд.

Упражнение 1.40. Рассмотрим две строки: `string1` и `string2`.

a) Напишите на языке высокого уровня функцию `concat`, которая конкатенирует их (склеивает вместе): `void concat(char string1[], char string2[], char string concat[])`. Отметим, что эта функция не возвращает значения (т. е. тип возвращаемого значения равен `void`). Результат конкатенации `string1` и `string2` помещается в строку в `stringconcat`. Предполагается, что массив символов `stringconcat` достаточно велик для размещения результата.
b) Перепишите код из части a) на языке ассемблера ARM.

Упражнение 1.41. Напишите на языке ассемблера ARM программу, которая складывает два положительных числа с плавающей точкой одинарной точности, находящихся в регистрах R0 и R1. Не используйте специальные команды для работы с плавающей точкой. В этом упражнении можете не беспокоиться о кодах значений, зарезервированных для специальных целей (например, 0, NaN и т. д.), а также о возможных переполнениях или потере значимости. Используйте симулятор Keil MDK-ARM для тестирования кода. Для этого нужно будет вручную установить значения R0 и R1. Продемонстрируйте, что ваш код работает надежно.

Упражнение 1.42. Рассмотрим приведенную ниже программу на языке ассемблера ARM. Предположим, что команды начинаются с адреса 0x8400 и что метке `L1` соответствует адрес 0x9024.

```
; код на языке ассемблера ARM
MAIN
```

```
  PUSH {LR}
  LDR  R2, =L1 ; транслируется в команду загрузки относительно PC
  LDR  R0, [R2]
  LDR  R1, [R2, #4]
  BL   DIFF
  POP  {LR}
  MOV  PC, LR
DIFF
  SUB  R0, R0, R1
  MOV  PC, LR
  ...
L1
```

a) Сначала выпишите рядом с каждой командой ее адрес.
b) Составьте таблицу символов, т. е. соответствие между метками и адресами.
c) Преобразуйте все команды в машинный код.
d) Укажите размер сегментов данных и кода в байтах.
e) Нарисуйте карту памяти (по аналогии с **Рис. 1.31**) и укажите, где хранятся данные и команды.

Упражнение 1.43. Повторите **упражнение 1.42** для показанного ниже кода. Предположим, что команды начинаются с адреса 0x8534 и что метке `L2` соответствует адрес 0x9024.

```
; код на языке ассемблера ARM
MAIN
  PUSH  {R4,LR}
  MOV   R4, #15
  LDR   R3, =L2 ; транслируется в команду загрузки относительно PC
  STR   R4, [R3]
  MOV   R1, #27
  STR   R1, [R3, #4]
  LDR   R0, [R3]
  BL    GREATER
  POP   {R4,LR}
  MOV   PC, LR
GREATER
  CMP   R0, R1
  MOV   R0, #0
  MOVGT R0, #1
  MOV   PC, LR
  ...
L2
```

Упражнение 1.44. Назовите две команды ARM, способные повысить плотность кода (т. е. уменьшить число команд в программе). Приведите примеры обеих, продемонстрировав эквивалентный код на языке ассемблера ARM, включающий и не включающий эти команды.

Упражнение 1.45. Объясните преимущества и недостатки условного выполнения.

Вопросы для собеседования

Приведенные ниже вопросы задавали на собеседованиях с кандидатами на вакансии разработчиков цифровой аппаратуры (но относятся они к любым языкам ассемблера).

Вопрос 1.1. Напишите на ассемблере ARM программу, которая меняет местами содержимое регистров R0 и R1. Использовать другие регистры не разрешается.

Вопрос 1.2. Предположим, что имеется массив, содержащий положительные и отрицательные целые числа. Напишите на языке ассемблера ARM программу, которая находит подмножество массива с максимальной суммой. Предполагается, что адрес массива и количество элементов хранятся в регистрах R0 и R1 соответственно. Программа должна разместить найденное подмножество массива, начиная с адреса, находящегося в регистре R2. Код должен работать максимально быстро.

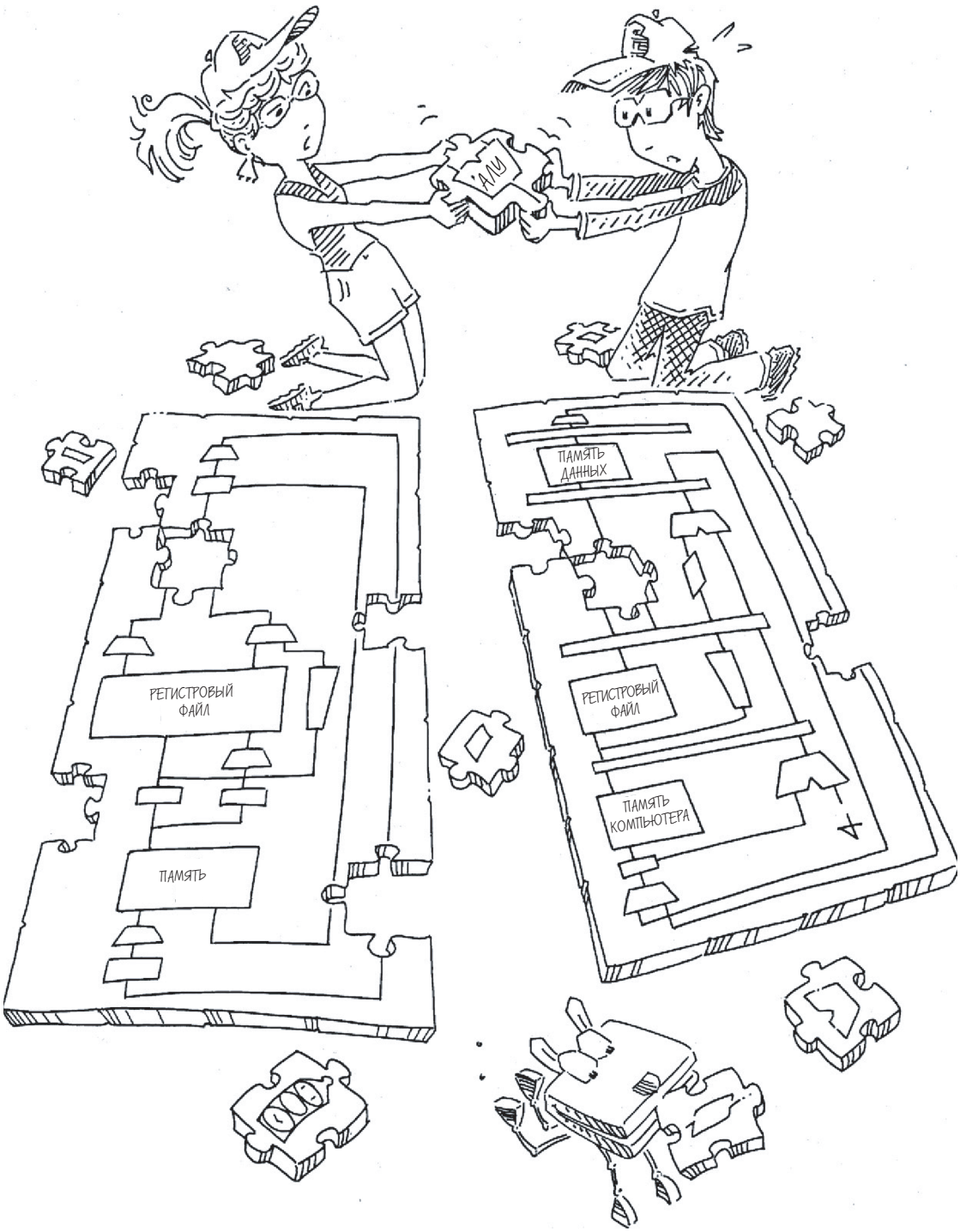
Вопрос 1.3. Дан массив, содержащий строку в смысле языка C. Строка содержит предложение. Придумайте алгоритм, который запишет слова предложения в обратном порядке и сохранит результат в тот же массив. Реализуйте свой алгоритм на языке ассемблера ARM.

Вопрос 1.4. Придумайте алгоритм подсчета единиц в 32-битовом числе. Реализуйте свой алгоритм на языке ассемблера ARM.

Вопрос 1.5. Напишите на языке ассемблера ARM программу, меняющую порядок битов в регистре на обратный. Используйте как можно меньше команд. Исходное значение хранится в регистре R3.

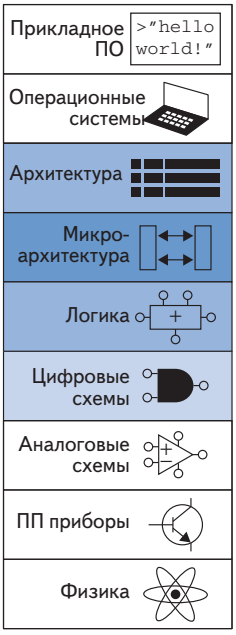
Вопрос 1.6. Напишите на языке ассемблера ARM программу, проверяющую, произошло ли переполнение при сложении регистров R2 и R3. Используйте как можно меньше команд.

Вопрос 1.7. Придумайте алгоритм, который проверяет, является ли заданная строка палиндромом (*палиндромом* называется слово, которое читается в обоих направлениях одинаково, например «wow» или «racecar»). Напишите на языке ассемблера ARM программу, реализующую этот алгоритм.


АЛУ
ПАМЯТЬ ДАННЫХ
РЕГИСТРОВЫЙ ФАЙЛ
ПАМЯТЬ КОМПЬЮТЕРА
РЕГИСТРОВЫЙ ФАЙЛ
ПАМЯТЬ

Глава 2

Микроархитектура



2.1. Введение

В этой главе мы узнаем, как собрать микропроцессор. На самом деле мы соберем три разные версии, отличающиеся соотношением производительности, цены и сложности.

Непосвященному создание микропроцессора может показаться черной магией. На самом деле это относительно простое дело, и сейчас мы уже знаем все, что нужно. В частности, мы изучили разработку комбинационных и последовательных схем по заданным функциональным и временным спецификациям. Мы познакомились с построением арифметических схем и блоков памяти. Мы также изучили архитектуру ARM, описывающую процессор с точки зрения программиста: в виде совокупности регистров, команд и памяти.

Эта глава посвящена *микроархитектуре* – связующему звену между логическими схемами и архитектурой. Микроархитектура описывает конкретную организацию регистров, АЛУ, конечных автоматов, блоков

памяти и других строительных блоков, необходимых для реализации архитектуры. Для каждой архитектуры, включая ARM, может быть много различных микроархитектур, обеспечивающих разное соотношение производительности, цены и сложности. Все они смогут выполнять одни и те же программы, но их внутреннее устройство может сильно различаться. В этой главе мы спроектируем три различные микроархитектуры, чтобы проиллюстрировать компромиссы, на которые приходится идти разработчику.

2.1.1. Архитектурное состояние и набор команд

Напомним, что компьютерная архитектура определяется набором команд и архитектурным состоянием. *Архитектурное состояние* процессора ARM включает 16 32-битовых регистров и регистр состояния. Любая микроархитектура ARM обязана содержать все это состояние. Зная текущее архитектурное состояние, процессор выполняет определенную команду в применении к конкретным данным и получает новое архитектурное состояние. В некоторых микроархитектурах есть дополнительное *неархитектурное* состояние, которое используется либо для упрощения логики, либо для повышения производительности; мы обратим на это внимание, когда придет время.

Чтобы не усложнять микроархитектуру, мы рассмотрим только подмножество набора команд ARM, а именно:

- команды обработки данных: `ADD`, `SUB`, `AND`, `ORR` (с регистровым и непосредственным режимом адресации, но без сдвигов);
- команды доступа к памяти: `LDR`, `STR` (с положительным смещением, заданным непосредственным операндом);
- команды перехода: `B`.

Эти команды были выбраны, потому что их достаточно для написания многих интересных программ. Поняв, как реализовать эти команды в аппаратуре, вы сможете добавить и другие.

2.1.2. Процесс проектирования

Мы разделим нашу микроархитектуру на две взаимодействующие части: *тракт данных* и *устройство управления*. Тракт данных работает со словами данных. Он содержит такие блоки, как память, регистры, АЛУ и мультиплексоры. ARM – 32-битовая архитектура, так что мы будем использовать 32-битовый тракт данных. Устройство управления получает текущую команду из тракта данных и в ответ сообщает, как именно выполнять эту команду. В частности, устройство управления генериру-

ет адресные сигналы для мультиплексоров, сигналы разрешения работы для регистров и сигналы разрешения записи в память.

При проектировании сложной системы стоит начинать с элементов, в которых хранится ее состояние. Это память и архитектурное состояние (счетчик команд, регистры и регистр состояния). Затем между этими элементами следует расположить комбинационные схемы, вычисляющие новое состояние на основе текущего. Команда читается из одной части памяти, а команды загрузки и сохранения читают или записывают данные в другую часть памяти. Поэтому зачастую бывает удобно разделить память на две меньшие по размеру части, одна из которых содержит команды, а другая – данные. На **Рис. 2.1** показаны пять элементов состояния: счетчик команд, регистровый файл, регистр состояния, память команд и память данных.

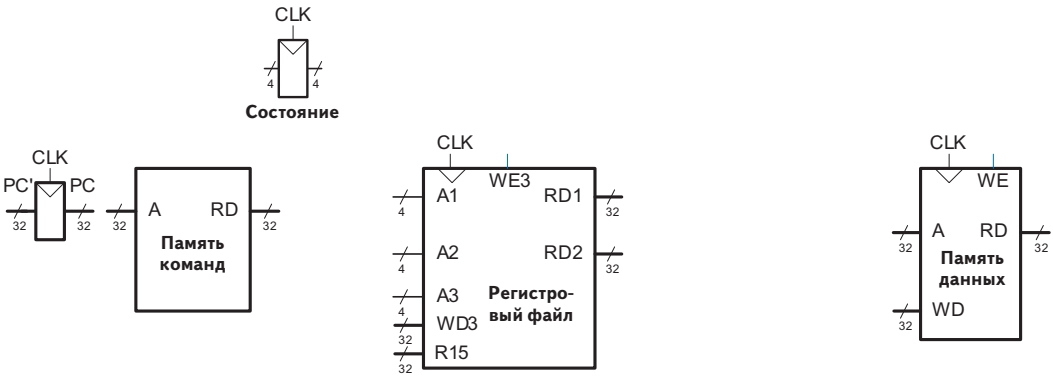


Рис. 2.1. Элементы состояния процессора ARM

На **Рис. 2.1** жирными линиями обозначены 32-битовые шины данных. Средними линиями обозначаются шины меньшей разрядности, например 4-битовые шины адреса регистрового файла. Тонкими линиями обозначаются 1-битовые шины, а синими линиями – шины управляющих сигналов, например сигнала разрешения записи в регистровый файл. Мы будем придерживаться этого соглашения и дальше, чтобы избежать загромождения диаграмм указанием разрядности шин. Кроме того, у элементов состояния обычно имеется сигнал сброса, который переводит их в известное состояние в момент включения. Чтобы не загромождать диаграммы, мы не будем его показывать.

Хотя *счетчик команд* (PC) логически является частью регистрового файла, он читается и записывается на каждом такте независимо от нормальных операций с регистровым файлом, поэтому более естественно конструировать его как автономный 32-битовый регистр.

Сброс счетчика команд

Как минимум, счетчик команд должен иметь сигнал сброса, по которому он будет инициализирован в момент включения процессора. Процессоры ARM обычно инициализируют PC значением `0x00000000` при получении сигнала сброса, с этого адреса и будут начинаться наши программы.

Трактовка PC как части регистрового файла усложняет проектирование системы и в конечном итоге приводит к большему числу логических вентилей и повышению энергопотребления. В большинстве других архитектур PC рассматривается как специальный регистр, обновляемый только командами перехода, а не обычными командами обработки данных. Как было сказано в **разделе 1.7.6**, в 64-битовой архитектуре ARMv8 счетчик команд PC также считается специальным регистром, отдельным от регистрового файла.

Его выход, *PC*, содержит адрес текущей команды, а вход, *PC'*, содержит адрес следующей команды.

Память команд имеет только порт чтения[1]. На адресный вход *A* подается 32-битовый адрес команды, после чего на выходе *RD* появляется 32-битовое число (т. е. команда), прочитанное из памяти по этому адресу.

Регистровый файл содержит 15 элементов по 32 бита каждый, соответствующих регистрам R0–R14, и дополнительный вход для получения R15 из PC. Он имеет два порта чтения и один порт записи данных. Порты чтения имеют 4-битовые входы адреса, *A1* и *A2*, каждый из которых определяет один из $2^4 = 16$ регистров в качестве операнда-источника. Каждый из портов читает 32-битовое значение из регистра и подает его на выходы *RD1* и *RD2* соответственно. Порт записи получает 4-битовый адрес регистра на адресный вход *A3*, 32-битовое число на вход данных *WD3*, сигнал разрешения записи *WE3* и тактовый сигнал. Если сигнал разрешения записи равен единице, то регистровый файл записывает данные в указанный регистр по положительному фронту тактового сигнала. Чтение из регистра R15 возвращает значение счетчика команд плюс 8, а запись в R15 следует обрабатывать специально, поскольку при этом обновляется счетчик команд PC, отдельный от регистрового файла.

Память данных имеет один порт чтения / записи. Если сигнал разрешения записи *WE* равен единице, то данные с входа *WD* записываются в ячейку памяти с адресом *A* по положительному фронту тактового сигнала. Если же сигнал разрешения записи равен нулю, то данные из ячейки с адресом *A* подаются на выход *RD*.

Чтение из памяти команд, регистрового файла и памяти данных происходит *комбинационно*. Другими словами, сразу же после изменения значения на адресном входе на выходе *RD* появляются новые данные. Это происходит не мгновенно, т. к. существует задержка распространения сигнала, однако сигнал синхронизации для чтения не требуется. Запись же производится исключительно по положительному фронту сигнала синхронизации. Таким образом, состояние системы изменяется только по фронту сигнала синхронизации. Адрес, данные и сигнал разрешения записи должны стать корректными за некоторое время до прихода фронта сигнала синхронизации (время предустановки, setup) и ни в коем случае не должны изменяться до тех пор, пока не пройдет некоторое время после прихода фронта (время удержания, hold).

[1] Это упрощение сделано для того, чтобы можно было считать память команд постоянной памятью (ROM); в большинстве реальных процессоров память команд должна быть доступна и для записи, чтобы операционная система могла загружать в нее новые программы. Многотактная микроархитектура, описанная в разделе 2.4, более реалистична в этом плане, т. к. в ней используется объединенная память команд и данных, доступная как для чтения, так и для записи.

В связи с тем, что элементы памяти изменяют состояние только по положительному фронту сигнала синхронизации, они являются синхронными последовательными схемами. Микропроцессор строится из тактируемых элементов состояния и комбинационной логики, поэтому он тоже является синхронной последовательной схемой. На самом деле процессор можно рассматривать как гигантский конечный автомат или как несколько более простых взаимодействующих между собой конечных автоматов.

Примерами классических многотактных процессоров могут служить проект Whirlwind, созданный в МТИ в 1947 году, IBM System/360, процессор VAX компании Digital Equipment Corporation, микропроцессор 6502, использовавшийся в компьютерах Apple II, и микропроцессор 8088, применявшийся в IBM PC. Многотактные микроархитектуры все еще используются в недорогих микроконтроллерах, например 8051, 68HC11 и устройствах серии PIC16, которые встречаются в бытовых приборах, игрушках и гаджетах.

2.1.3. Микроархитектуры

В этой главе мы разработаем три микроархитектуры для процессорной архитектуры ARM: однотактную, многотактную и конвейерную. Они различаются способом соединения элементов состояния, а также объемом неархитектурного состояния.

В однотактной микроархитектуре вся команда выполняется за один такт. Ее принцип работы легко объяснить, а устройство управления довольно простое. Из-за того, что все действия выполняются за один такт, эта микроархитектура не требует никакого неархитектурного состояния. Однако длительность такта ограничена самой медленной командой. Кроме того, процессор требует наличия раздельной памяти для команд и данных, что в общем случае нереально.

В *многотактной микроархитектуре* команда выполняется за несколько более коротких тактов. Простым командам нужно меньше тактов, чем сложным. Вдобавок многотактная микроархитектура уменьшает количество необходимого оборудования благодаря повторному использованию таких дорогих блоков, как сумматоры и блоки памяти. Например, при выполнении команды один и тот же сумматор на разных тактах может быть использован для разных целей. Повторное использование блоков достигается путем добавления в многотактный процессор нескольких неархитектурных регистров для сохранения промежуточных результатов. Многотактный процессор выполняет только одну команду за раз, но каждая команда занимает несколько тактов. Многотактному процессору нужна только одна память, к которой он на одном такте обращается для выборки команды, а на другом – для чтения или записи данных. Поэтому исторически многотактные процессоры использовались в недорогих системах.

Процессоры Intel стали конвейерными, когда в 1989 году был выпущен 80486. Конвейерными являются также почти все микропроцессоры RISC. Процессоры ARM были конвейерными, начиная с самого первого ARM1, выпущенного в 1985 году. Конвейерный ARM Cortex-M0 насчитывает всего около 12 000 логических вентилей, поэтому при современном развитии технологии производства интегральных схем он настолько мал, что разглядеть его можно только в микроскоп, а его изготовление обходится в сущие копейки. В сочетании с памятью и периферийными устройствами коммерческие микросхемы Cortex-M0, например Freescale Kinetis, стоят меньше 50 центов. Поэтому конвейерные процессоры приходят на смену своим более медленным многотактным собратьям даже в большинстве приложений, где дешевизна является важным фактором.

Конвейерная микроархитектура – результат применения принципа конвейерной обработки к однотактной микроархитектуре. Поэтому она позволяет выполнять несколько команд одновременно, значительно улучшая пропускную способность процессора. Конвейерная микроархитектура требует дополнительной логики для разрешения конфликтов между одновременно выполняемыми в конвейере командами. Она также требует несколько неархитектурных регистров, расположенных между стадиями конвейера. Конвейерный процессор должен обращаться к командам и данным на одном и том же такте, для этого обычно используются раздельные кэши команд и данных, о чем пойдет речь в **главе 3** (книга 1). Дополнительная логика и регистры окупаются – в наши дни во всех коммерческих высокопроизводительных процессорах используют конвейеры.

Мы изучим детали и компромиссы этих трех микроархитектур в последующих разделах. В конце главы мы кратко упомянем дополнительные способы увеличения производительности, используемые в современных высокопроизводительных микропроцессорах.

К числу популярных эталонных тестов относятся Dhrystone, CoreMark и SPEC. Первые два – синтетические тесты, включающие куски, встречающиеся во многих типичных программах. Тест Dhrystone был разработан в 1984 и по-прежнему применяется для оценки встраиваемых процессоров, хотя его код не слишком характерен для реальных программ. Тест CoreMark – шаг вперед по сравнению с Dhrystone, в него входит операция умножения матриц, подвергающая испытанию умножитель и сумматор, связные списки для проверки подсистемы памяти, конечные автоматы для проверки логики ветвления и вычисление циклического избыточного кода, в котором участвуют многие компоненты процессора. Оба теста занимают меньше 16 КБ памяти и не нагружают кэш команд.

Тест SPEC CINT2006, разработанный компанией Standard Performance Evaluation Corporation, включает реальные программы, в т. ч. h264ref (сжатие видео), sjeng (программа для игры в шахматы с искусственным интеллектом), hmmer (анализ последовательности белка) и gcc (компилятор языка C). Этот тест широко используется для оценки высокопроизводительных процессоров, поскольку нагружает все компоненты ЦП способами, характерными для настоящих программ.

2.2. Анализ производительности

Как уже отмечалось, у каждой процессорной архитектуры может быть много микроархитектур, обеспечивающих разное соотношение цены и производительности. Цена зависит от количества логических элементов и от технологии производства. Точный расчет цены невозможен без детального знания конкретной технологии производства, но в целом чем больше логических вентилей и памяти, тем больше цена.

В этом разделе мы познакомимся с основами анализа производительности. Производительность компьютерной системы можно измерить разными способами, и маркетологи стараются выбрать те из них, которые позволяют их продукции выглядеть в наилучшем свете вне зависимости от того, имеют эти измерения какое-либо отношение к реальной жизни или нет. Например, производители микропроцессоров часто кладут в основу рекламы тактовую частоту и количество процессорных ядер. Но при этом не акцентируют внимания на том, что одни процессоры могут выполнить за один такт больше работы, чем другие, и что эта характеристика зависит от конкретной программы. И что делать бедному покупателю?

Единственный по-настоящему честный способ узнать производительность компьютера – измерить время выполнения интересующей вас программы. Чем быстрее компьютер выполнит ее, тем выше его производительность. Еще один хороший способ – измерить время выполнения не одной, а нескольких программ, похожих на те, которые вы планируете запускать; это особенно важно, если ваша программа еще не написана или измерения проводит кто-то, у кого ее нет. Такие программы называются *эталонными тестами* (benchmark), а полученные времена выполнения обычно публикуются, чтобы было ясно, насколько быстр компьютер.

Время выполнения программы в секундах можно вычислить по формуле **(2.1)**:

$$\text{Время выполнения} = (\text{число команд})\left(\frac{\text{такты}}{\text{команда}}\right)\left(\frac{\text{секунда}}{\text{такт}}\right). \qquad (2.1)$$

Количество команд в программе зависит от архитектуры процессора. В некоторых архитектурах встречаются сложные команды, каждая из которых выполняет много действий, что уменьшает общее количество команд в программе. Однако такие команды зачастую медленнее выполняются логическими схемами процессора. Количество команд также сильно зависит от смекалки программиста. В этой главе мы будем считать, что выполняются известные программы на процессоре ARM, так что количество команд в каждой программе постоянно и не зависит от микроархитектуры. Количество *тактов на команду*, часто называемое *CPI* (cycles per instruction), – это среднее количество тактов процессора, необходимых для выполнения команды. Этот показатель обратно пропорционален пропускной способности, измеряемой в *командах на такт* (instructions per cycle, IPC). В разных микроархитектурах значения CPI разнятся. В этой главе мы будем считать, что процессор работает с идеальной подсистемой памяти, которая никак не влияет на CPI. В **главе 3** (книга 1) мы рассмотрим случаи, когда процессору иногда приходится ждать ответа от памяти, что увеличивает CPI.

Число секунд на такт называется периодом тактового сигнала T_c. Он зависит от имеющей наибольшую задержку цепи, соединяющей логические элементы внутри процессора (критический путь). В разных микроархитектурах периоды тактового сигнала могут различаться. В частности, на него существенно влияет выбор способов реализации логических и схемных элементов. Например, сумматор с ускоренным переносом работает быстрее, чем сумматор с последовательным переносом. До сих пор благодаря прогрессу в сфере производства удавалось удваивать скорость переключения транзисторов каждые четыре–шесть лет, так что микропроцессор, произведенный сегодня, работает гораздо быстрее, чем его собрат с точно такой же микроархитектурой и логическими элементами, но произведенный десять лет назад.

Главная задача, стоящая перед разработчиком микроархитектуры, – выбрать такую конструкцию, которая обеспечит минимальное время выполнения программ, не выходя в то же время за рамки ограничений по цене и (или) энергопотреблению. Так как решения, принятые на микроархитектурном уровне, влияют и на CPI, и на T_c, и, в свою очередь, зависят от выбранных аппаратных блоков и схемотехнических решений, выбор наилучшего варианта требует очень внимательного анализа.

Существует много других факторов, влияющих на общую производительность компьютера. Например, производительность жестких дисков, памяти, графической или сетевой подсистемы может быть настолько низкой, что производительность процессора на их фоне не будет иметь значения. Даже самый быстрый в мире процессор не поможет вам загружать веб-сайты быстрее, если вы подключены к Интернету через обычную телефонную линию. Такие факторы выходят за рамки этой книги, и рассматривать их мы не будем.

2.3. Однотактный процессор

Сначала мы разработаем микроархитектуру, в которой команды выполняются за один такт. Начнем с конструирования тракта данных путем соединения показанных на **Рис. 2.1** элементов состояния при помощи комбинационной логики, которая и будет выполнять различные команды. Управляющие сигналы определяют, как именно тракт данных должен выполнять команду, находящуюся в нем в текущий момент времени. Устройство управления содержит комбинационную логику, которая формирует необходимые управляющие сигналы в зависимости от того, какая команда выполняется в данный момент. В заключение мы оценим производительность однотактного процессора.

2.3.1. Однотактный тракт данных

В этом разделе мы шаг за шагом создадим однотактный тракт данных, используя элементы состояния, показанные на **Рис. 2.1**. Новые соединения будем выделять черным (или синим, в случае управляющих сигналов) цветом, а уже рассмотренное оборудование будем показывать серым цветом. Регистр состояния является частью устройства управления, поэтому при рассмотрении тракта данных мы его опустим.

Счетчик команд содержит адрес подлежащей выполнению команды. Первым делом нам надо прочитать эту команду из памяти команд. Как показано на **Рис. 2.2**, счетчик команд напрямую подключен к адресному входу памяти команд. 32-битовая команда, прочитанная, или *выбранная*, из памяти команд, отмечена на рисунке как *Instr*.

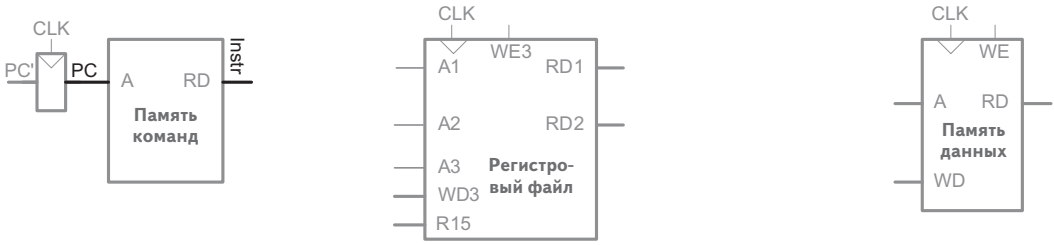


Рис. 2.2. Выборка команды из памяти

Действия процессора зависят от того, какая именно команда была выбрана. Для начала давайте создадим тракт данных для команды `LDR` с положительным непосредственным смещением. А затем подумаем, как обобщить его, чтобы можно было выполнять и другие команды.

Команда LDR

Для команды `LDR` следующим шагом мы должны прочитать регистр-источник, содержащий базовый адрес. Номер этого регистра указан в поле *Rn* команды ($Instr_{19:16}$). Эти биты команды подключены к адресному входу одного из портов регистрового файла (*A1*), как показано на **Рис. 2.3**. Значение, прочитанное из регистрового файла, появляется на выходе *RD1*.

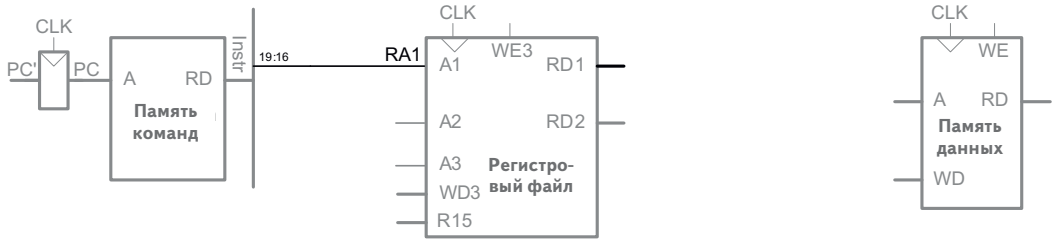


Рис. 2.3. Чтение операнда-источника из регистрового файла

Команде `LDR` также требуется смещение. Смещение хранится в команде как непосредственный операнд в поле $Instr_{11:0}$. Это значение без знака, поэтому его следует дополнить нулями до 32 бит, как показано на **Рис. 2.4.** Полученное 32-битовое значение называется *ExtImm*. Напомним, что под дополнением нулями понимается просто дописывание нулей в начало: $ImmExt_{31:12} = 0$, а $ImmExt_{11:0} = Instr_{11:0}$.

Процессор должен прибавить смещение к базовому адресу, чтобы получить адрес, по которому будет произведено чтение из памяти. Для суммирования мы добавляем в тракт данных АЛУ, как показано на **Рис. 2.5**. АЛУ получает на входы два операнда, *SrcA* и *SrcB*. *SrcA* поступает из регистрового файла, а *SrcB* – из непосредственного смещения, дополненного нулями. АЛУ умеет выполнять много операций, о

чем шла речь в **разделе 5.2.4** (книга 1). Двухбитовый управляющий сигнал *ALUControl* определяет конкретную операцию. На выходе из АЛУ получается 32-битовый результат *ALUResult*. Для команды `LDR` сигнал *ALUControl* должен быть равен 00, чтобы было произведено сложение. Далее *ALUResult* подается на адресный вход памяти данных, как показано на **Рис. 2.5**.

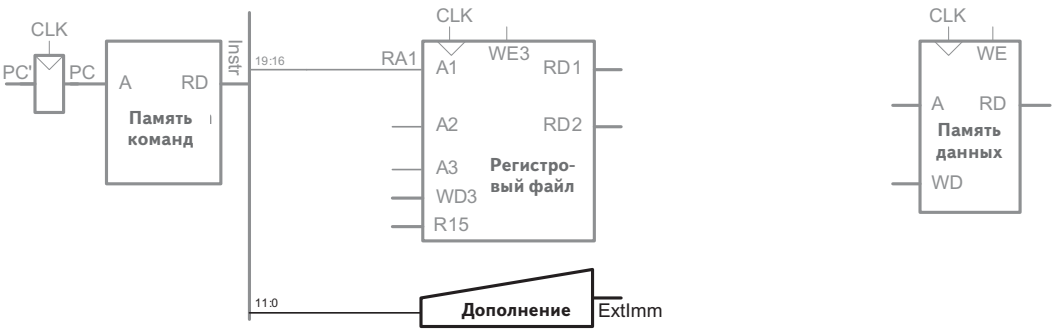


Рис. 2.4. Дополнение непосредственного операнда нулями

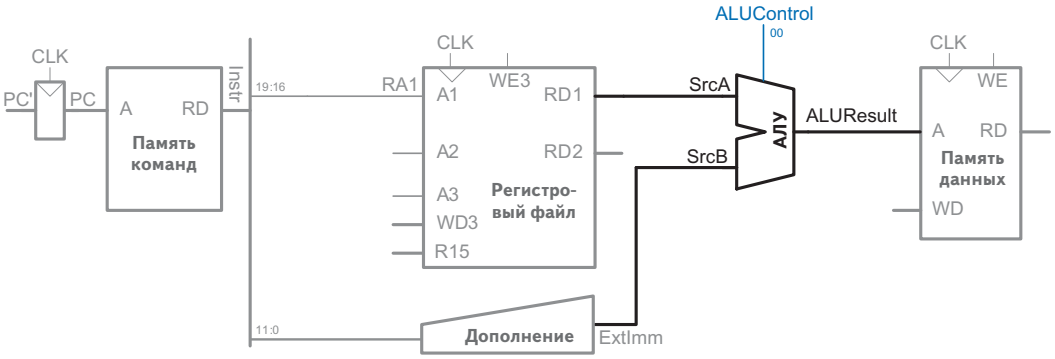


Рис. 2.5. Вычисление адреса в памяти

Значение, прочитанное из памяти данных, попадает на шину *ReadData*, после чего записывается обратно в регистр-приемник в конце такта, как показано на **Рис. 2.6**. Третий порт регистрового файла – это порт записи. *Регистр-приемник* для команды `LDR` задается в поле *Rd* ($Instr_{15:12}$), которое подключено к адресному входу третьего порта (*A3*) регистрового файла. Шина *ReadData* подключена к входу записи данных третьего порта (*WD3*) регистрового файла. Управляющий сигнал *RegWrite* соединен с входом разрешения записи третьего порта (*WE3*) и активен во время выполнения команды `LDR`, так что значение записывается в регистровый файл. Сама запись происходит по положительному фронту сигнала синхронизации, которым заканчивается такт процессора.

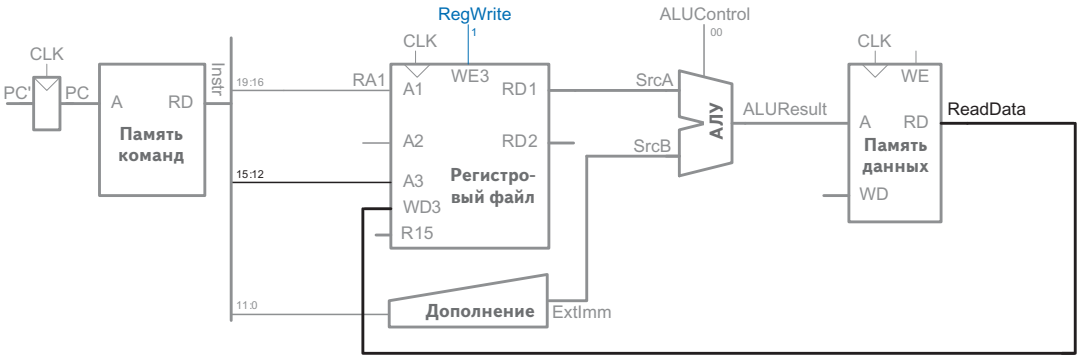


Рис. 2.6. Запись данных в регистровый файл

Одновременно с выполнением команды процессор должен вычислить адрес следующей команды, *PC'*. Так как команды 32-битовые, то есть четырехбайтные, то адрес следующей команды равен *PC* + 4. На **Рис. 2.7** для увеличения PC на 4 используется сумматор. Новый адрес записывается в счетчик команд по следующему положительному фронту сигнала синхронизации. На этом создание тракта данных для команды `LDR` можно было бы считать завершенным, если бы не все портящий случай, когда базовым регистром или регистром-приемником является R15.

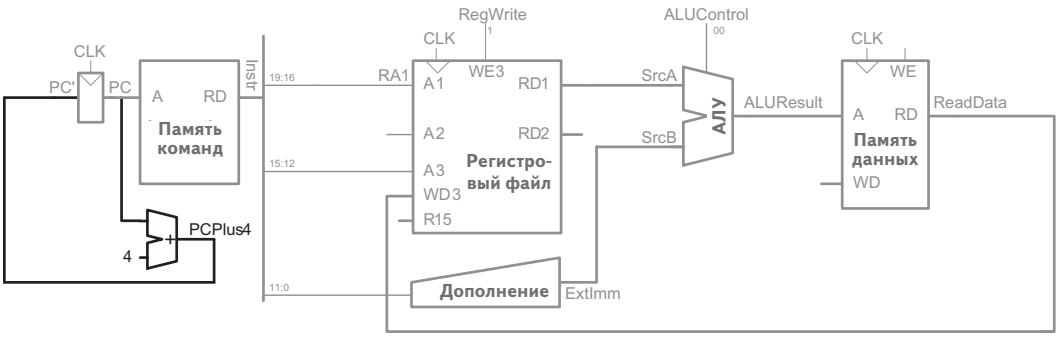


Рис. 2.7. Увеличение счетчика команд

Напомним (см. **раздел 1.4.6**), что в архитектуре ARM чтение из регистра R15 возвращает *PC* + 8. Поэтому нам нужен дополнительный сумматор, чтобы еще раз увеличить *PC* и передать сумму в порт R15 регистрового файла. Аналогично при записи в регистр R15 изменяется *PC*. Следовательно, *PC'* может явиться результатом команды (*ReadData*), а не *PCPlus4*. Одну из этих двух возможностей выбирает мультиплексор. Чтобы выбрать *PCPlus4*, управляющий сигнал *PCSrc* выставляется в 0, а чтобы выбрать *ReadData* – в 1. Особенности, связанные с *PC*, показаны на **Рис. 2.8**.

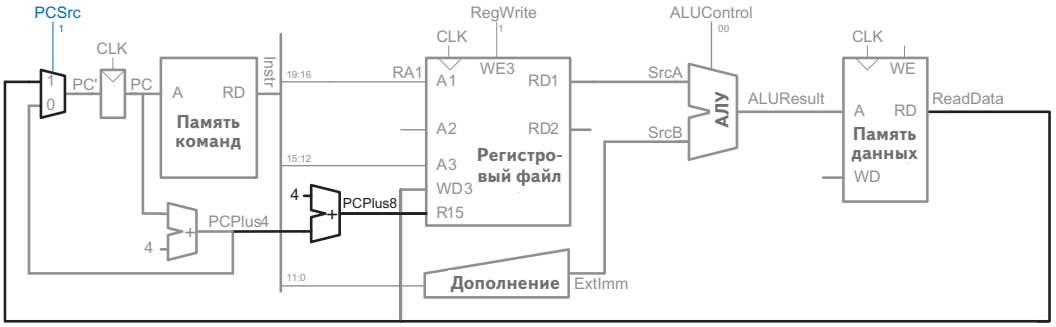


Рис. 2.8. Чтение или запись счетчика команд как регистра R15

Команда STR

Теперь обобщим тракт данных, чтобы он мог выполнять еще и команду `STR`. Как и команда `LDR`, `STR` читает базовый адрес из порта 1 регистрового файла и дополняет нулями смещение, заданное в виде непосредственного операнда. АЛУ складывает базовый адрес со смещением, чтобы получить адрес в памяти. Все эти функции уже реализованы в тракте данных.

Команда `STR` также читает из регистрового файла второй регистр и записывает его содержимое в память данных. На **Рис. 2.9** показаны новые соединения, необходимые для этой функции. Номер регистра задается в поле *Rd* ($Instr_{15:12}$), которое подключено к порту *A2* регистрового файла. Прочитанное значение появляется на выходе *RD2*, который соединен с входом записи в память данных (*WD*). Порт разрешения записи в память данных (*WE*) управляется сигналом *MemWrite*. Для команды `STR` сигнал *MemWrite* = 1, чтобы данные были записаны в память; *ALUControl* = 00, чтобы сложить базовый адрес со смещением; и *RegWrite* = 0, потому что команда ничего не пишет в регистровый файл. Отметим, что данные в любом случае читаются из памяти, но прочитанное значение *ReadData* игнорируется, т. к. *RegWrite* = 0.

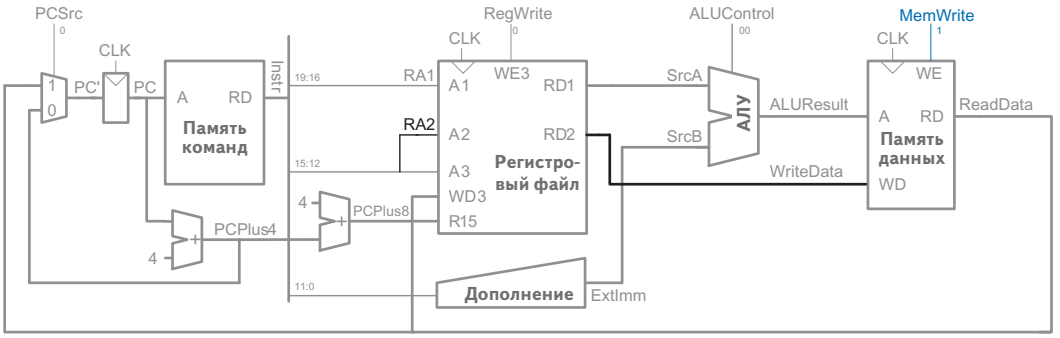


Рис. 2.9. Запись данных в память командой `STR`

Команды обработки данных с непосредственной адресацией

Теперь добавим поддержку команд типа обработки данных: ADD, SUB, AND и ORR – в режиме непосредственной адресации. Все эти команды читают регистр-источник из регистрового файла и непосредственный операнд из младших битов команды, выполняют над ними некую операцию в АЛУ и записывают результат обратно в регистр-приемник. Единственное различие между ними – в типе операции. Таким образом, все они могут быть выполнены одним и тем же оборудованием и будут отличаться только значением управляющего сигнала *ALUControl*. Как было описано в **разделе 5.2.4** (книга 1), *ALUControl* равен 00 для ADD, 01 для SUB, 10 для AND и 11 для ORR. АЛУ также порождает четыре флага $ALUFlags_{3:0}$ (Zero, Negative, Carry, oVerflow), которые передаются обратно устройству управления.

Дополненный тракт данных с поддержкой команд обработки данных показан на **Рис. 2.10**. Как и в случае команды LDR, тракт данных читает первый операнд-источник из порта 1 регистрового файла и дополняет нулями непосредственный операнд, находящийся в младших битах *Instr*. Однако для команд обработки данных используется только 8-битовый, а не 12-битовый непосредственный операнд. Поэтому мы подаем управляющий сигнал *ImmSrc* блоку дополнения. Если этот сигнал равен 0, то *ExtImm* получается расширением нулями поля $Instr_{7:0}$ – для команд обработки данных. А если 1, то *ExtImm* получается расширением нулями поля $Instr_{11:0}$ – для команд LDR и STR.

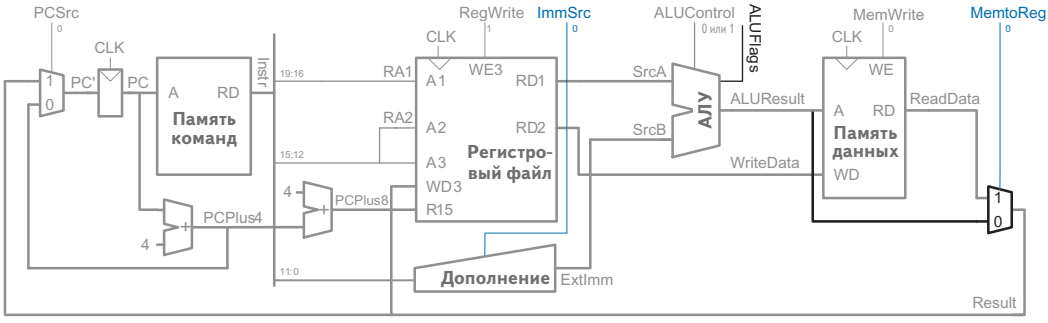


Рис. 2.10. Расширение тракта данных для поддержки команд обработки данных с непосредственной адресацией

Для команды LDR порт записи регистрового файла был всегда подключен к памяти данных. Однако команды обработки данных записывают в регистровый файл значение *ALUResult*. Чтобы выбирать между *ReadData* и *ALUResult*, мы добавляем еще один мультиплексор, выход которого назовем *Result*. Этот мультиплексор управляется еще одним новым сигналом, *MemtoReg*. Сигнал *MemtoReg* равен 0 для команд об-

работки данных – в этом случае *Result* принимает значение *ALUResult*. Для команды `LDR` сигнал *MemtoReg* равен 1, а *Result* принимает значение *ReadData*. Для команды `STR` значение *MemtoReg* не играет никакой роли, так как она ничего в регистровый файл не пишет.

Команды обработки данных с регистровой адресацией

Команды обработки данных с регистровой адресацией получают второй операнд-источник из поля *Rm* ($Instr_{3:0}$), а не из непосредственного операнда. Поэтому мы добавляем на входах в регистровый файл и в АЛУ мультиплексоры, выбирающие второй операнд-источник (см. **Рис 2.11**).

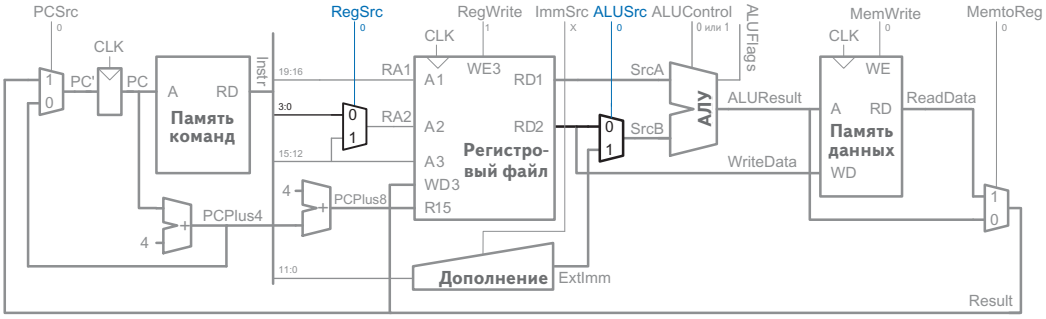


Рис. 2.11. Расширение тракта данных для поддержки команд обработки данных с регистровой адресацией

RA2 берется из поля Rd ($Instr_{15:12}$) для команды `STR` или из поля *Rm* ($Instr_{3:0}$) для команд обработки данных с регистровой адресацией в зависимости от управляющего сигнала *RegSrc*. Аналогично в зависимости от управляющего сигнала *ALUSrc* второй вход в АЛУ выбирается из *ExtImm* для команд с непосредственным операндом или из регистрового файла для команд обработки данных с регистровой адресацией.

Команда B

Наконец, расширим тракт данных для обработки команды `B`, как показано на **Рис. 2.12**. Эта команда безусловного перехода прибавляет 24-битовое непосредственное значение к *PC* + 8 и записывает результат обратно в *PC*. Непосредственный операнд умножается на 4 и расширяется со знаком. Таким образом, в блоке дополнения появляется еще один режим. *ImmSrc* расширяется до 2 бит, которые кодируются, как показано в **Табл. 2.1**.

PC + 8 читается из первого порта регистрового файла. Следовательно, необходим мультиплексор для выбора в качестве входа *RA1*. Этот мультиплексор управляется еще одним битом *RegSrc* и для большинства команд выбирает $Instr_{19:16}$, но для команды `B` – 15.

Для выбора нового значения *PC*, соответствующего переходу, из *ALUResult* сигнал *MemtoReg* устанавливается в 0, а *PCSrc* – в 1.

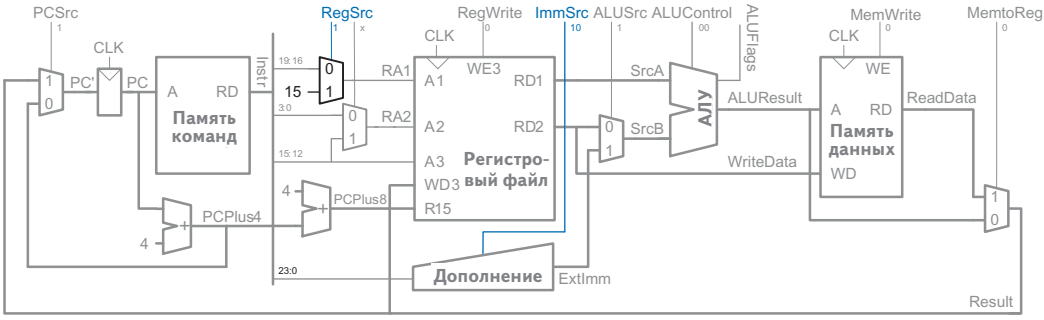


Рис. 2.12. Расширение тракта данных для поддержки команды `B`

Таблица 2.1. Кодирование ImmSrc

ImmSrc	ExtImm	Описание
00	{24 нуля} $Instr_{7:0}$	8-битовый непосредственный операнд без знака для команд обработки данных
01	{20 нулей} $Instr_{11:0}$	12-битовый непосредственный операнд без знака для команд `LDR`/`STR`
10	{6 $Instr_{23}$} $Instr_{23:0}$ 00	24-битовый непосредственный операнд со знаком, умноженный на 4, для команды `B`

На этом разработка тракта данных однотактного процессора завершается. Мы рассмотрели не только устройство процессора, но и сам процесс проектирования, во время которого идентифицировали элементы состояния и соединяли их при помощи все усложняющейся комбинационной логики. В следующем разделе мы рассмотрим, как формировать управляющие сигналы, настраивающие тракт данных на выполнение той или иной команды.

2.3.2. Однотактное устройство управления

Устройство управления формирует управляющие сигналы на основе полей команды *cond*, *op* и *funct* ($Instr_{31:28}$, $Instr_{27:26}$ и $Instr_{25:20}$), а также флагов и в зависимости от того, является ли регистр-приемник счетчиком команд *PC*. Устройство управления хранит также текущие флаги состояния и обновляет их по мере необходимости. На **Рис. 2.13** показан однотактный процессор с устройством управления, подключенным к тракту данных.

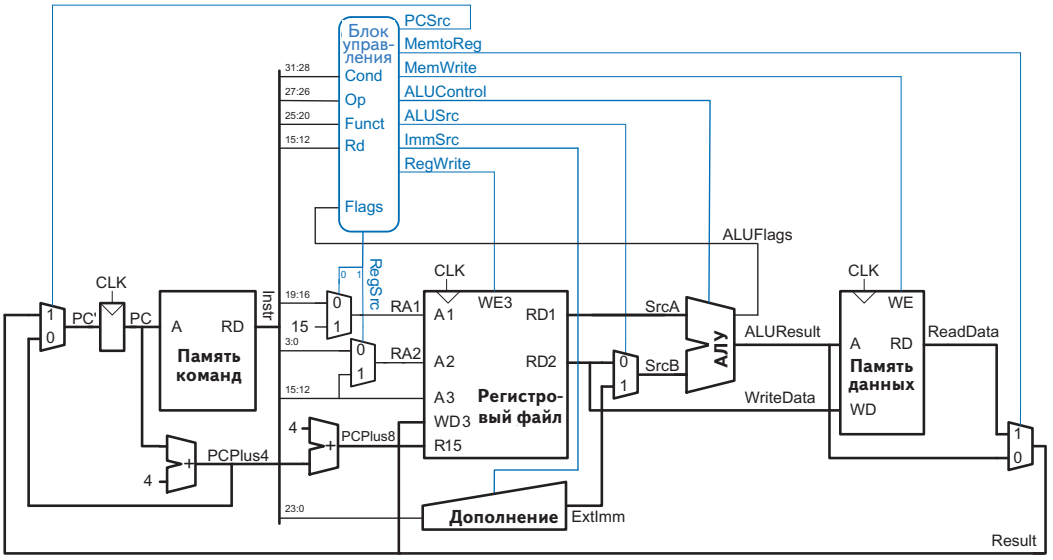


Рис. 2.13. Законченный однотактный процессор

На **Рис. 2.14** показана детальная диаграмма устройства управления. Мы поделили ее на две части: декодер, который формирует управляющие сигналы на основе *Instr*, и блок условной логики, который запоминает флаги состояния и разрешает обновление архитектурного состояния, только когда требуется условное выполнение команды. Декодер, показанный на **Рис. 2.14 (b)**, состоит из главного декодера, который формирует большинство управляющих сигналов, декодера АЛУ, который на основе поля *Funct* определяет тип команды обработки данных, и блока логики PC, который определяет, нужно ли обновить PC вследствие выполнения перехода или записи в R15.

Поведение главного декодера определяется таблицей истинности, показанной в **Табл. 2.2**. Главный декодер определяет тип команды: обработка данных с регистровой адресацией, обработка данных с непосредственной адресацией, `STR`, `LDR` или `B`. Он формирует необходимые управляющие сигналы для тракта данных. Сигналы *MemtoReg*, *ALUSrc*, $ImmSrc_{1:0}$ и $RegSrc_{1:0}$ передаются тракту данных напрямую. Но сигналы разрешения записи *MemW* и *RegW* должны пройти через блок условной логики, прежде чем превратиться в сигналы тракта данных *MemWrite* и *RegWrite*. Блок условной логики может сбросить эти сигналы в 0, если условие не выполнено. Главный декодер также формирует сигналы *Branch* и *ALUOp*, которые используются внутри устройства управления и показывают, что обрабатывается соответственно команда `B` или команда обработки данных. Для разработки главного декодера можно воспользоваться таблицей истинности и тем методом синтеза комбинационных схем, который вам больше нравится.

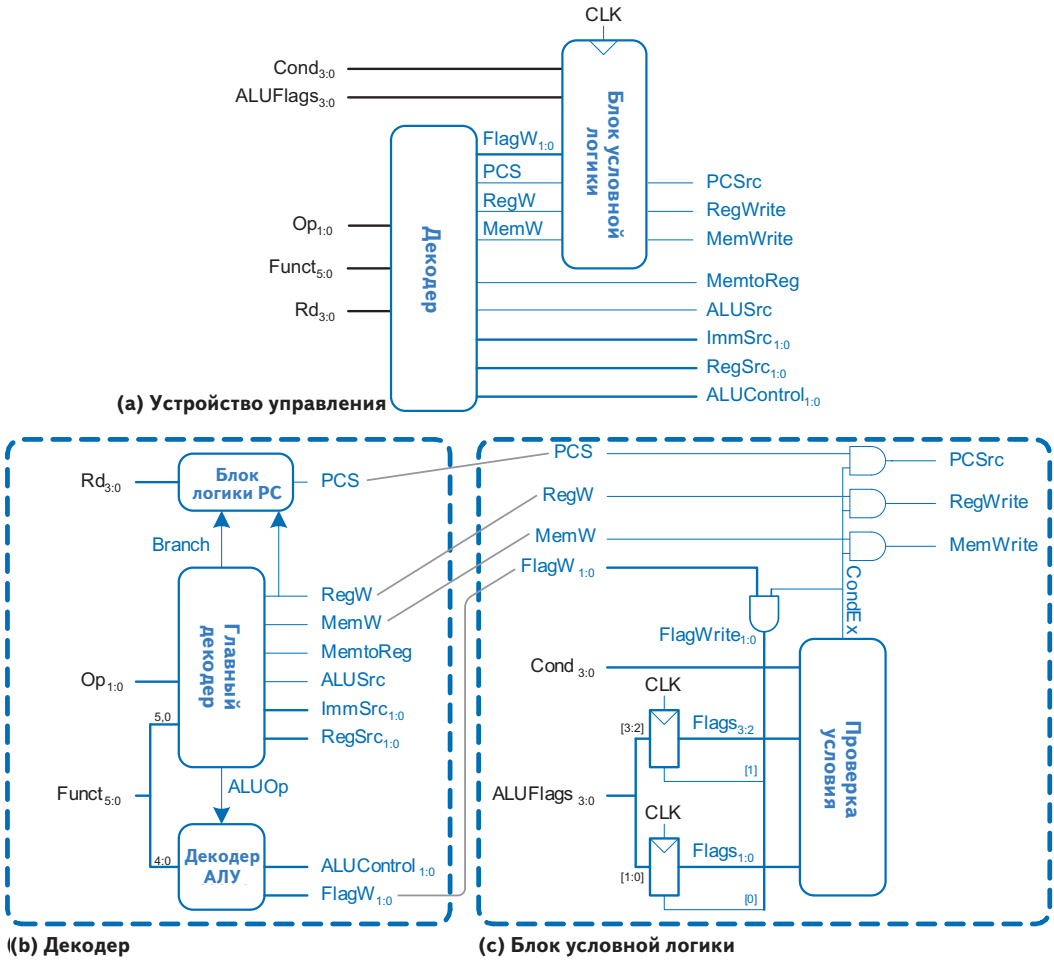


Рис. 2.14. Однотактное устройство управления

Поведение декодера АЛУ описывается таблицами истинности в **Табл. 2.3**. Для команд обработки данных декодер АЛУ формирует сигнал *ALUControl* в зависимости от типа команды (`ADD`, `SUB`, `AND`, `ORR`). Кроме того, если бит *S* поднят, то декодер устанавливает в 1 сигнал *FlagW*, чтобы обновить флаги состояния. Отметим, что команды `ADD` и `SUB` обновляют все флаги, а команды `AND` и `ORR` – только флаги *N* и *Z*, поэтому сигнал *FlagW* должен быть двухбитовым: бит $FlagW_1$ индицирует обновление *N* и *Z* ($Flags_{3:2}$), а бит $FlagW_0$ – обновление *C* и *V* ($Flags_{1:0}$). Сигнал $FlagW_{1:0}$ сбрасывается в 0 блоком условной логики, если условие не выполнено ($CondEx = 0$).

Таблица 2.2. Таблица истинности главного декодера

Op	$Funct_5$	$Funct_0$	Type	Branch	MemtoReg	MemW	ALUSrc	ImmSrc	RegW	RegSrc	ALUOp
00	0	X	DP Reg	0	0	0	0	XX	1	00	1
00	1	X	DP Imm	0	0	0	1	00	1	X0	1
01	X	0	STR	0	X	1	1	01	0	10	0
01	X	1	LDR	0	1	0	1	01	1	X0	0
10	X	X	B	1	0	0	1	10	0	X1	0

Таблица 2.3. Таблица истинности декодера АЛУ

ALUOp	$Funct_{4:1}$ (cmd)	$Funct_0$ (S)	Тип	$ALUControl_{1:0}$	$FlagW_{1:0}$
0	X	X	Не ОД	00 (Add)	00
1	0100	0	ADD	00 (Add)	00
		1			11
	0010	0	SUB	01 (Sub)	00
		1			11
	0000	0	AND	10 (And)	00
		1			10
	1100	0	ORR	11 (Or)	00
		1			10

Блок логики PC проверяет, является ли команда записью в регистр R15 или переходом, поскольку в этих случаях следует обновить счетчик команд *PC*. Логика такова:

$$PCS = ((Rd == 15) \,\&\, RegW) \mid Branch.$$

Сигнал *PCS* может быть сброшен блоком условной логики до передачи его тракту данных в виде *PCSrc*.

Блок условной логики, показанный на **Рис. 2.14 (c)**, определяет, следует ли выполнять команду (*CondEx*), анализируя поле *cond* и текущие значения флагов *N*, *Z*, *C* и *V* ($Flags_{3:0}$). Логика принятия решения описана в **Табл. 1.3**. Если команду выполнять не следует, то сигналы разрешения записи и *PCSrc* сбрасываются в 0, так что команда не изменяет архитектурного состояния. Блок условной логики также обновляет некоторые или все флаги в *ALUFlags*, если сигнал *FlagW* установлен в 1 декодером АЛУ и условие команды выполнено (*CondEx* = 1).

Пример 2.1. ФУНКЦИОНИРОВАНИЕ ОДНОТАКТНОГО ПРОЦЕССОРА

Определите значения управляющих сигналов, а также части тракта данных, которые задействованы при выполнении команды ORR с регистровым режимом адресации.

Решение. На **Рис. 2.15** показаны управляющие сигналы и поток данных во время выполнения команды ORR. Счетчик команд указывает на ячейку памяти, из которой выбирается команда.

Главный поток данных через регистровый файл и АЛУ показан жирной синей линией. Из регистрового файла читаются два операнда-источника, определяемые полями $Instr_{19:16}$ и $Instr_{3:0}$, поэтому сигнал *RegSrc* должен быть равен 00. На вход *SrcB* нужно подать значение, прочитанное из второго порта регистрового файла (а не *ExtImm*), так что *ALUSrc* должен быть равен нулю. АЛУ выполняет поразрядную операцию OR, поэтому значение сигнала *ALUControl* должно быть равно 11. Результат формируется в АЛУ, поэтому *MemtoReg* должен быть равен 0. Результат записывается в регистровый файл, поэтому *RegWrite* будет равен 1. Команда ничего не пишет в память, так что *MemWrite* = 0.

Запись нового значения *PCPlus4* в счетчик команд показана жирной серой линией. Сигнал *PCSrc* равен 0, что означает простой инкремент счетчика команд.

Важно иметь в виду, что по цепям, не отмеченным пунктиром, тоже передаются какие-то данные, однако для этой конкретной команды совершенно не имеет значения, что они из себя представляют. Например, происходит дополнение нулями непосредственного операнда, а данные читаются из памяти, но это не оказывает никакого влияния на следующее состояние системы.

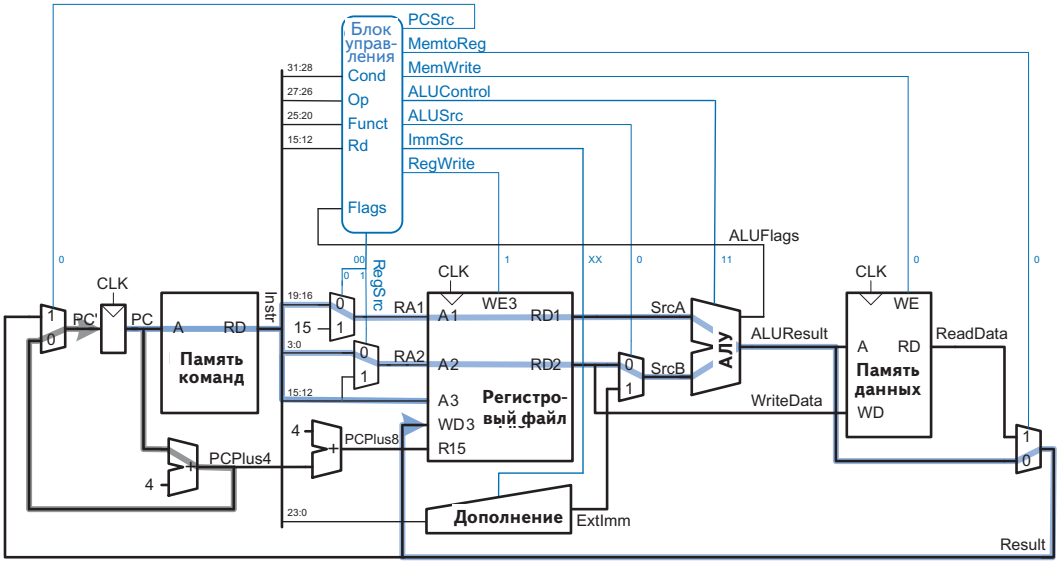


Рис. 2.15. Управляющие сигналы и поток данных при выполнении команды ORR

2.3.3. Дополнительные команды

Мы рассмотрели лишь малое подмножество полной системы команд ARM. В этом разделе мы добавим поддержку команды сравнения CMP и режимов адресации, в которых второй операнд-источник является сдвинутым регистром. Это позволит проиллюстрировать процесс добавления новых команд. Приложив некоторые усилия и действуя по аналогии, вы сможете расширить однотактный процессор, так что он будет поддерживать все команды ARM. Кроме того, мы увидим, что поддержка некоторых новых команд зачастую заключается всего лишь в усложнении декодеров, тогда как для других команд могут понадобиться дополнительные аппаратные блоки в тракте данных.

Пример 2.2. КОМАНДА CMP

Команда сравнения CMP вычитает *SrcB* из *SrcA* и устанавливает флаги, но не записывает разность в регистр. В тракте данных уже есть вся необходимая функциональность для выполнения этой команды. Определите, какие изменения необходимо внести в устройство управления, чтобы поддержать команду CMP.

Решение. Следует ввести новый управляющий сигнал *NoWrite*, который предотвращает запись в *Rd* при выполнении команды CMP. (Этот сигнал был бы полезен и для других команд, например TST, которые не осуществляют запись в регистр.) Мы расширим декодер АЛУ, добавив формирование этого сигнала, а также логику формирования сигнала *RegWrite* с учетом его наличия – см. **Рис. 2.16**. Дополненная таблица истинности декодера АЛУ приведена в **Табл. 2.4**, новая команда и сигнал выделены цветом.

Таблица 2.4. Таблица истинности декодера АЛУ, дополненная командой CMP

ALUOp	$Funct_{4:1}$ (cmd)	$Funct_0$ (S)	Примечания	$ALUControl_{1:0}$	$FlagW_{1:0}$	NoWrite
0	X	X	Не ОД	00	00	0
1	0100	0	ADD	00	00	0
		1		11		0
	0010	0	SUB	01	00	0
		1		11		0
	0000	0	AND	10	00	0
		1		10		0
	1100	0	ORR	11	00	0
		1		10		0
	1010	1	CMP	01	11	1

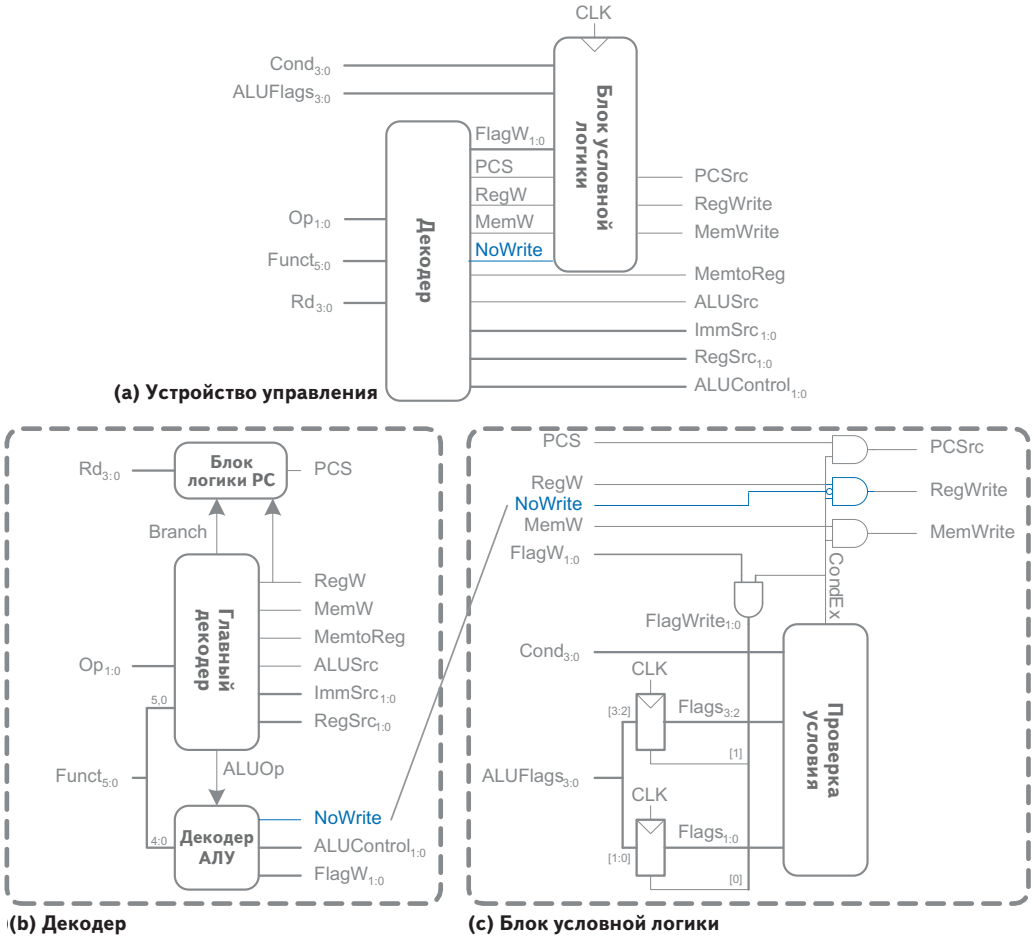


Рис. 2.16. Модификация устройства управления для поддержки команды CMP

Пример 2.3. ДОПОЛНИТЕЛЬНЫЙ РЕЖИМ АДРЕСАЦИИ: РЕГИСТРЫ С КОНСТАНТНЫМИ СДВИГАМИ

До сих пор мы предполагали, что в командах обработки данных с регистровой адресацией второй регистр-источник не сдвигается. Модифицируйте однотактный процессор, добавив поддержку сдвига на непосредственно заданную величину.

Решение. Следует вставить сдвиговый регистр перед АЛУ. На **Рис. 2.17** показан модифицированный тракт данных. Сдвиговый регистр использует поле $Instr_{11:7}$ для определения величины сдвига и поле $Instr_{6:5}$ для определения его типа.

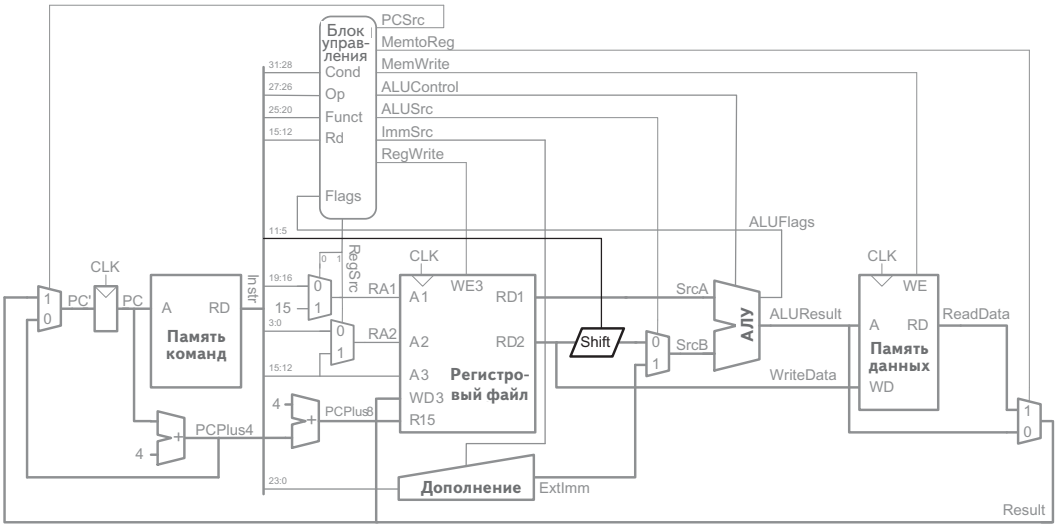


Рис. 2.17. Тракт данных, дополненный регистровой адресацией с константным сдвигом

2.3.4. Анализ производительности

Каждая команда в однотактном процессоре выполняется ровно за один такт, так что CPI равно 1. Критический путь для команды `LDR` показан на **Рис. 2.18** жирной синей линией. Путь начинается там, где в счетчик команд по положительному фронту сигнала синхронизации записывается новый адрес. Затем обновленное значение PC используется для выборки следующей команды. Главный декодер формирует сигнал $RegSrc_0$, который понуждает мультиплексор выбрать $Instr_{19:16}$ в качестве *RA1*, и этот регистр читается из регистрового файла как операнд *SrcA*. При чтении из регистрового файла непосредственно заданное поле дополняется нулями, и мультиплексор *ALUSrc* выбирает его в качестве *SrcB*. АЛУ складывает *SrcA* и *SrcB*, в результате чего определяется конечный адрес. По этому адресу производится чтение из памяти данных. Мультиплексор *MemtoReg* выбирает *ReadData*. Наконец, сигнал *Result* должен стать стабильным на входе регистрового файла, до того как придет следующий положительный фронт сигнала синхронизации, иначе будет записано неверное значение. Таким образом, длительность одного такта равна:

$$T_{c1} = t_{pcq\_PC} + t_{mem} + t_{dec} + max[t_{mux} + t_{RFread}, t_{ext} + t_{mux}] + t_{ALU} + t_{mem} + t_{mux} + t_{RFsetup}. \quad (2.2)$$

Нижний индекс 1 служит для того, чтобы отличить эту длительность такта от значений в последующих вариантах процессора. В большинстве технологий производства микросхем доступ к АЛУ, памяти и регистро-

вым файлам занимает гораздо больше времени, чем прочие операции. Таким образом, мы можем приближенно посчитать длительность одного такта как:

$$T_{c1} = t_{pcq\_PC} + 2t_{mem} + t_{dec} + t_{RFread} + t_{ALU} + 2t_{mux} + t_{RFsetup}. \qquad (2.3)$$

Численные значения участвующих в формуле длительностей зависят от конкретной технологии.

У других команд критические пути короче. Например, командам обработки данных не нужно обращаться к памяти данных. Тем не менее раз уж мы проектируем синхронные последовательные схемы, период сигнала синхронизации постоянный и всегда должен определяться самой медленной командой.

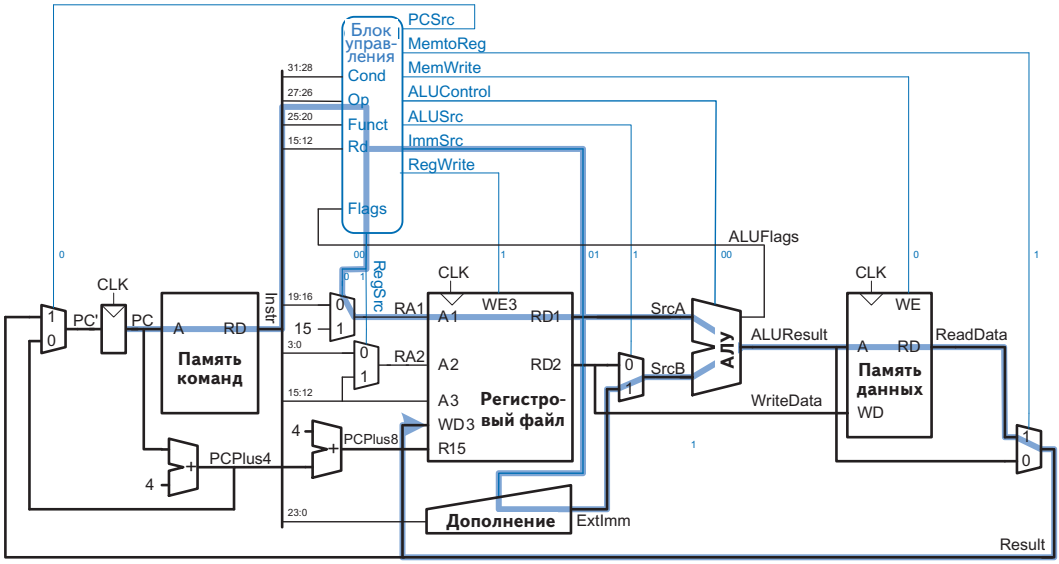


Рис. 2.18. Критический путь для команды LDR

Пример 2.4. ПРОИЗВОДИТЕЛЬНОСТЬ ОДНОТАКТНОГО ПРОЦЕССОРА

Бен Битдидл подумывает построить однотактный процессор по 16-нм КМОП-техпроцессу. Он выяснил, что задержки логических элементов такие, как в **Табл. 2.5**. Помогите ему вычислить время выполнения программы, состоящей из 100 миллиардов команд.

Решение: Согласно формуле (2.3), длительность такта однотактного процессора равна T_{c1} = 40 + 2(200) + 70 + 100 + 120 + 2(25) + 60 = 840 пс. Согласно формуле (2.1), общее время выполнения программы составит T_1 = (100 × 10^9 команд) (1 такт / команду) (840 × 10^{-12} с / такт) = 84 с.

Таблица 2.5. Задержки элементов

Элемент	Параметр	Задержка (пс)
Задержка распространения clk-to-Q в регистре	t_{pcq}	40
Время предустановки регистра	t_{setup}	50
Мультиплексор	t_{mux}	25
АЛУ	t_{ALU}	120
Декодер	t_{dec}	70
Чтение из памяти	t_{mem}	200
Чтение из регистрового файла	t_{RFread}	100
Время предустановки регистрового файла	$t_{RFsetup}$	60

2.4. Многотактный процессор

У однотактного процессора три основные проблемы. Во-первых, ему требуется отдельная память команд и данных, тогда как в большинстве процессоров имеется только одна общая память для команд и данных. Во-вторых, период его тактового сигнала должен быть достаточно большим, чтобы успела выполниться самая медленная команда (`LDR`), несмотря на то что большинство команд выполняется гораздо быстрее. Наконец, ему нужно три сумматора (один для АЛУ и два для вычисления нового значения счетчика команд); сумматоры, особенно быстрые, стоят относительно дорого.

Один из способов решить эти проблемы – использовать многотактный процессор, в котором выполнение каждой команды разбивается на несколько этапов. На каждом этапе процессор может читать или писать данные в память или регистровый файл, или выполнять какую-нибудь операцию в АЛУ. Команда читается на одном этапе, а данные можно прочитать или записать на одном из последующих этапов, поэтому процессор сможет обойтись общей памятью для команд и данных. У разных команд в этом случае будет разное количество этапов, так что простые команды смогут выполняться быстрее, чем сложные. Понадобится только один сумматор; на разных этапах он может использоваться для разных целей.

Мы спроектируем многотактный процессор тем же способом, что и однотактный. Сначала сконструируем тракт данных, соединяя при помощи комбинационной логики блоки памяти и блоки, хранящие архитектурное состояние процессора. Но на этот раз мы добавим и другие блоки неархитектурного состояния для хранения промежуточных результатов между этапами. После этого займемся устройством управления. Так как теперь оно должно формировать разные управляющие сигналы в зави-

симости от текущего этапа выполнения команды, то вместо комбинационных схем нам понадобится конечный автомат. Напоследок мы снова оценим производительность и сравним ее с производительностью однотактного процессора.

2.4.1. Многотактный тракт данных

Как и прежде, в основу проекта положим показанные на **Рис. 2.19** память и архитектурное состояние процессора. В однотактном процессоре мы использовали раздельную память команд и данных, потому что нужно было за один такт и читать из памяти команд, и обращаться к памяти данных. Теперь будем использовать общую память, в которой хранятся и команды, и данные. Это более реалистичный сценарий, и теперь он возможен благодаря тому, что мы можем выбирать команду на одном такте, а читать или записывать данные на другом. Счетчик команд и регистровый файл при этом не изменяются. Как и в случае однотактного процессора, мы шаг за шагом будем строить тракт данных, добавляя компоненты, необходимые для каждого из этапов выполнения команды.

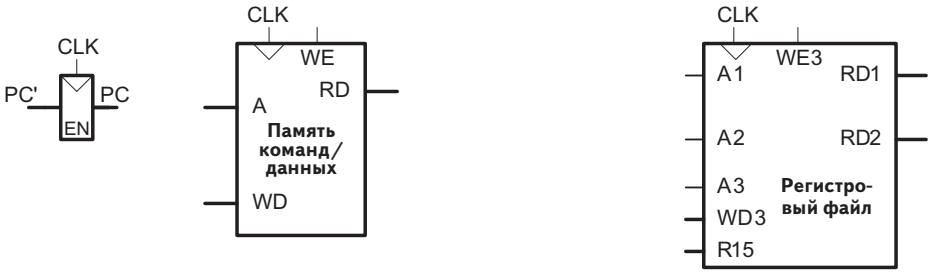


Рис. 2.19. Элементы состояния, включая объединенную память для команд и данных

Счетчик команд содержит адрес команды, которая должна быть выполнена следующей. Соответственно, первым делом следует прочитать ее из памяти команд. Как показано на **Рис. 2.20**, счетчик команд напрямую подсоединен к адресному входу памяти. Прочитанная из памяти команда сохраняется в новом неархитектурном регистре команд (Instruction Register, IR), чтобы ее можно было использовать на следующих тактах. Сигнал разрешения записи в регистр команд назовем *IRWrite* и будем использовать, когда потребуется записать в этот регистр новую команду.

Команда LDR

Как и в случае однотактного процессора, начнем разработку тракта данных с команды `LDR`. После выборки `LDR` из памяти следующим шагом будет чтение регистра-источника, содержащего базовый адрес. Номер регистра указывается в поле *Rn* ($Instr_{19:16}$) и подается на адресный вход

A1 регистрового файла, как показано на **Рис. 2.21**. Значение, прочитанное из регистрового файла, появляется на выходе *RD1*, после чего сохраняется еще в одном неархитектурном регистре *A*.

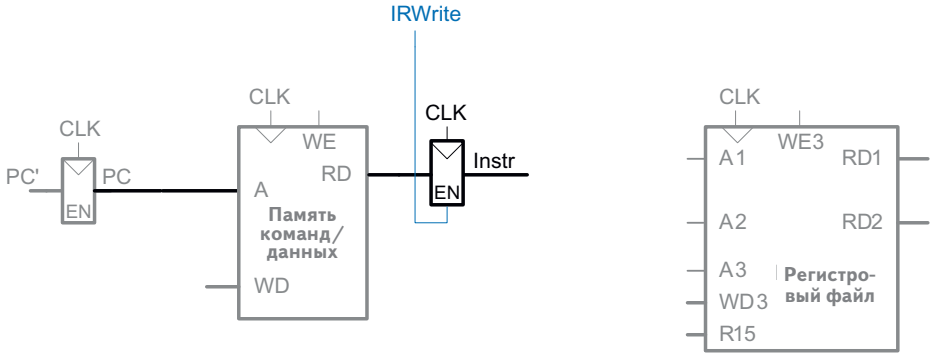


Рис. 2.20. Выборка команды из памяти

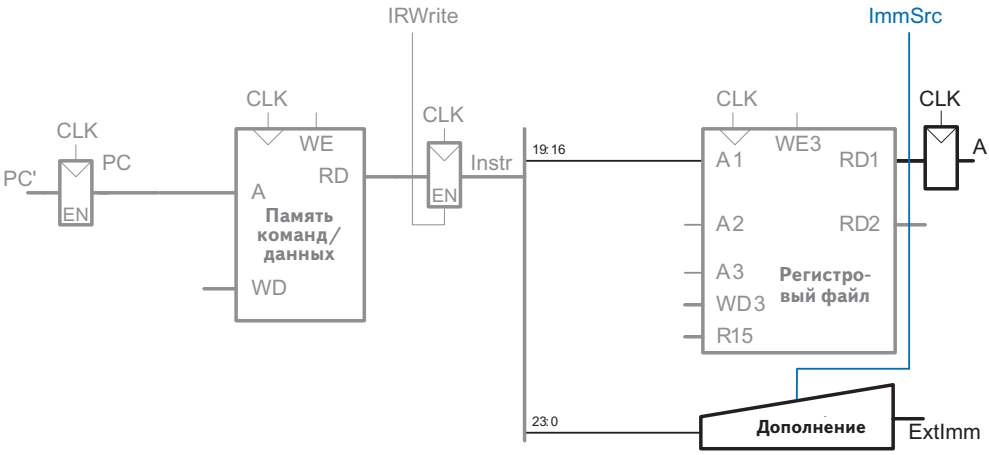


Рис. 2.21. Чтение одного операнда из регистрового файла и дополнение нулями непосредственного поля второго операнда

Команде `LDR` также нужно 12-битовое смещение, которое представляет собой непосредственный операнд, находящийся в поле команды $Instr_{19:16}$. Над этим операндом выполняется операция дополнения нулями до 32 бит, как показано на **Рис. 2.21**. Как и в случае однотактного процессора, на вход блока дополнения подается управляющий сигнал *ImmSrc*, который определяет, какое число дополнять нулями: 8-, 12- или 24-битовое. Дополненный до 32 бит непосредственный операнд обозначается *ExtImm*. Для единообразия мы могли бы сохранить *ExtImm* еще в одном неархитектурном регистре, но так как *ExtImm* – комбинационная функция от *Instr* и не изменяется на всем протяжении обработки теку-

щей команды, то нет смысла заводить специальный регистр для хранения константного значения.

Адрес, по которому следует читать из памяти, получается путем сложения базового адреса и смещения. Для сложения мы используем АЛУ, как показано на **Рис. 2.22**. Чтобы АЛУ выполнило сложение, управляющий сигнал *ALUControl* должен быть равен 00. *ALUResult* сохраняется в неархитектурном регистре *ALUOut*.

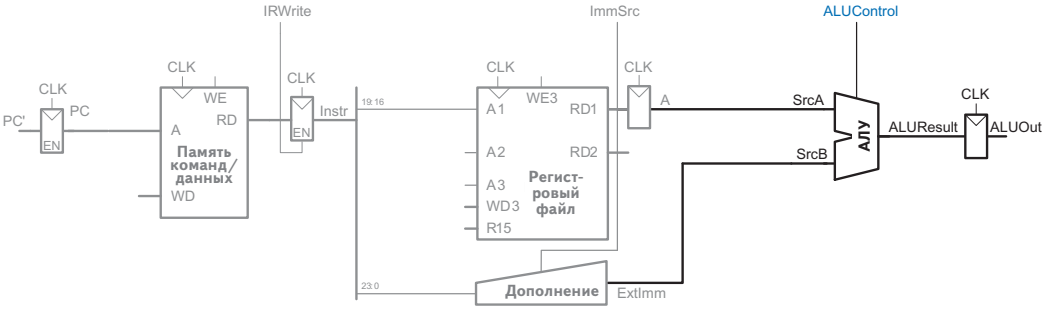


Рис. 2.22. Сложение базового адреса со смещением

Следующим шагом мы должны прочитать данные из ячейки памяти с только что вычисленным адресом. Для этого перед адресным входом памяти необходимо поставить мультиплексор, который в качестве адреса *Adr* будет выбирать либо *PC*, либо *ALUOut* в зависимости от сигнала *AdrSrc*, как показано на **Рис. 2.23**. Прочитанные из памяти данные сохраняются в неархитектурном регистре *Data*. Отметим, что мультиплексор адреса позволяет использовать память во время выполнения команды `LDR`. На первом этапе в качестве адреса используется *PC*, что позволяет выбрать команду. А на последующем этапе мы в качестве адреса используем *ALUOut* и читаем данные. Таким образом, управляющий сигнал *AdrSrc* должен принимать разные значения на разных этапах выполнения команды. В **разделе 2.4.2** мы создадим конечный автомат, который будет формировать требуемую последовательность управляющих сигналов.

Наконец, данные должны быть записаны в регистровый файл, как показано на **Рис. 2.24**. Номер регистра-приемника определяется полем *Rd* ($Instr_{15:12}$). Результат поступает из регистра *Data*. Вместо того чтобы подключать регистр *Data* напрямую к порту записи *WD3* регистрового файла, добавим на шину *Result* мультиплексор, который будет выбирать либо *ALUOut*, либо *Data*, перед тем как подавать *Result* обратно в порт записи регистрового файла. Это полезно, поскольку другим командам нужно будет записывать в регистровый файл результат, вычисленный АЛУ. Сигнал *RegWrite* равен 1, если необходимо обновить регистровый файл.

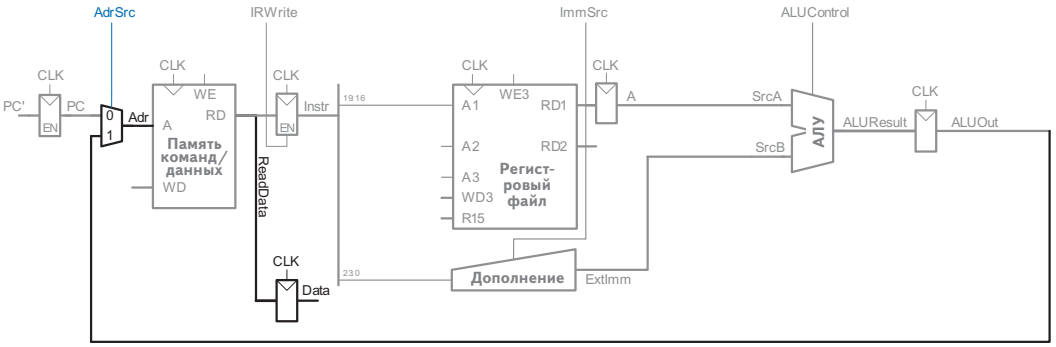


Рис. 2.23. Чтение данных из памяти

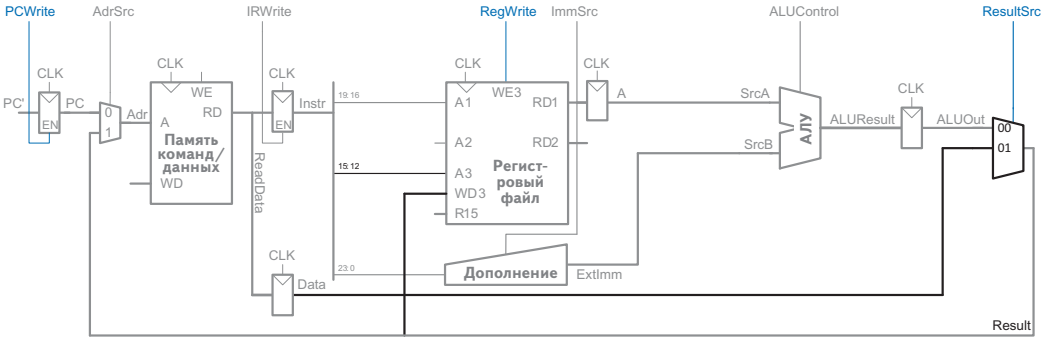


Рис. 2.24. Запись данных обратно в регистровый файл

За то время, пока выполняются все вышеперечисленные операции, процессор должен увеличить счетчик команд на четыре. В однотактном процессоре для этого был нужен отдельный сумматор. В многотактном процессоре мы можем использовать уже имеющееся АЛУ на этапе выборки, поскольку оно еще не занято. Для этого понадобится добавить пару мультиплексоров, которые позволят подавать на входы АЛУ содержимое счетчика команд *PC* и константу 4, как показано на **Рис. 2.25**. Мультиплексор, управляемый сигналом *ALUSrcA*, подает на *SrcA* либо *PC*, либо регистр *A*. Второй мультиплексор подает на *SrcB* либо константу 4, либо *ExtImm*. Для того чтобы обновить *PC*, АЛУ складывает *SrcA* (*PC*) и *SrcB* (4) и записывает полученный результат в счетчик команд. Мультиплексор *ResultSrc* выбирает эту сумму из *ALUResult*, а не *ALUOut*; для этого понадобится третий вход. Управляющий сигнал *PCWrite* разрешает запись в счетчик команд только на определенных тактах.

И снова на нашем пути встает особенность архитектуры ARM – при чтении регистра R15 возвращается *PC* + 8, а при записи в R15 обновляется *PC*. Сначала рассмотрим чтение R15. На этапе выборки мы уже вычислили *PC* + 4, и эта сумма находится в регистре *PC*. Поэтому на втором этапе мы можем получить *PC* + 8, использовав АЛУ, чтобы прибавить 4

к обновленному счетчику команд. *ALUResult* выбирается в качестве *Result* и подается на входной порт R15 регистрового файла. На **Рис. 2.26** показан тракт данных для команды *LDR* с этой новой цепью. Таким образом, чтение из R15, которое также производится на втором этапе, формирует значение *PC* + 8 на выходе чтения данных регистрового файла. При записи в R15 необходимо записывать не в регистровый файл, а в счетчик команд. Следовательно, на последнем этапе выполнения команды *Result* следует перенаправить в регистр *PC* (вместо регистрового файла) и установить в 1 сигнал *PCWrite* (а не *RegWrite*). В тракте данных это уже учтено, поэтому никакие изменения не требуются.

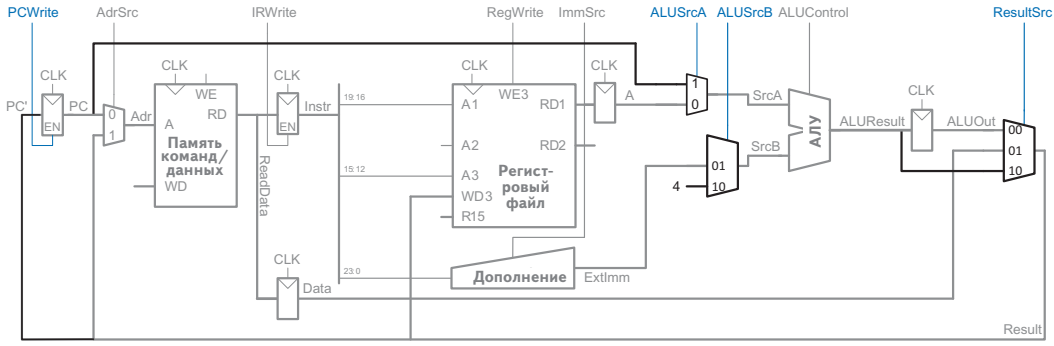


Рис. 2.25. Увеличение PC на 4

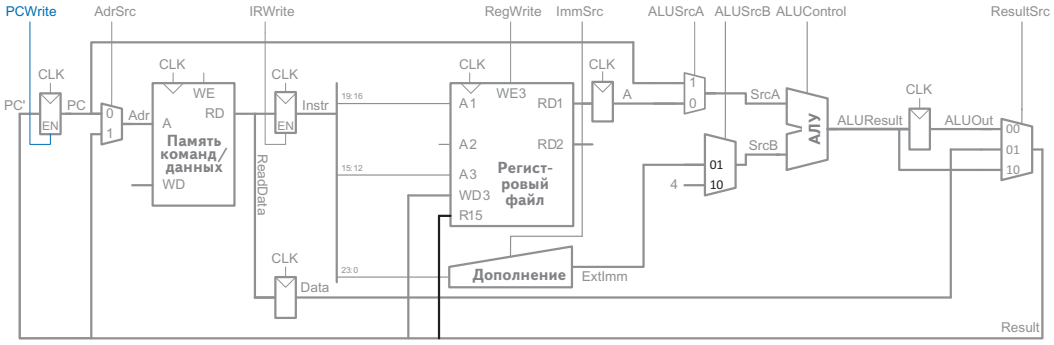


Рис. 2.26. Обработка чтения и записи в R15

Команда STR

Теперь добавим в тракт данных поддержку команды `STR`. Как и `LDR`, команда `STR` читает базовый адрес из первого порта регистрового файла и выполняет дополнение нулями непосредственного операнда, после чего АЛУ складывает их, получая адрес для записи в память. Все эти функции уже есть в тракте данных.

Единственное отличие STR состоит в том, что мы должны прочитать второй регистр из регистрового файла и записать его содержимое в память, как показано на **Рис. 2.27**. Номер регистра указан в поле *Rd* ($Instr_{15:12}$), которое подключено ко второму порту регистрового файла. Прочитанное значение сохраняется в неархитектурном регистре *WriteData*. На следующем этапе оно подается в порт записи данных (*WD*) памяти. Управляющий сигнал *MemWrite* показывает, когда именно данные должны быть записаны в память.

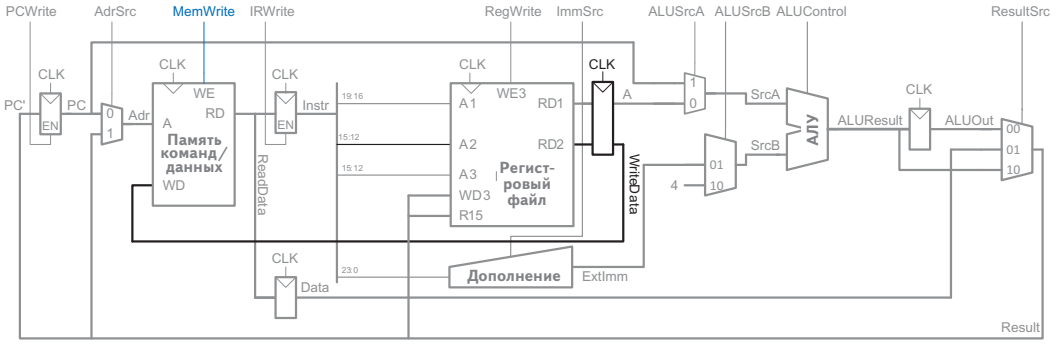


Рис. 2.27. Тракт данных, доработанный для поддержки команды STR

Команды обработки данных с непосредственной адресацией

Команды обработки данных с непосредственной адресацией сначала читают первый операнд из *Rn* и дополняют нулями 8-битовое непосредственное поле второго операнда. Над этими двумя величинами выполняется операция, и ее результат записывается в регистровый файл. Тракт данных уже содержит все необходимые для этих этапов цепи. АЛУ использует управляющий сигнал *ALUControl*, чтобы определить тип подлежащей выполнению команды обработки данных. Флаги *ALUFlags* передаются обратно устройству управления для обновления регистра состояния.

Команды обработки данных с регистровой адресацией

Команды обработки данных с регистровой адресацией берут второй операнд из регистрового файла. Номер регистра задается в поле *Rm* ($Instr_{3:0}$), поэтому мы вставляем мультиплексор, который выбирает это поле в качестве порта *RA2* регистрового файла. Мы также добавляем в мультиплексор *SrcB* еще один вход, на который подается значение, прочитанное из регистрового файла, как показано на **Рис. 2.28**. Во всех

остальных отношениях поведение такое же, как у команд обработки данных с непосредственной адресацией.

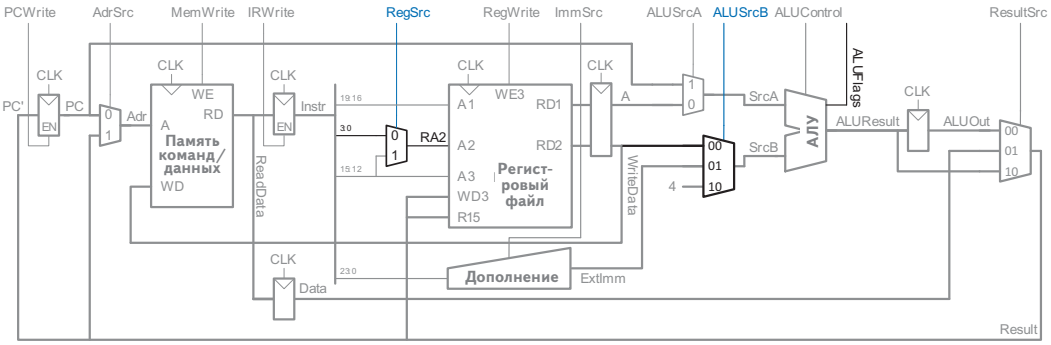


Рис. 2.28. Тракт данных, доработанный для поддержки команд обработки данных с регистровой адресацией

Команда B

Команда перехода `B` читает $PC + 8$ и 24-битовый непосредственный операнд, вычисляет их сумму и прибавляет результат к PC. Напомним (см. **раздел 1.4.6**), что чтение из R15 возвращает $PC + 8$, поэтому мы добавляем мультиплексор, который выбирает R15 в качестве порта *RA1* регистрового файла, как показано на **Рис. 2.29**. Все остальные элементы, необходимые для сложения и записи в PC, уже присутствуют в тракте данных.

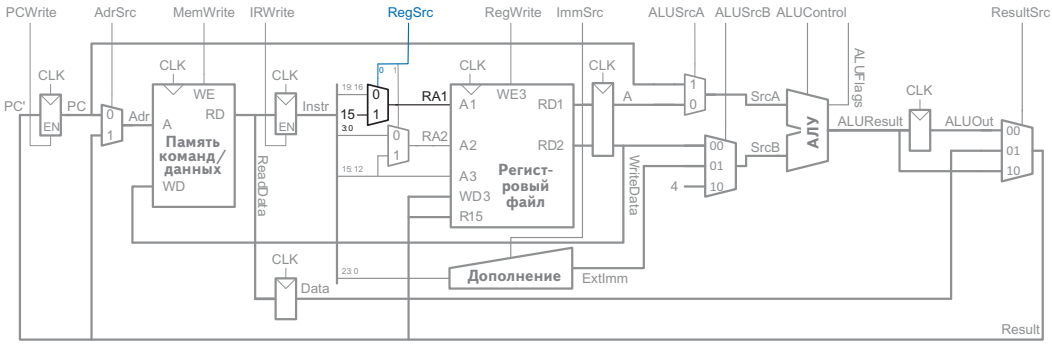


Рис. 2.29. Тракт данных, доработанный для поддержки команды `B`

На этом проектирование многотактного тракта данных завершается. Процесс проектирования мало чем отличался от примененного для однотактного процессора, когда мы методически добавляли блок за блоком между элементами, хранящими состояние процессора. Главное отличие заключается в том, что каждая команда выполняется в несколько этапов. Нам потребовались неархитектурные регистры, чтобы сохранять результаты каждого этапа. За счет этого удалось поместить команды и данные

в общую память, а также повторно использовать одно и то же АЛУ, что позволило избавиться от нескольких сумматоров. В следующем разделе мы разработаем конечный автомат, который будет формировать управляющие сигналы для каждого этапа в нужной последовательности.

2.4.2. Многотактное устройство управления

Как и в однотактном процессоре, устройство управления формирует управляющие сигналы в зависимости от полей *cond*, *op* и *funct* команды ($Instr_{31:28}$, $Instr_{27:26}$ и $Instr_{25:20}$ соответственно), а также флагов и того, является ли регистр-приемник счетчиком команд *PC*. Устройство управления также хранит текущие флаги состояния и по мере необходимости обновляет их. На **Рис. 2.30** показан многотактный процессор с устройством управления, подключенным к тракту данных. Тракт данных показан черным цветом, а устройство управления – синим.

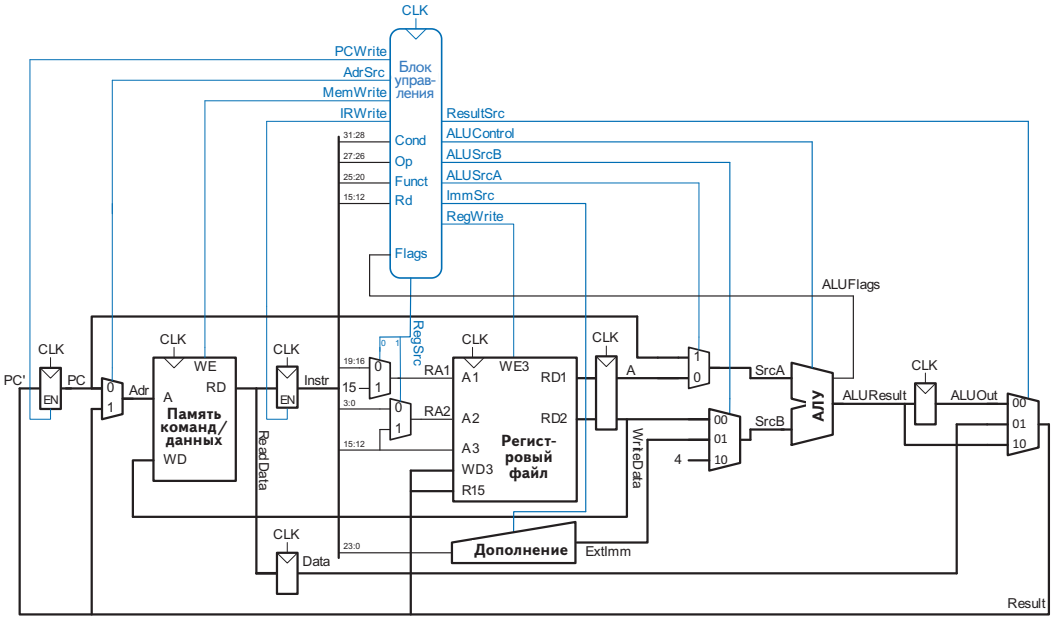


Рис. 2.30. Законченный многотактный процессор

Как и в однотактном процессоре, устройство управления поделено на декодер и блок условной логики, как показано на **Рис. 2.31 (a)**. Декодер, в свою очередь, состоит из частей, показанных на **Рис. 2.31 (b)**. Комбинационный главный декодер, который мы имели в однотактном процессоре, заменен главным конечным автоматом, который формирует последовательность управляющих сигналов в соответствующих тактах. Мы спроектируем главный конечный автомат в виде автомата Мура, так что выходы

будут зависеть только от текущего состояния. Однако по ходу проектирования мы обнаружим, что *ImmSrc* и *RegSrc* являются функциями *Op*, а не текущего состояния, поэтому для формирования этих сигналов понадобится также небольшой декодер команд, описанный в **Табл. 2.6**. Декодер АЛУ и блок логики PC не отличаются от разработанных для однотактного процессора. Блок условной логики почти совпадает с разработанным ранее. Мы добавили сигнал *NextPC*, который форсирует запись в PC при вычислении *PC* + 4. Мы также задерживаем *CondEx* на один такт перед подачей на вход *PCWrite*, *RegWrite* и *MemWrite*, чтобы флаги условий не были видны до окончания команды. Оставшуюся часть раздела посвятим разработке диаграммы состояний главного автомата.

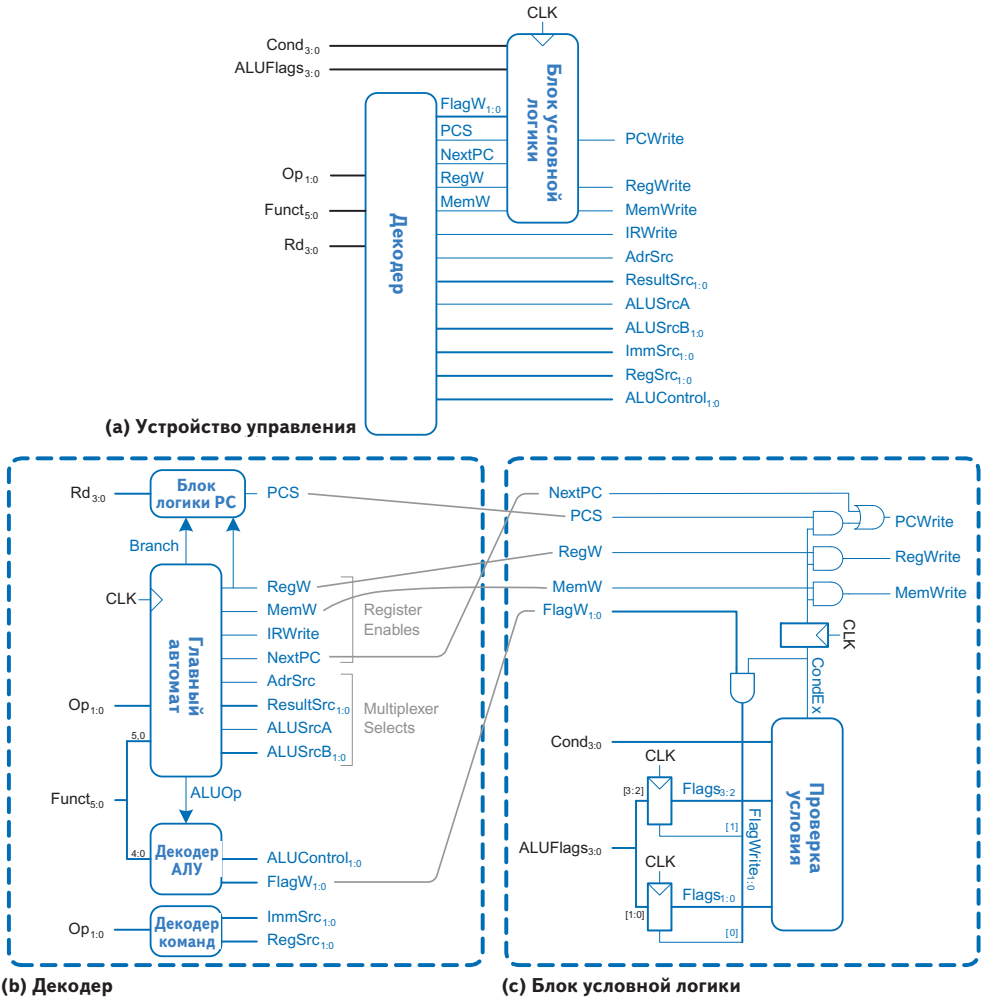


Рис. 2.31. Многотактное устройство управления

Таблица 2.6. Вычисление *RegSrc* и *ImmSrc* декодером команд

Команда	Op	$Funct_5$	$Funct_0$	$RegSrc_1$	$RegSrc_0$	$ImmSrc_{1:0}$
LDR	01	X	1	X	0	01
STR	01	X	0	1	0	01
ОД, непосредственная	00	1	X	X	0	00
ОД, регистровая	00	0	X	0	0	00
B	10	X	X	X	1	10

Главный автомат формирует сигналы управления мультиплексорами, сигналы разрешения записи в регистры и сигналы разрешения записи для тракта данных. Чтобы не загромождать диаграмму состояний, показаны только существенные для обсуждения управляющие сигналы. Сигналы управления мультиплексорами приводятся, только когда их значения важны. Сигналы записи (*RegW*, *MemW*, *IRWrite* и *NextPC*) приводятся, только когда они равны 1.

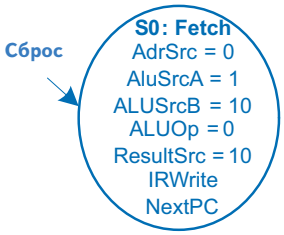


Рис. 2.32. Этап выборки команды

Первым этапом каждой команды является выборка из памяти по адресу, находящемуся в счетчике команд, и увеличение счетчика команд, так чтобы он указывал на следующую команду. В состояние Fetch автомат переходит по сигналу сброса (Reset). Управляющие сигналы показаны на **Рис. 2.32**. Поток данных на этом этапе приведен на **Рис. 2.33**. Выборка команды показана синим цветом, а увеличение счетчика команд – серым. Чтобы адрес для чтения был взят из счетчика команд, *AdrSrc* должен быть равен нулю. Чтобы прочитанное значение попало в регистр команд (*IR*), *IRWrite* устанавливается в единицу. Одновременно с этим счетчик команд должен быть увеличен на четыре – после этого он будет указывать на следующую команду. Так как АЛУ в этот момент свободно, процессор может использовать его для вычисления *PC* + 4 одновременно с выборкой команды из памяти. *ALUSrcA* = 1, поэтому на первый вход АЛУ (*SrcA*) подается *PC*. *ALUSrcB* = 10, поэтому на второй вход АЛУ (*SrcB*) подается константа 4. *ALUOp* = 0, так что декодер АЛУ устанавливает сигнал *ALUControl* равным 00, чтобы АЛУ выполнило именно сложение. Чтобы записать в счетчик команд значение *PC* + 4, *ResultSrc* устанавливается равным 10 (чтобы было выбрано *ALUResult*), а *NextPC* – равным 1 (чтобы разрешить *PCWrite*).

На втором этапе происходит чтение из регистра и (или) непосредственного операнда и декодирование команды. Регистры и непосредственные операнды выбираются, исходя из значений *RegSrc* и *ImmSrc*, вычисленных декодером команд по *Instr*. Бит $RegSrc_0$ должен быть равен 1, чтобы команда перехода в качестве *SrcA* прочитала *PC* + 8. Бит

$RegSrc_1$ должен быть равен 1, чтобы команда сохранения в качестве *SrcB* прочитала подлежащее сохранению значение. Сигнал *ImmSrc* должен быть равен 00, чтобы команда обработки данных выбрала 8-битовый непосредственный операнд, 01 – чтобы команда загрузки или сохранения выбрала 12-битовый непосредственный операнд, и 10 – чтобы команда перехода выбрала 24-битовый непосредственный операнд. Поскольку многотактный конечный автомат является автоматом Мура, для которого выходное значение зависит только от текущего состояния, он не может непосредственно формировать результат, зависящий от *Instr*. Мы могли бы построить автомат Мили, для которого выходное значение зависит не только от состояния, но и от *Instr*, но это было бы слишком хлопотно. Вместо этого выберем простейшее решение – сделать результаты комбинационными функциями от *Instr*, показанными в **Табл. 2.6**. Поскольку значения в некоторых ячейках таблицы несущественны, логику декодера команд можно упростить до:

$$RegSrc_1 = (\text{Op} == 01)$$
$$RegSrc_0 = (\text{Op} == 10)$$
$$ImmSrc_{1:0} = \text{Op}$$

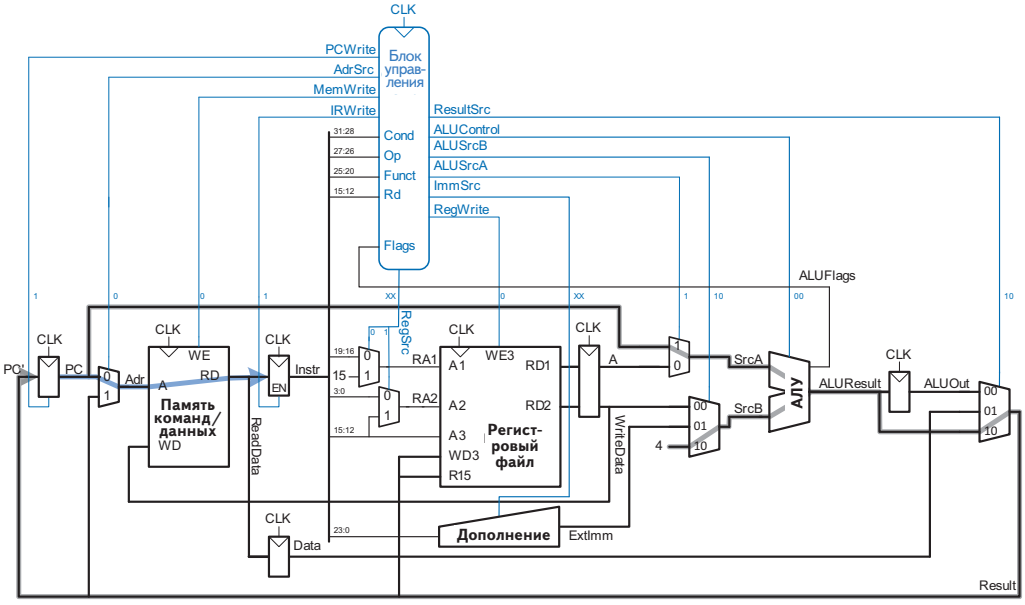


Рис. 2.33. Поток данных на этапе выборки

А тем временем АЛУ используется для вычисления *PC* + 8 путем прибавления константы 4 к значению *PC*, которое уже было увеличено на этапе выборки команды. Чтобы PC был выбран в качестве первого операнда АЛУ, формируется управляющий сигнал *ALUSrcA* = 1;

чтобы константа 4 была выбрана в качестве второго операнда – сигнал *ALUSrcB* = 10, а чтобы АЛУ выполнило сложение – сигнал *ALUOp* = 0. Сумма выбирается в качестве *Result* (*ResultSrc* = 10) и подается на вход R15 регистрового файла, чтобы чтение R15 возвращало *PC* + 8. Этап декодирования команды показан на **Рис. 2.34**, а поток данных на этом этапе – на **Рис. 2.35**, где вычисление R15 и чтение из регистрового файла выделены синим цветом.

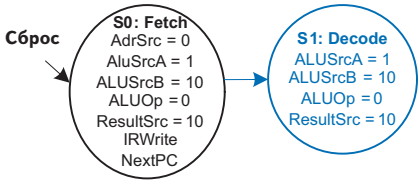


Рис. 2.34. Этап декодирования команды

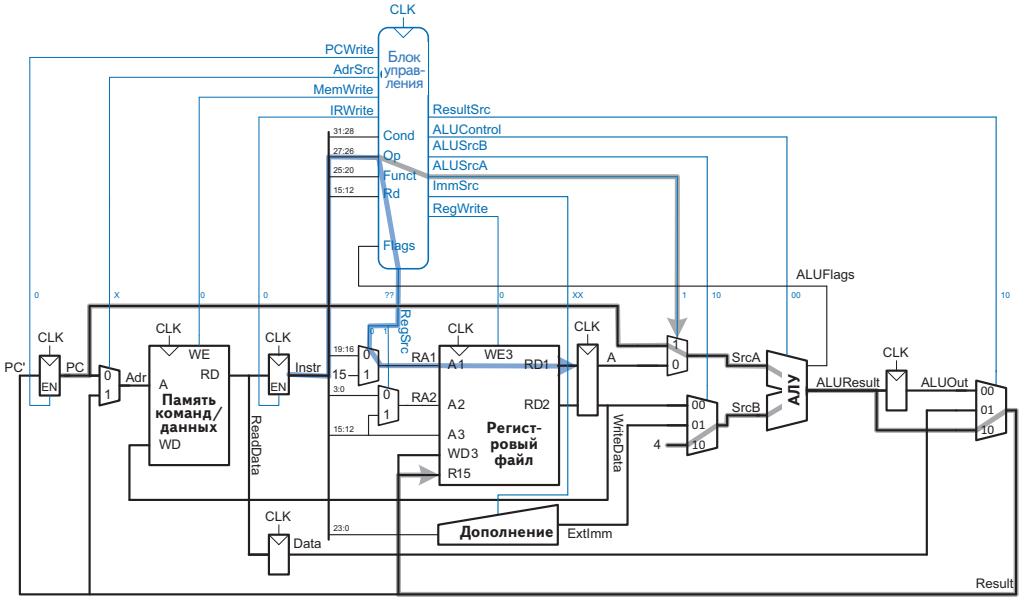


Рис. 2.35. Поток данных на этапе декодирования команды

Далее автомат переходит к одному из нескольких возможных последующих этапов в зависимости от значений полей *Op* и *Funct*, рассмотренных на этапе декодирования. Если команда читает или пишет в память (`LDR` или `STR`, *Op* = 01), то многотактный процессор вычисляет адрес путём добавления базового адреса к расширенному нулями смещению. Чтобы выбрать базовый адрес из регистрового файла, формируется сигнал *ALUSrcA* = 0, а чтобы выбрать смещение *ExtImm* – сигнал *ALUSrcB* = 01. *ALUOp* = 0 для выполнения сложения в АЛУ. Получен-

ный эффективный адрес сохраняется в регистре *ALUOut* для использования на следующем этапе. Соответствующее состояние *MemAdr* автомата изображено на **Рис. 2.36**, а поток данных – на **Рис. 2.37**.

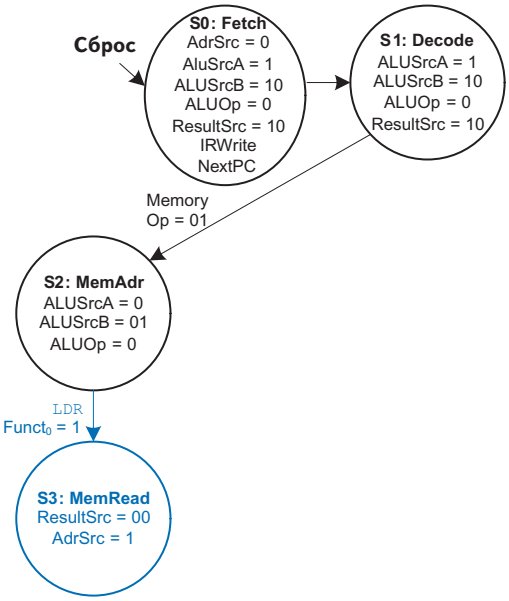


Рис. 2.36. Этап вычисления адреса памяти

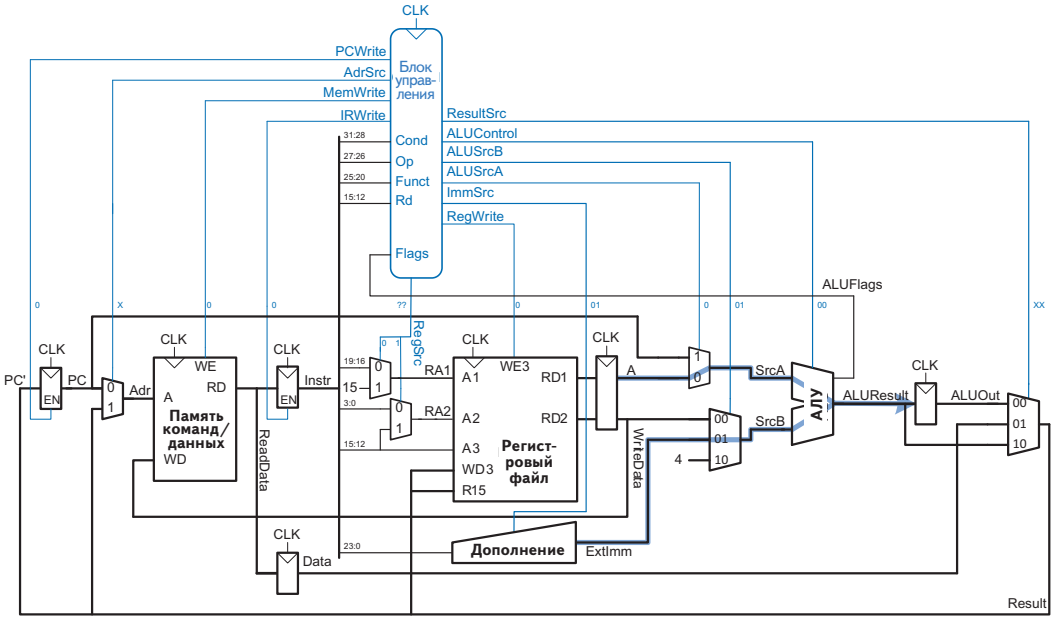


Рис. 2.37. Поток данных на этапе вычисления адреса памяти

В случае команды LDR ($Funct_0 = 1$) многотактный процессор должен на следующих этапах прочитать данные из памяти и записать их в регистровый файл. Эти два этапа показаны на **Рис. 2.38**. Для чтения из памяти формируются управляющие сигналы $ResultSrc = 00$ и $AdrSrc = 1$, чтобы был выбран только что вычисленный и сохраненный в $ALUOut$ адрес памяти. Этот адрес считывается и сохраняется в регистре $Data$ на этапе $MemRead$. Затем на этапе записи в память $MemWB$ значение из $Data$ записывается в регистровый файл. Сигнал $ResultSrc = 01$, чтобы $Result$ выбирался из $Data$, а $RegW$ устанавливается в 1, чтобы запись была произведена в регистровый файл, завершая команду LDR. Наконец, автомат возвращается в состояние выборки, чтобы приступить к обработке следующей команды. Поток данных на этих и последующих этапах постарайтесь изобразить самостоятельно.

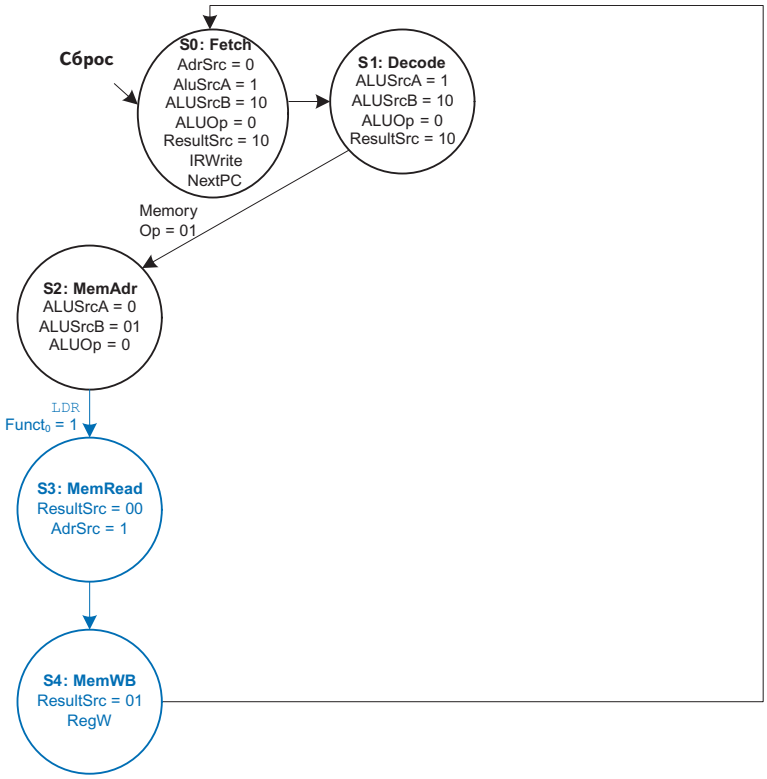


Рис. 2.38. Этап чтения из памяти

Если же выбрана команда STR ($Funct_0 = 0$), после состояния $MemAdr$ данные, прочитанные из второго порта регистрового файла, просто записываются в память. Соответствующее состояние $MemWrite$ устанавливает $ResultSrc = 00$ и $AdrSrc = 1$, чтобы был выбран адрес, вычисленный

в состоянии *MemAdr* и сохраненный в *ALUOut*. Сигнал *MemW* устанавливается в 1, чтобы была произведена запись в память. После этого автомат снова возвращается в состояние выборки. Состояние *MemWrite* показано на **Рис. 2.39**.

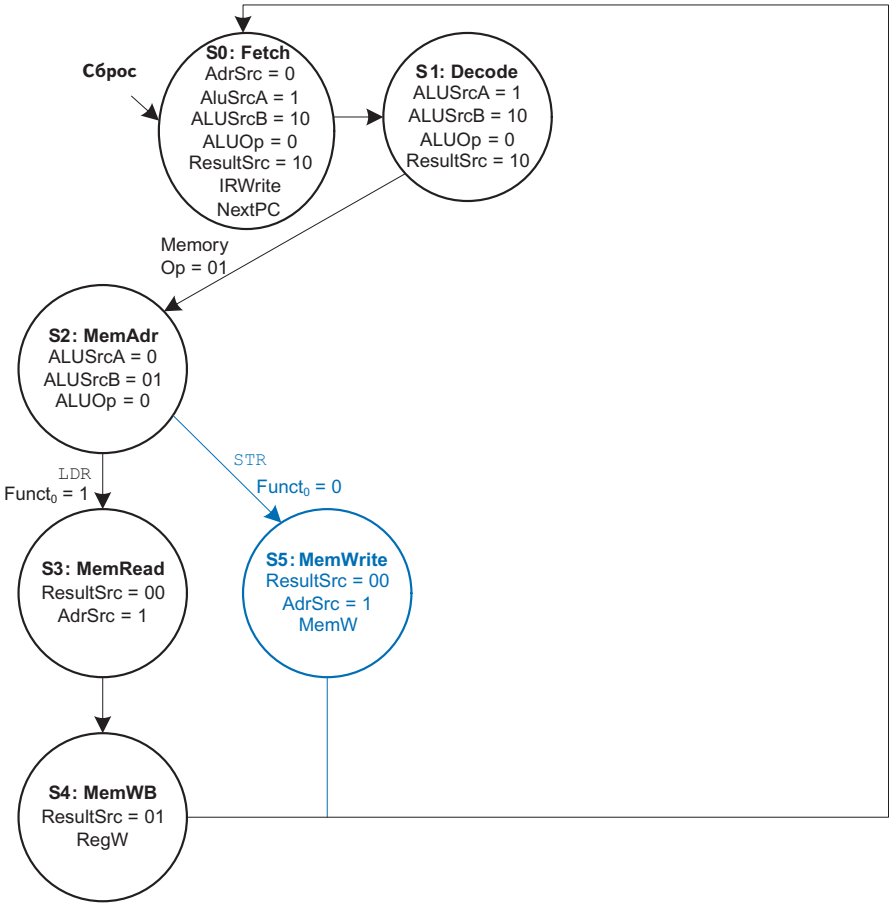


Рис. 2.39. Этап записи в память

Для команд обработки данных ($Op = 00$) многотактный процессор должен вычислить результат с помощью АЛУ и записать его в регистровый файл. Первый операнд всегда находится в регистре ($ALUSrcA = 0$). Сигнал $ALUOp = 1$, поэтому декодер АЛУ формирует *ALUControl* для конкретной команды в зависимости от поля *cmd* ($Funct_{4:1}$). Второй операнд берется из регистрового файла для команд с регистровой адресацией ($ALUSrcB = 00$) или из *ExtImm* для команд с непосредственной адресацией ($ALUSrcB = 01$). Таким образом, в автомате должны быть состояния *ExecuteR* и *ExecuteI*, соответствующие этим двум возмож-

ностям. В любом случае команда обработки данных переводит автомат в состояние *ALU Writeback* (*ALUWB*), в котором результат выбирается из *ALUOut* (*ResultSrc* = 00) и записывается в регистровый файл (*RegW* = 1). Все эти состояния показаны на **Рис. 2.40**.

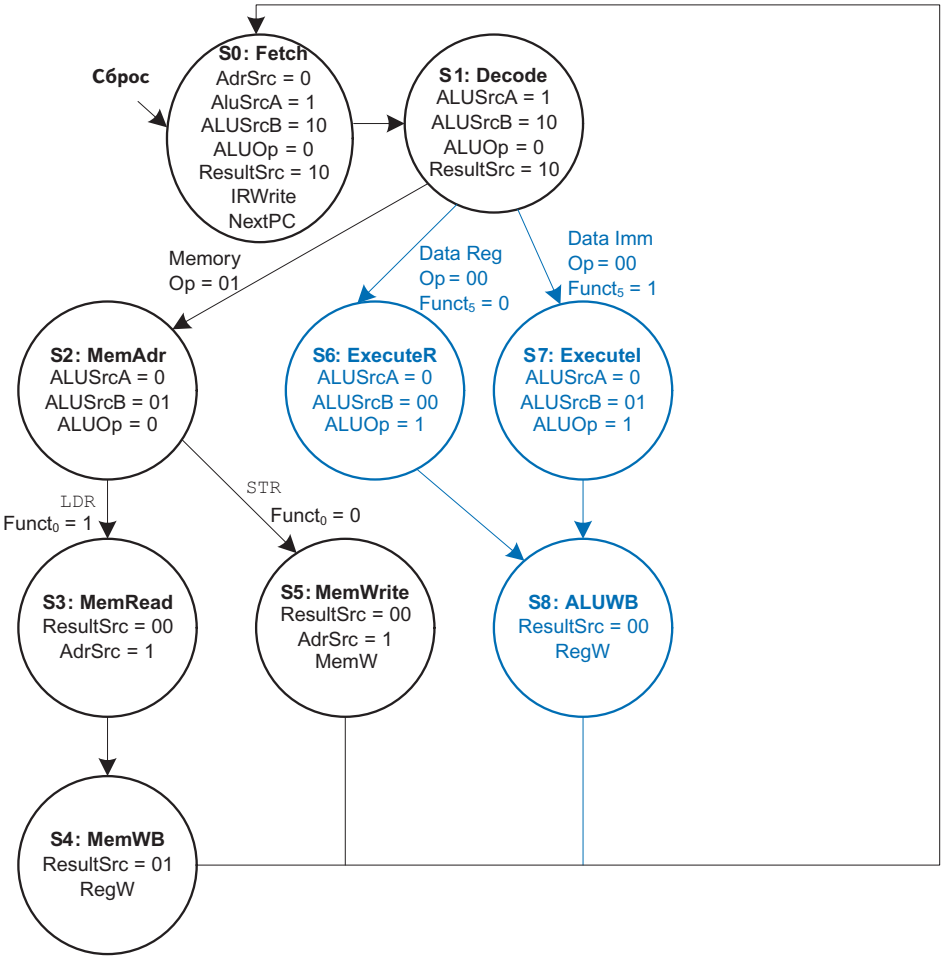
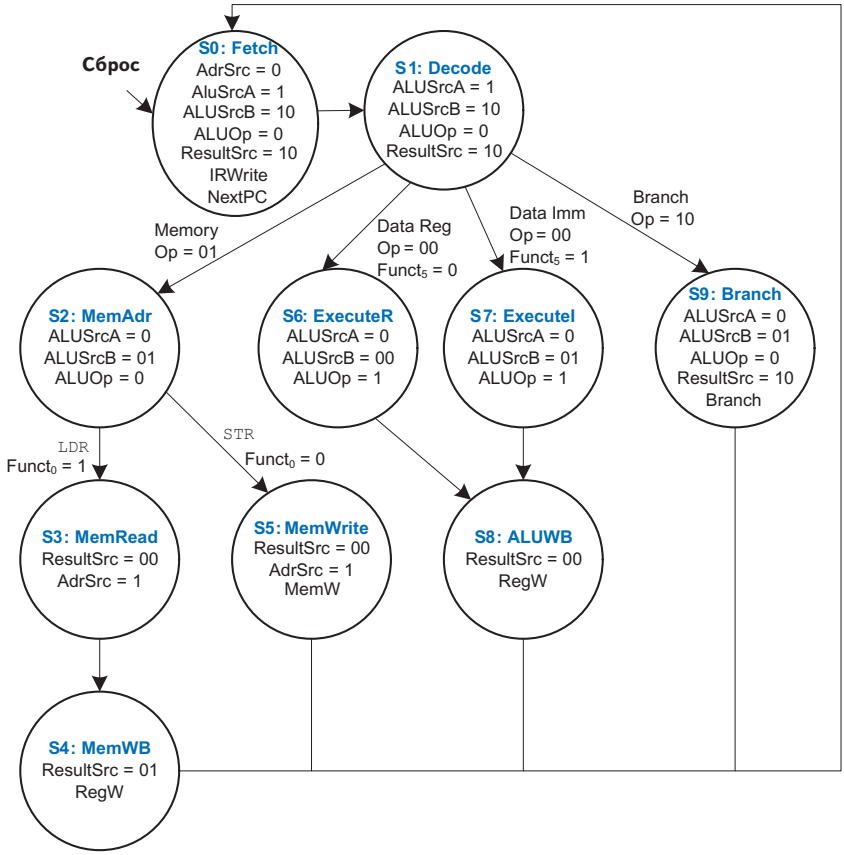


Рис. 2.40. Обработка данных

В случае команды перехода процессор должен вычислить конечный адрес (*PC* + 8 + смещение) и записать его в PC. В состоянии S1:Decode величина *PC* + 8 уже была вычислена и прочитана из порта *RD1* регистрового файла. Поэтому в состоянии Branch устройство управления формирует *ALUSrcA* = 0, чтобы выбрать R15 (*PC* + 8), *ALUSrcB* = 01, чтобы выбрать *ExtImm*, и *ALUOp* = 0, чтобы выполнить сложение. Мультиплексор *Result* выбирает *ALUResult* (*ResultSrc* = 10). Сигнал *Branch* устанавливается в 1, чтобы результат был записан в PC.

Полная диаграмма состояний главного автомата многотактного процессора показана на **Рис. 2.41**. Под рисунком описано назначение каждого состояния. Превращение этой диаграммы в логическую схему – простая, но утомительная задача, которую можно решить одним из способов, описанных в **главе 3** (книга 1). А еще лучше закодировать автомат на языке описания аппаратуры и синтезировать, как описано в **главе 4** (книга 1).



Состояние	Тракт данных μ Оп
Fetch	Instr ← Mem[PC]; PC ← PC+4
Decode	ALUOut ← PC+4
MemAdr	ALUOut ← Rn + Imm
MemRead	Data ← Mem[ALUOut]
MemWB	Rd ← Data
MemWrite	Mem[ALUOut] ← Rd
ExecuteR	ALUOut ← Rn op Rm
ExecuteI	ALUOut ← Rn op Imm
ALUWB	Rd ← ALUOut
Branch	PC ← R15 + offset

Рис. 2.41. Полная диаграмма состояний многотактного управляющего автомата

2.4.3. Анализ производительности

Время выполнения команды зависит от требуемого количества тактов и длительности такта. В отличие от однотактного процессора, который выполняет все команды за один такт, многотактному процессору для выполнения разных команд требуется разное число тактов. Однако поскольку он выполняет меньше действий за такт, его такты гораздо короче, чем у однотактного.

Для команд перехода многотактному процессору нужно три такта, для команд обработки данных и сохранения – четыре такта, а для команд загрузки – пять. Суммарное *число тактов на команду* (CPI) будет зависеть от частоты использования каждой команды.

Пример 2.5. CPI МНОГОТАКТНОГО ПРОЦЕССОРА

Эталонный тест SPECINT2000 содержит приблизительно 25% команд загрузки, 10% команд сохранения, 13% команд перехода и 52% команд обработки данных[2]. Определите среднее CPI для этого теста.

Решение. Среднее CPI можно вычислить, перемножив число тактов, требуемое для выполнения команды каждого типа, на относительное количество соответствующих команд и просуммировав полученные значения. Для этого теста среднее CPI = (0.13)(3) + (0.52 + 0.10)(4) + (0.25)(5) = 4.12. Если бы все команды выполнялись одинаковое время, то CPI было бы равно пяти.

Напомним, что мы спроектировали многотактный процессор так, чтобы на каждом такте он выполнял либо арифметическую операцию в АЛУ, либо операцию доступа к памяти, либо операцию доступа к регистровому файлу. Давайте предположим, что доступ к регистровому файлу быстрее, чем к памяти, и что запись в память быстрее, чем чтение. При внимательном изучении тракта данных оказывается, что существует два критических пути, лимитирующих минимальную длительность такта:

1) из PC через мультиплексор *SrcA*, АЛУ и мультиплексор *Result* в порт *R15* регистрового файла и в регистр *A*;
2) из *ALUOut* через мультиплексоры *Result* и *Adr* к чтению памяти в регистр *Data*.

$$T_{c2} = t_{pcq} + 2t_{mux} + \max[t_{ALU} + t_{mux}, t_{mem}] + t_{setup}. \quad (2.4)$$

Численные значения этих времен зависят от конкретной технологии производства.

[2] Данные взяты из книги: *Patterson, Hennessy*. Computer Organization and Design. 4-е изд. Morgan Kaufmann, 2011.

Пример 2.6. СРАВНЕНИЕ ПРОИЗВОДИТЕЛЬНОСТИ ПРОЦЕССОРОВ

Бен Битдидл задумался о том, не будет ли многотактный процессор быстрее однотактного. В обоих случаях он планирует использовать 16-нм КМОП-техпроцесс. Задержки, характерные для этого техпроцесса, приведены в **Табл. 2.5**. Помогите ему сравнить время выполнения теста SPECINT2000, состоящего из 100 миллиардов команд, для обоих процессоров (см. **пример 2.5**).

Решение. Согласно уравнению (2.4), длительность такта многотактного процессора равна $T_{c2} = 40 + 2(25) + 200 + 50 = 340$ пс. Используя значение CPI, рассчитанное в **примере 2.5** и равное 4.12, получаем, что общее время выполнения программы многотактным процессором равно $T_2 = (100 \times 10^9 \text{ команд})(4.12 \text{ такта/команду})(340 \times 10^{-12} \text{ с/такт}) = 140$ с. Как мы выяснили в **примере 2.4**, общее время выполнения однотактным процессором равно 84 с.

При разработке многотактного процессора мы хотели избежать того, чтобы все команды выполнялись с той же скоростью, что и самая медленная. К сожалению, этот пример показывает, что для данных значений CPI и задержек элементов многотактный процессор медленнее, чем однотактный. Фундаментальная проблема оказалась в том, что, несмотря на разбиение самой медленной команды (`LDR`) на пять этапов, длительность такта в многотактном процессоре уменьшилась вовсе не в пять раз. Одной из причин явилось то, что длительности этапов не одинаковы. Другой причиной стали накладные расходы, связанные с временными регистрами, – задержка в 90 пс, равная сумме времени предустановки и времени срабатывания регистра, теперь добавляется не к общему времени выполнения команды, а к каждому этапу в отдельности. Инженеры выяснили, что в общем случае довольно трудно использовать тот факт, что одни вычисления могут происходить быстрее, чем другие, если разница во времени вычислений мала.

В сравнении с однотактным процессором многотактный процессор, скорее всего, окажется дешевле, так как нет необходимости в двух дополнительных сумматорах, а память команд и данных объединена в один блок. Правда, взамен ему требуется пять неархитектурных регистров и несколько дополнительных мультиплексоров.

2.5. Конвейерный процессор

Конвейеризация, о которой мы говорили в **разделе 3.6** (книга 1), является мощным средством увеличения пропускной способности цифровой системы. Мы спроектируем конвейерный процессор, разделив однотактный процессор на пять стадий. Таким образом, пять команд смогут выполняться одновременно, по одной на каждой стадии. Так как каждая стадия содержит только одну пятую от всей логики процессора, то частота тактового сигнала может быть почти в пять раз выше. В идеальном случае латентность команд не изменится, а пропускная способность вырастет в пять раз. Микропроцессоры выполняют миллионы и миллиарды команд в секунду, так что пропускная способность важнее, чем латент-

ность. Конвейеризация требует определенных накладных расходов, так что в реальной жизни пропускная способность будет ниже, чем в идеальном случае, но все равно у конвейеризации так много преимуществ, а обходится она так дешево, что все современные высокопроизводительные микропроцессоры – конвейерные.

Чтение и запись в память и регистровый файл, а также использование АЛУ обычно привносят наибольшие задержки в процессор. Мы поделим конвейер на пять стадий таким образом, чтобы каждая из них включала ровно одну из этих медленных операций. Стадии назовем *Fetch* (Выборка), *Decode* (Декодирование), *Execute* (Выполнение), *Memory* (Доступ к памяти) и *Writeback* (Запись результатов). Они похожи на пять этапов выполнения команды `LDR` в многотактном процессоре. На стадии *Fetch* процессор читает команду из памяти команд. На стадии *Decode* процессор читает операнды-источники из регистрового файла и декодирует команду, чтобы сформировать управляющие сигналы. На стадии *Execute* процессор выполняет вычисления в АЛУ. На стадии *Memory* процессор читает или пишет в память данных. Наконец, на стадии *Writeback* процессор, если нужно, записывает результат в регистровый файл.

На **Рис. 2.42** приведена временная диаграмма для сравнения однотактного и конвейерного процессоров. По горизонтальной оси отложено время, а по вертикальной – команды. Значения задержек логических элементов на диаграмме взяты из **Табл. 2.5**; задержками мультиплексоров и регистров мы пренебрежем. В однотактном процессоре, показанном на **Рис. 2.42 (a)**, первая команда выбирается из памяти в момент времени 0, далее процессор читает операнды из регистрового файла, после чего АЛУ выполняет необходимые вычисления. Наконец, может быть произведен доступ к памяти данных, и результат записывается в регистровый файл через 680 пс. Выполнение второй команды начинается после завершения первой. Таким образом, на этой диаграмме однотактный процессор обеспечивает латентность команд, равную 250 + 100 + 120 + 200 + 60 = 680 пс, и пропускную способность, равную одной команде за 680 пс (1.47 миллиарда команд в секунду).

В конвейерном процессоре, приведенном на **Рис. 2.42 (b)**, длина стадии равна 200 пс и определяется самой медленной стадией – доступа к памяти (Fetch или Memory). В начальный момент времени первая команда выбирается из памяти. Через 200 пс первая команда попадает на стадию Decode, а вторая выбирается из памяти. По прошествии 400 пс первая команда начинает выполняться, вторая попадает на стадию Decode, а третья выбирается. И так далее, пока все команды не завершатся. Латентность команд составляет 5 × 200 = 1000 пс. Пропускная способность – одна команда за 200 пс (пять миллиардов команд в секунду). Латентность команд в конвейерном процессоре немного больше, чем в однотактном, потому что стадии конвейера не идеально сбалансированы, то есть не содержат абсолютно одинаковое количество логики. Ана-

логично пропускная способность процессора с пятистадийным конвейером не в пять раз больше, чем у однотактного. Тем не менее увеличение пропускной способности весьма значительно.

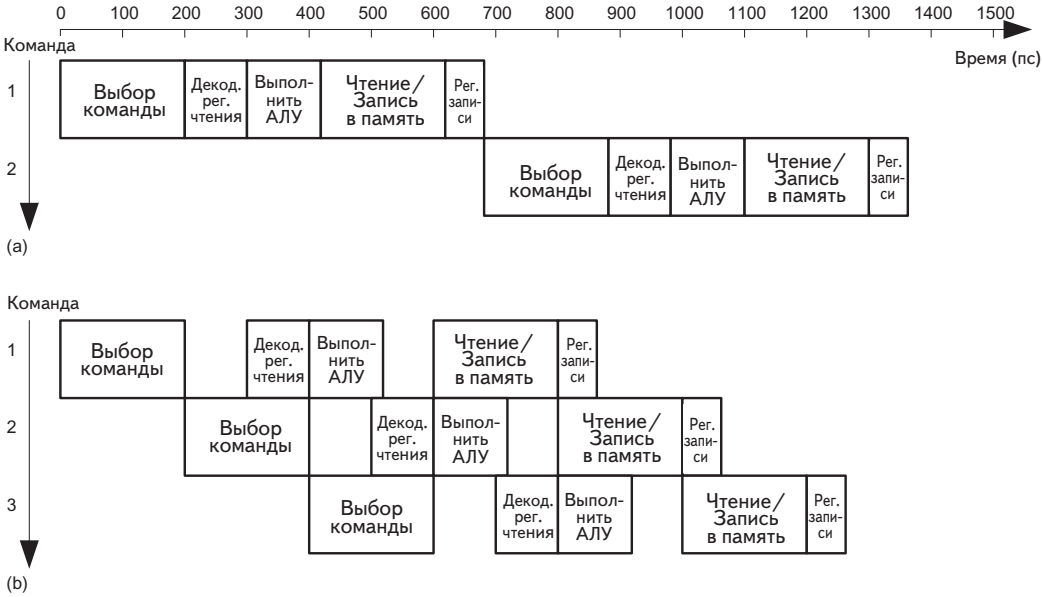


Рис. 2.42. Временная диаграмма (a) однотактного и (b) конвейерного процессоров

На **Рис. 2.43** показано абстрактное представление работающего конвейера, иллюстрирующее продвижение команд по нему. Каждая стадия представлена своим основным компонентом – память команд (instruction memory, IM), чтение регистрового файла (register file, RF), выполнение в АЛУ, память данных (DM) и запись в регистровый файл. Если смотреть на строки, то можно узнать, на каком такте команда находится в той или иной стадии. Например, команда `SUB` выбирается из памяти на третьем такте и выполняется на пятом. Если смотреть на столбцы, то можно узнать, чем заняты стадии конвейера на конкретном такте. Например, на шестом такте команда `ORR` выбирается из памяти команд, регистр R1 читается из регистрового файла, АЛУ вычисляет логическое И регистров R12 и R13, память данных не используется и простаивает, а в регистр R3 записывается результат сложения. Используемые стадии закрашены серым цветом. Например, память данных используется командой `LDR` на четвертом такте и командой `STR` на восьмом. Память команд и АЛУ используются на каждом такте. Результат записывается в регистровый файл всеми командами, кроме `STR`. В конвейерном процессоре значения записываются в регистровый файл в первой части такта, а читаются во

второй, что также отмечено на рисунке. Таким образом, данные могут быть записаны и затем прочитаны обратно в одном такте.

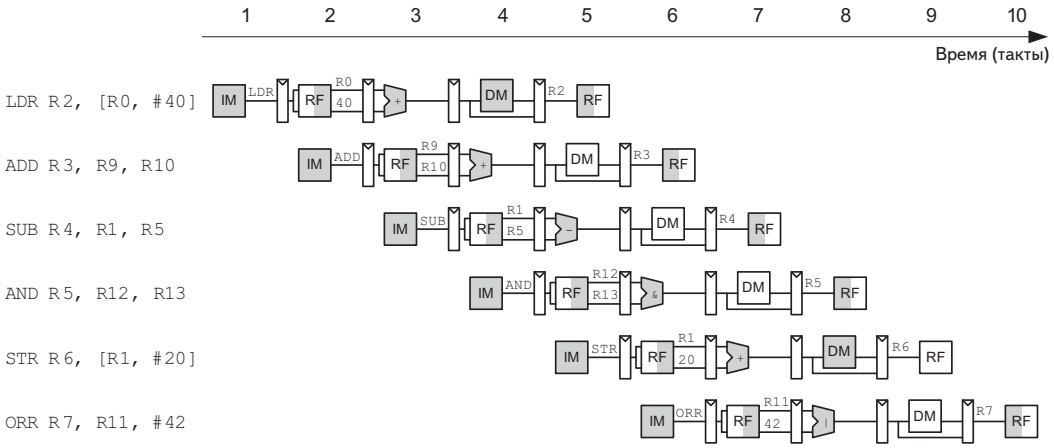


Рис. 2.43. Абстрактное представление работающего конвейера

Главная проблема в конвейерных системах – разрешение *конфликтов* (hazard), которые возникают, когда результаты одной из команд требуются для выполнения последующей команды, до того как первая завершится. Например, если бы в команде `ADD` на **Рис. 2.43** использовался регистр R2 вместо R10, то возник бы конфликт, потому что регистр R2 еще не был бы записан командой `LDR` в тот момент, когда команда `ADD` должна была бы его прочитать. В этом разделе мы рассмотрим несколько методов разрешения конфликтов: *байпас* (forwarding), *приостановку* (stall) и *очистку* (flush) конвейера. И в конце еще раз оценим производительность, приняв во внимание накладные расходы на организацию конвейерной обработки (sequencing) и влияние конфликтов.

2.5.1. Конвейерный тракт данных

Конвейерный тракт данных можно получить, порезав однотактный тракт данных на пять стадий, разделенных конвейерными регистрами (pipeline register).

На **Рис. 2.44 (a)** показан однотактный тракт данных, растянутый таким образом, чтобы оставить место для конвейерных регистров между стадиями. На **Рис. 2.44 (b)** показан конвейерный тракт данных, поделенный на пять стадий путем вставки в него четырех конвейерных регистров. Названия стадий и границы между ними показаны синим цветом. Ко всем сигналам добавлен суффикс (F, D, E, M или W), показывающий, к какой стадии они относятся.

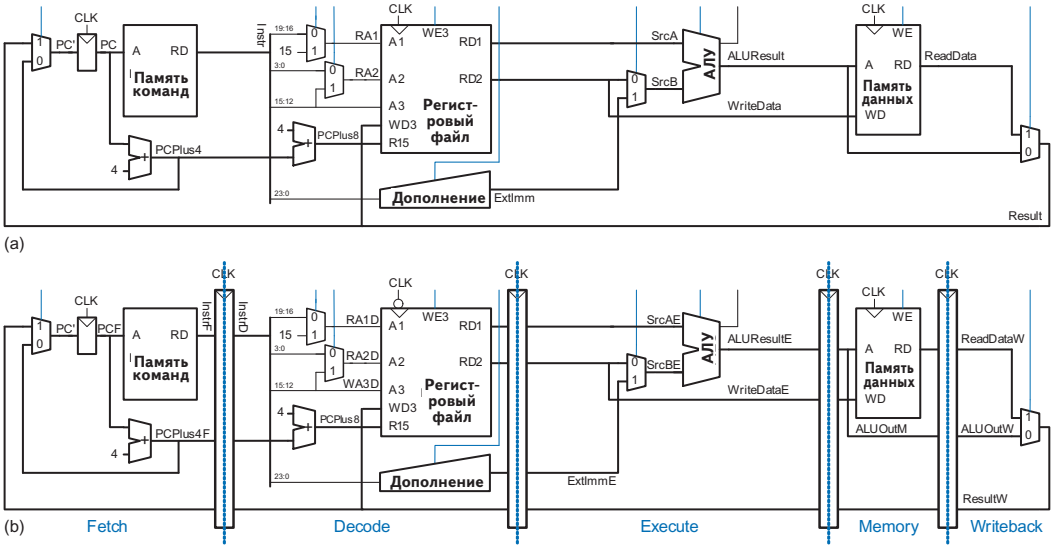


Рис. 2.44. Тракты данных: (a) однотактный и (b) конвейерный

Регистровый файл особенный в том смысле, что процессор читает из него на стадии *Decode*, а пишет на стадии *Writeback*. Поэтому, несмотря на то что на рисунке он находится на стадии *Decode*, адрес и данные для записи приходят со стадии *Writeback*. Эта обратная связь будет приводить к конфликтам конвейера, которые мы рассмотрим в **разделе 2.5.3**. В конвейерном процессоре значения записываются в регистровый файл по отрицательному фронту сигнала синхронизации *CLK*, поэтому запись результата может производиться в первой половине такта, а чтение для использования в следующей команде – во второй половине.

Одна тонкая, но чрезвычайно важная проблема организации конвейерной обработки данных состоит в том, что все сигналы, относящиеся к конкретной команде, должны обязательно продвигаться по конвейеру одновременно друг с другом, в унисон. На **Рис. 2.44 (b)** есть связанная с этим ошибка. Можете ли вы ее найти?

Ошибка – в логике записи в регистровый файл на стадии Writeback. Значение поступает в виде сигнала *ResultW* со стадии Writeback, но в качестве адреса записи используется значение $InstrD_{15:12}$ (называемое также *WA3D*), а это сигнал со стадии Decode. В конвейерной диаграмме на **Рис. 2.44** на пятом такте результат команды `LDR` был бы ошибочно записан в регистр R5, а не в R2.

На **Рис. 2.45** показан исправленный тракт данных, изменения выделены черным цветом. Сигнал *WA3* теперь проходит через стадии Execution, Memory и Writeback, поэтому остается синхронизированным с остальными частями команды. Теперь *WA3W* и *ResultW* совместно подаются на входы регистрового файла на стадии *Writeback*.

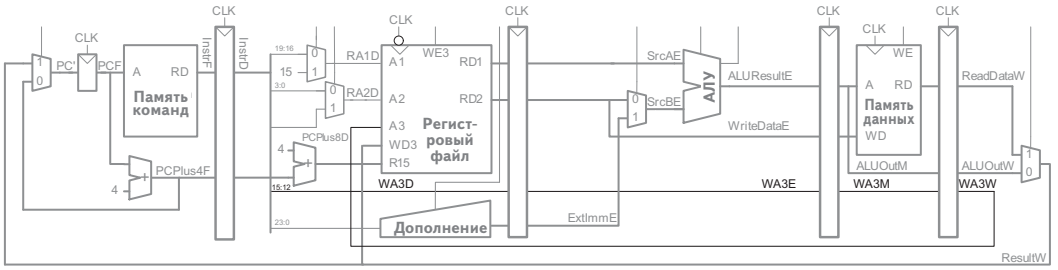


Рис. 2.45. Исправленный конвейерный тракт данных

Внимательный читатель, вероятно, заметил, что в логике сигнала *PC'* тоже есть проблемы, потому что этот сигнал может понадобиться изменить одновременно на стадиях Fetch и Memory (используя сигналы *PCPlus4F* или *ResultW* соответственно). В **разделе 2.5.3** мы покажем, как разрешить данный конфликт.

На **Рис. 2.46** показана еще одна оптимизация, помогающая избавиться от одного 32-битового сумматора и регистра в логике работы с PC. Заметим, что на **Рис. 2.45** всякий раз, как увеличивается счетчик команд, *PCPlus4F* одновременно записывается в PC и в конвейерный регистр между стадиями Fetch и Decode. Кроме того, на следующем такте значение в обоих этих регистрах снова увеличивается на 4. Следовательно, сигнал *PCPlus4F* для команды на стадии Fetch логически эквивалентен *PCPlus8D* для команды на стадии Decode. Пробрасывание этого сигнала дальше позволяет обойтись без конвейерного регистра и второго сумматора[3].

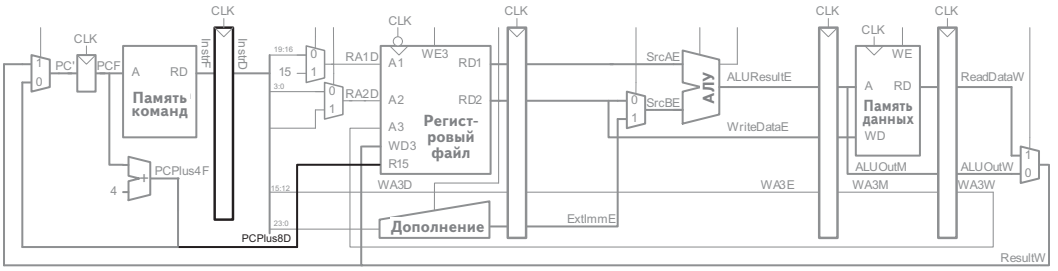


Рис. 2.46. Оптимизированная логика работы с PC с устранением одного регистра и одного сумматора

2.5.2. Конвейерное устройство управления

Конвейерный процессор использует те же управляющие сигналы, что и однотактный процессор, поэтому и устройство управления такое же. На

[3] Это упрощение влечет за собой потенциальную проблему в случае, когда PC записывается с помощью сигнала *ResultW*, а не *PCPlus4F*. Однако этот случай разрешается в разделе 2.5.3 путем очистки конвейера, поэтому *PCPlus8D* становится несущественным, и конвейер по-прежнему работает правильно.

стадии *Decode* оно в зависимости от полей *Op* и *Funct* команды формирует управляющие сигналы, как было показано в **разделе 2.3.2**. Эти управляющие сигналы должны передаваться по конвейеру вместе с данными, чтобы сохранялась синхронизация с командой. Устройство управления анализирует также поле *Rd* при обработке записи в регистр R15 (PC).

Весь конвейерный процессор вместе с устройством управления показан на **Рис. 2.47**. Аналогично сигналу *WA3* на **Рис. 2.45**, сигнал *RegWrite* обязательно должен дойти по конвейеру до стадии *Writeback*, перед тем как попасть на вход регистрового файла.

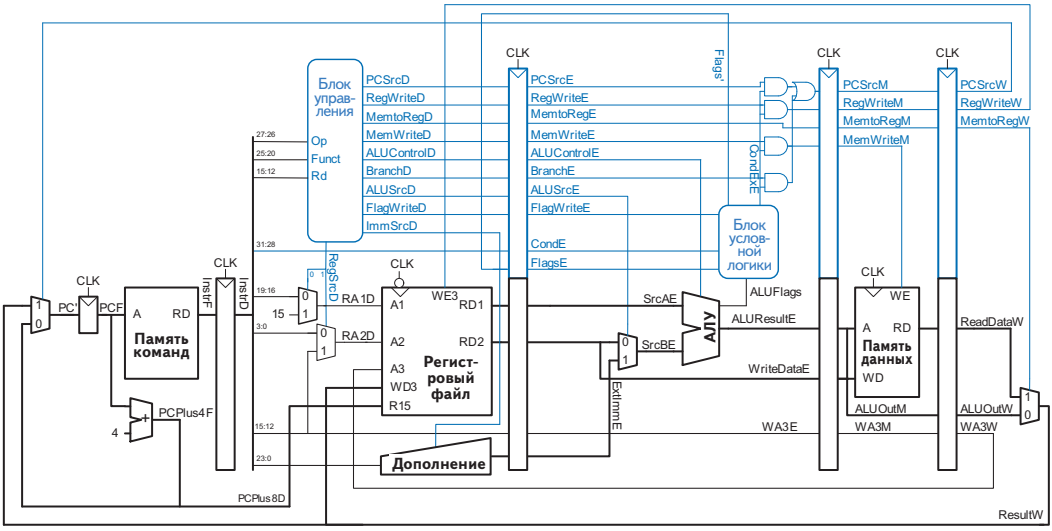


Рис. 2.47. Конвейерный процессор с устройством управления

2.5.3. Конфликты

В конвейерном процессоре выполняется несколько команд одновременно. Когда одна из них *зависит* от результатов другой, еще не завершенной команды, говорят, что произошел *конфликт* (hazard).

Процессор может читать и писать в регистровый файл в одном такте. Запись происходит в первой части такта, а чтение во второй, так что значение в регистр можно записать и затем прочитать обратно в одном такте, и это не приведет к конфликту.

На **Рис. 2.48** показаны конфликты, которые возникают, когда одна команда пишет в регистр (R1), а следующие за ней читают из него. Это называется *конфликтом чтения после записи* (read after write, RAW). Команда `ADD` записывает результат в R1 в первой половине пятого такта, однако команда `AND` читает R1 в третьем такте, то есть получает неверное значение. Команда `ORR` читает R1 в четвертом такте и тоже получает не-

верное значение. Команда SUB читает R1 во второй половине пятого такта, то есть наконец-то получает верное значение, которое было записано в первой половине пятого такта. Все последующие команды также прочитают верное значение из R1. Как видно из диаграммы, конфликт в конвейере возникает тогда, когда команда записывает значение в регистр и хотя бы одна из следующих двух команд читает его. Если не принять мер, конвейер вычислит неправильный результат.

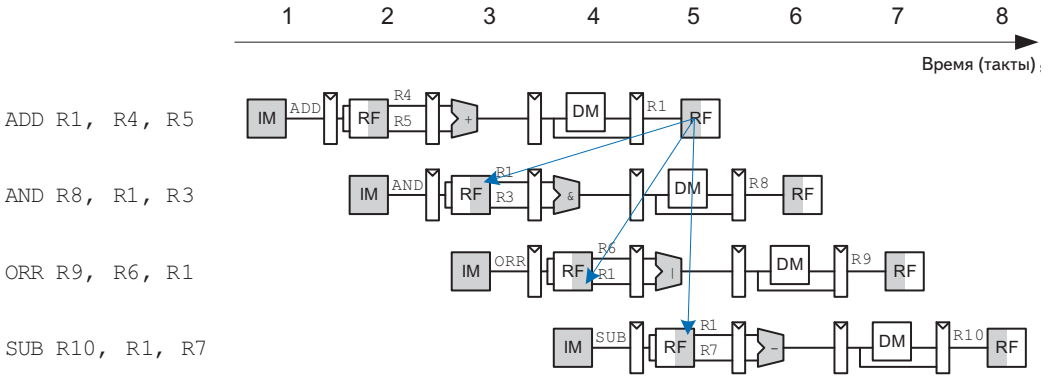


Рис. 2.48. Абстрактная диаграмма конвейера, иллюстрирующая конфликты

Программно проблему можно было бы решить, заставив программиста или компилятор вставлять команды NOP между ADD и AND, так чтобы зависимая команда не читала результат (R1), пока он не попадет в регистровый файл (**Рис. 2.49**). Но такая *программная блокировка* (software interlock) усложняет программирование и ведет к снижению производительности, поэтому решение не идеально.

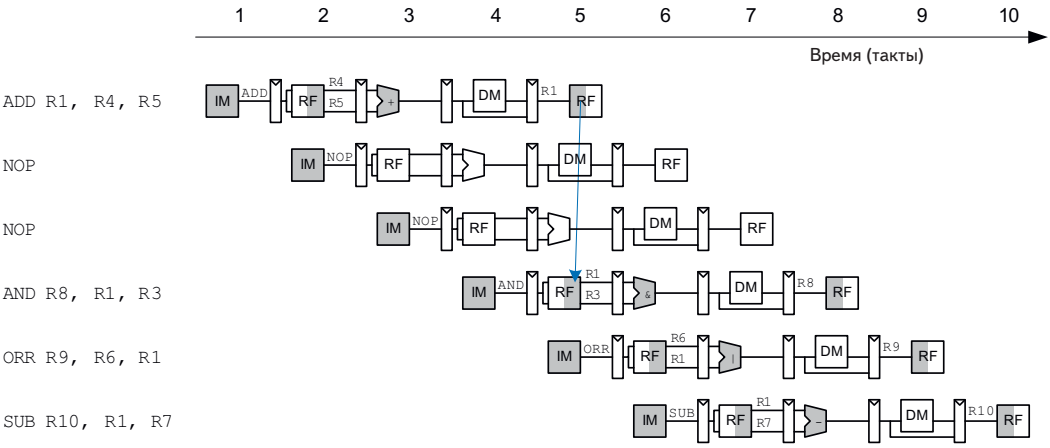


Рис. 2.49. Разрешение конфликтов с помощью вставки NOP

Однако при ближайшем рассмотрении **Рис. 2.48** оказывается, что результат команды ADD вычисляется в АЛУ на третьем такте, а команде AND он требуется лишь на четвертом. В принципе, мы могли бы пробросить результат выполнения первой команды следующей за ней, не дожидаясь его появления в регистровом файле, и тем самым разрешить конфликт чтения после записи без необходимости замедлять конвейер. Часть тракта данных, обеспечивающая такой проброс, называется *байпасом*. В других ситуациях, которые мы рассмотрим далее в этом разделе, конвейер все-таки придется приостанавливать, чтобы дать процессору время вычислить требуемый результат, до того как он понадобится последующим командам. В любом случае, чтобы программа выполнялась корректно, несмотря на конвейеризацию, мы должны предпринять какие-то действия для разрешения конфликтов.

Конфликты можно разделить на конфликты по данным и конфликты управления. *Конфликт по данным* происходит, когда команда пытается прочитать из регистра значение, которое еще не было записано предыдущей командой. *Конфликт управления* происходит, когда решение о том, какую команду выбирать следующей, не было принято к моменту выборки. В оставшейся части этого раздела мы добавим в конвейерный процессор блок *разрешения конфликтов* (hazard unit), который будет выявлять и разрешать конфликты таким образом, чтобы процессор выполнял программу корректно.

Разрешение конфликтов пересылкой через байпас

Некоторые конфликты по данным можно разрешить путем пробрасывания *через байпас* (bypassing, также используется термин forwarding) результата со стадий Memory или Writeback в ожидающую этого результата команду, находящуюся на стадии Execute. Чтобы организовать байпас, придется добавить мультиплексоры перед АЛУ, чтобы операнд можно было получить либо из регистрового файла, либо напрямую со стадий Memory или Writeback, как показано на **Рис. 2.50**. На четвертом такте R1 пробрасывается со стадии Memory команды ADD на стадию Execute зависимой от нее команды AND. На пятом такте R1 пробрасывается со стадии Writeback команды ADD на стадию Execute зависимой от нее команды ORR.

Пробрасывание через байпас необходимо, если номер любого из регистров-источников команды, находящейся в стадии Execute, равен номеру регистра-приемника команды, находящейся в стадии Memory или Writeback. На **Рис. 2.51** показан модифицированный конвейерный процессор с байпасом. В него добавлен *блок разрешения конфликтов* и два новых *пробрасывающих мультиплексора*. Блок разрешения конфликтов получает на входе четыре сигнала совпадения (для краткости обозначенные на **Рис. 2.51** *Match*), которые показывают, совпадают ли регистры-источники на стадии Execute с регистрами-приемниками на стадиях Memory и Writeback:

```
Match_1E_M = (RA1E == WA3M)
Match_1E_W = (RA1E == WA3W)
Match_2E_M = (RA2E == WA3M)
Match_2E_W = (RA2E == WA3W)
```

Блок разрешения конфликтов также получает сигналы RegWrite со стадий Memory и Writeback. Эти сигналы показывают, нужно ли на самом деле писать в регистр-приемник или нет (например, команды `STR` и `B` не записывают результаты в регистровый файл, поэтому и пробрасывать их не нужно). Отметим, что эти сигналы *соединены по имени*. Иными словами, чтобы не загромождать диаграмму линиями, соединяющими управляющие сигналы вверху и блок разрешения конфликтов внизу, соединения изображены короткими отрезками, помеченными названиями соответствующих сигналов. Логика сигналов Match и конвейерные регистры для *RA1E* и *RA2E* опущены из тех же соображений.

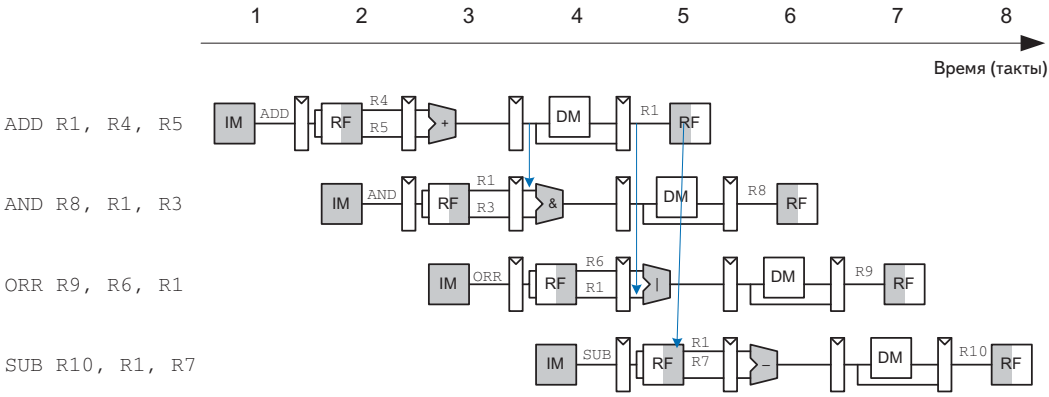


Рис. 2.50. Абстрактная диаграмма конвейера, иллюстрирующая пробрасывание

Блок разрешения конфликтов формирует управляющие сигналы для пробрасывающих мультиплексоров, которые определяют, брать ли операнды из регистрового файла или пробросить напрямую со стадии Memory или Writeback (*ALUOutM* или *ResultW*). Если в одной из этих стадий происходит запись в регистр-приемник и номер этого регистра совпадает с номером регистра-источника, то используется байпас. Если номера регистров-приемников на стадиях Memory и Writeback одинаковы, то приоритет отдается стадии Memory, т. к. она содержит команду, выполненную позже. Ниже приведена функция, определяющая логику пробрасывания для операнда *SrcAE*. Логика для операнда *SrcBE* (*ForwardBE*) точно такая же, с тем исключением, что проверяется сигнал *Match_2E*.

```
if      (Match_1E_M • RegWriteM) ForwardAE = 10; // SrcAE = ALUOutM
else if (Match_1E_W • RegWriteW) ForwardAE = 01; // SrcAE = ResultW
else                             ForwardAE = 00; // SrcAE из регистрового
                                                 // файла
```

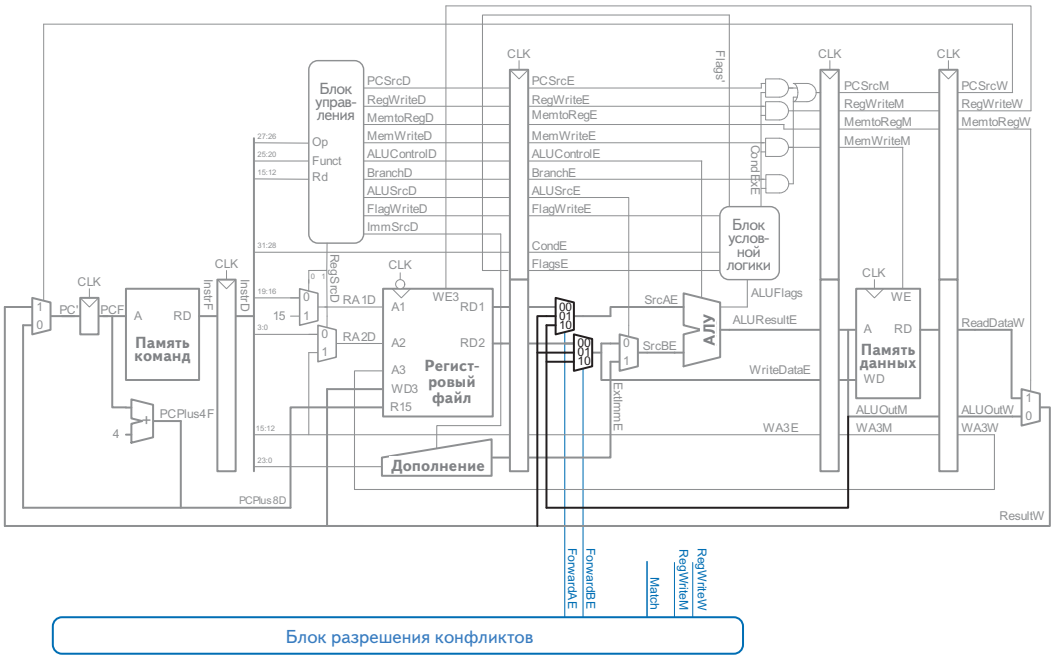


Рис. 2.51. Конвейерный процессор с пробрасыванием для разрешения конфликтов

Разрешение конфликтов данных путем приостановки конвейера

Пробрасывание данных может разрешить конфликт чтения после записи, только если результат вычисляется на стадии Execute, потому что только в этом случае его можно сразу пробросить на стадию Execute следующей команды. К сожалению, команда `LDR` не может прочитать данные раньше, чем в конце стадии Memory, поэтому ее результат нельзя пробросить на стадию Execute следующей команды. В этом случае мы будем говорить, что латентность команды `LDR` равна двум тактам, потому что зависимая команда не может использовать ее результат раньше, чем через два такта. Эта проблема показана на **Рис. 2.52**. Команда `LDR` получает данные из памяти в конце четвертого такта. Однако команде `AND` эти данные требуются в качестве операнда-источника уже в самом начале четвертого такта. Пробрасывание данных через байпас не поможет разрешить этот конфликт.

Альтернативное решение – *приостановить* конвейер, задержав все операции, до тех пор пока данные не станут доступны. На **Рис. 2.53** показана приостановка зависимой команды (`AND`) на стадии Decode. Команда `AND` попадает на эту стадию в третьем такте и остается там в четвер-

том такте. Следующая команда (ORR) должна также оставаться на стадии Fetch в течение обоих тактов, так как стадия Decode занята.

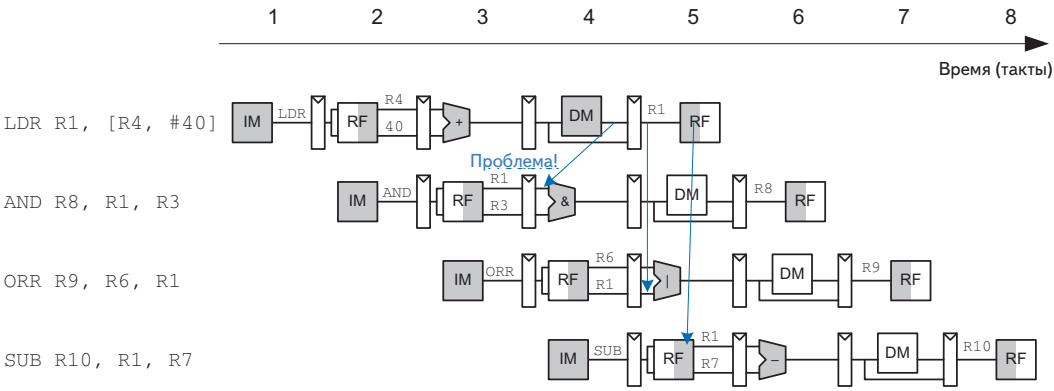


Рис. 2.52. Абстрактная диаграмма конвейера, иллюстрирующая проблему с пробрасыванием в случае команды LDR

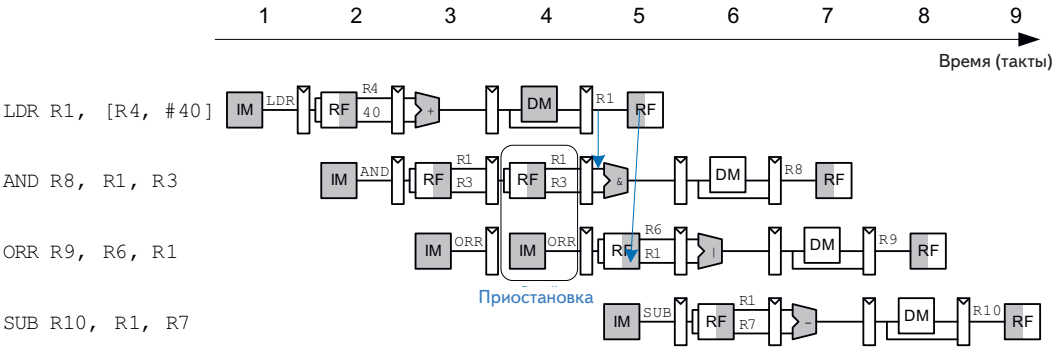


Рис. 2.53. Абстрактная диаграмма конвейера, иллюстрирующая, как приостановка разрешает конфликты

На пятом такте результат можно пробросить со стадии Writeback команды LDR на стадию Execute команды AND. На этом же такте операнд R1 команды ORR может быть прочитан прямо из регистрового файла, без какого-либо пробрасывания.

Отметим, что стадия Execute на четвертом такте не используется. Аналогично стадия Memory не используется на пятом такте, а Writeback – на шестом. Эта неиспользуемая стадия, распространяющаяся по конвейеру, называется *пузырьком* (bubble), и ведет себя так же, как команда NOP. Пузырек получается путем обнуления всех управляющих сигналов стадии Execute на время приостановки стадии Decode, так что он не приводит ни к каким действиям и не изменяет архитектурное состояние.

Короче говоря, для приостановки стадии конвейера следует запретить обновление конвейерного регистра. Если какая-либо стадия приоста-

новлена, то все предыдущие стадии тоже должны быть приостановлены, чтобы ни одна из команд не пропала. Конвейерный регистр, находящийся сразу после приостановленной стадии, должен быть очищен (сброшен), чтобы «мусор» не распространялся дальше. Приостановки ухудшают производительность конвейера, поэтому должны использоваться только при необходимости.

На **Рис. 2.54** показан модифицированный конвейерный процессор, в который добавлены приостановки для команд, зависимых от `LDR`. Блок разрешения конфликтов смотрит, какая команда находится на стадии Execute. Если это `LDR`, а номер регистра-приемника (*WA3E*) совпадает с номером одного из регистров-источников команды, находящейся на стадии Decode (*RA1D* или *RA2D*), то стадия Decode должна быть приостановлена, до тех пор пока операнд-источник не будет готов.

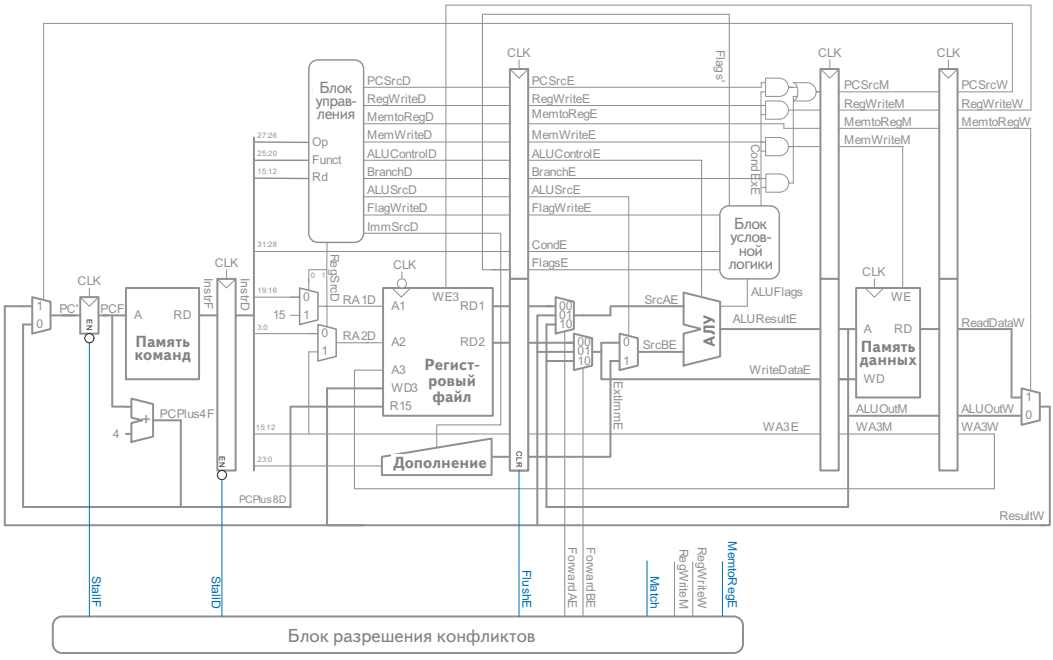


Рис. 2.54. Разрешение конфликта данных в конвейере из-за команды `LDR` при помощи приостановок

Для поддержки приостановки нужно добавить вход разрешения работы (*EN*) в конвейерные регистры Fetch и Decode, а также вход синхронной очистки (*CLR*) в конвейерный регистр Execute. Когда возникает необходимость приостановить конвейер из-за команды `LDR`, устанавливаются сигналы *StallD* и *StallF*, запрещающие конвейерным регистрам перед стадиями Decode и Fetch изменять находящееся в них значение. Также устанавливается сигнал *FlushE*, очищающий содержимое кон-

вейерного регистра перед стадией Execute, что приводит к появлению пузырька.

Для команды LDR всегда устанавливается сигнал *MemtoReg*. Таким образом, логика формирования сигналов *приостановки* и *очистки* выглядит так:

```
Match_12D_E = (RA1D == WA3E) + (RA2D == WA3E)
LDRstall = Match_12D_E • MemtoRegE
StallF = StallD = FlushE = LDRstall
```

Разрешение конфликтов управления

Выполнение команды B приводит к конфликту управления: конвейерный процессор не знает, какую команду выбрать следующей, поскольку в этот момент еще не ясно, нужно ли будет выполнить переход или нет. Запись в регистр R15 (PC) приводит к аналогичному конфликту.

Один из способов разрешить этот конфликт – приостановить конвейер до тех пор, пока не будет принято решение о переходе (т. е. до тех пор, пока не будет вычислен сигнал *PCSrcW*). Поскольку решение принимается на стадии Writeback, конвейер придется приостанавливать на четыре такта при каждом переходе. Такое решение повлекло бы весьма плачевные последствия для производительности системы.

Есть и альтернативный способ – предсказать, будет выполнен переход или нет, и, основываясь на этом, начать выполнять команды. Как только условие перехода будет вычислено, процессор может прервать выполнение команд, если предсказание было неверным. В рассматриваемом конвейере (**Рис. 2.54**) процессор предсказывает, что переход не произойдет, и просто продолжает выполнять команды по порядку, пока установленный сигнал *PCSrcW* не покажет, что следующее значение счетчика команд нужно брать из *ResultW*. Если окажется, что переход должен был быть выполнен, то конвейер должен быть *очищен* (flushed) от четырех команд, следующих за командой перехода, путем очистки соответствующих конвейерных регистров. Зря потраченные в этом случае такты называются *штрафом за неправильное предсказание перехода* (branch misprediction penalty).

На **Рис. 2.55** показано, что происходит в конвейере, если выполнен переход с адреса 0x20 по адресу 0x64. Запись в счетчик команд не производится до пятого такта; к этому моменту процессор уже выбрал команды AND, ORR и обе команды SUB из ячеек с адресами 0x24, 0x28, 0x2C и 0x30 соответственно. Конвейер должен быть очищен от этих команд, после чего на шестом такте будет выбрана команда ADD по адресу 0x64. Мы добились небольшого улучшения, однако очистка такого большого количества команд в случае выполненного перехода все еще существенно ухудшает производительность.

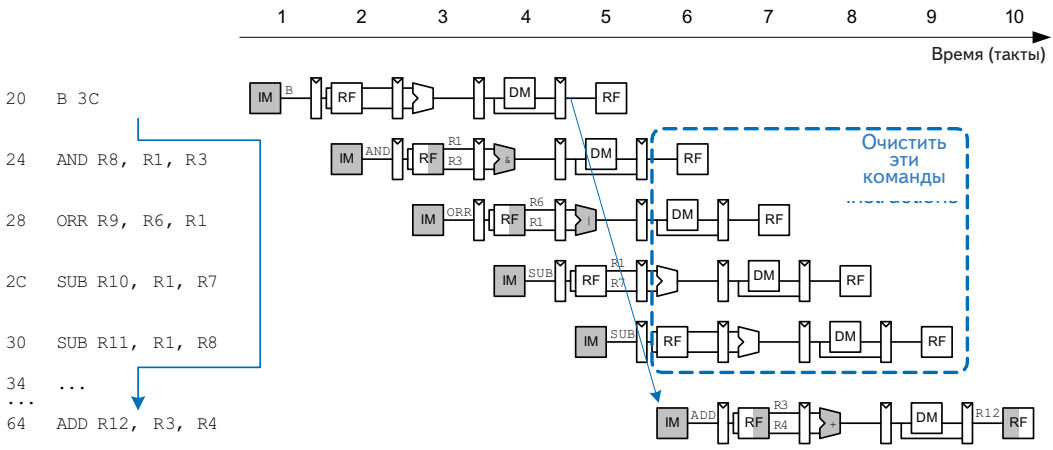


Рис. 2.55. Абстрактная диаграмма конвейера, иллюстрирующая очистку в случае перехода

Мы могли бы уменьшить штраф за неправильное предсказание перехода, если бы сумели принять решение о переходе пораньше. Отметим, что принять это решение можно на стадии Execute, когда был вычислен конечный адрес и стало известным значение *CondEx*. На **Рис. 2.56** показано, как работает конвейер в случае раннего предсказания перехода на третьем такте. На четвертом такте команды `AND` и `ORR` очищаются, и выбирается команда `ADD`. Теперь штраф за неправильное предсказание перехода сократился с четырех команд до двух.

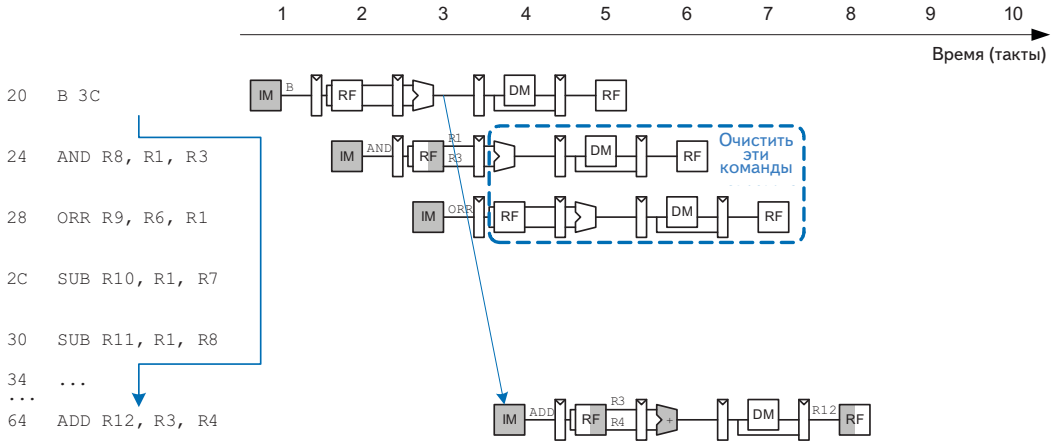


Рис. 2.56. Абстрактная диаграмма конвейера, иллюстрирующая раннее принятие решения о переходе

На **Рис. 2.57** показан модифицированный конвейерный процессор, который принимает решение о переходе и разрешает конфликты управ-

ления раньше. Перед регистром PC поставлен мультиплексор, который выбирает адрес перехода на основе *ALUResultE*. Управляющий этим мультиплексором сигнал *BranchTakenE* устанавливается на тех ветвях, для которые выполнено условие перехода. Сигнал *PCSrcW* теперь устанавливается только для записи в PC, которая по-прежнему производится на стадии Writeback.

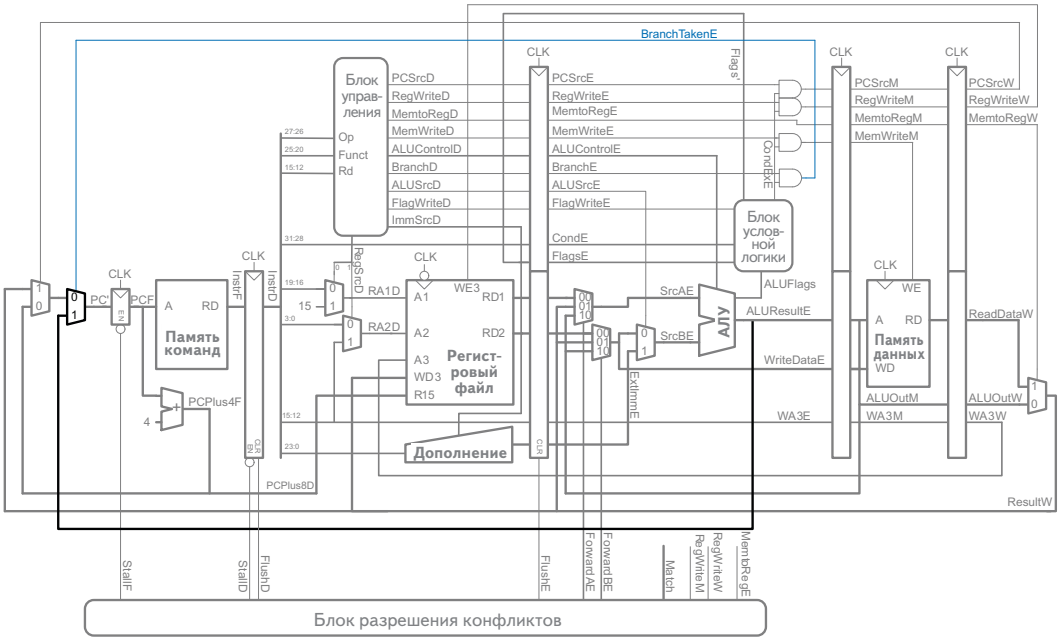


Рис. 2.57. Конвейерный процессор с устранением конфликта управления из-за команды перехода

Наконец, мы должны продумать сигналы приостановки и очистки для управления переходами и записью в PC. При проектировании этой части конвейерного процессора часто допускают ошибки, потому что условия довольно сложные. Если переход производится, то следующие две команды должны быть вычищены из конвейерных регистров перед стадиями Decode и Execute. Если в конвейере есть операция записи в PC, то конвейер следует приостановить до тех, пор пока запись не завершится. Это делается путем приостановки стадии Fetch. Напомним, что приостановка одной стадии влечет за собой необходимость приостановки следующей, чтобы предотвратить повторное выполнение команды. Логика обработки этих случаев описана ниже. Сигнал *PCWrPending* устанавливается на время, пока производится запись в PC (на стадии Decode, Execute или Memory). На протяжении этого времени стадия Fetch при-

остановлена, а стадия Decode очищается. Когда операция записи в PC достигнет стадии Writeback (сигнал *PCSrcW* установлен), сигнал *StallF* сбрасывается, чтобы разрешить запись, но сигнал *FlushD* все еще установлен, чтобы предотвратить выполнение нежелательной команды на стадии Fetch.

Чтобы не загромождать рисунки, сигналы *PCSrcD*, *PCSrcE*, *PCSrcM* и *BranchTakenE*, соединяющие блок обнаружения конфликтов с трактом данных, на **Рис. 2.57** и **2.58** не показаны.

```
PCWrPendingF = PCSrcD + PCSrcE + PCSrcM;
StallD = LDRstall;
StallF = LDRstall + PCWrPendingF;
FlushE = LDRstall + BranchTakenE;
FlushD = PCWrPendingF + PCSrcW + BranchTakenE;
```

Команды перехода встречаются очень часто, и даже два такта штрафа за неправильное предсказание перехода плохо сказываются на производительности. Приложив еще немного усилий, мы могли бы сократить штраф до одного такта для многих переходов. Конечный адрес следует вычислять на стадии Decode в виде $PCBranchD = PCPlus8D + ExtImmD$. Сигнал *BranchTakenD* также следует вычислять на стадии Decode на основе флагов *ALUFlagsE*, установленных при выполнении предыдущей команды. Это может увеличить длительность такта процессора, если флаги поступают поздно. Оставляем реализацию этих изменений в качестве упражнения для читателей (см. **упражнение 2.36**).

Подводя итоги

Конфликты при чтении данных после записи (RAW data hazards) возникают, когда одна из команд зависит от результата другой команды, еще не записанного в регистровый файл. Такие конфликты можно разрешить при помощи пробрасывания, если результат вычислен достаточно рано; в противном случае потребуется приостановка конвейера, до тех пор пока результат не будет готов. Конфликты управления возникают, когда нужно выбрать из памяти команду, а решение о том, какую именно, еще не принято. Эти конфликты разрешаются путем предсказания того, какая именно команда должна быть выбрана, и очисткой конвейера от ошибочно выбранных команд в случае, если предсказание не сбылось, либо приостановкой конвейера до момента принятия решения. Количество команд, от которых придется очистить конвейер в случае неправильно предсказанного перехода, минимизируется путем перемещения логики вычисления условия переходов ближе к началу конвейера. Как вы могли заметить, при проектировании конвейерных процессоров нужно понимать, как различные команды могут взаимодействовать между собой, а также заранее обнаруживать все возможные конфликты. На **Рис. 2.58** показан окончательный конвейерный процессор, способный разрешить все конфликты.

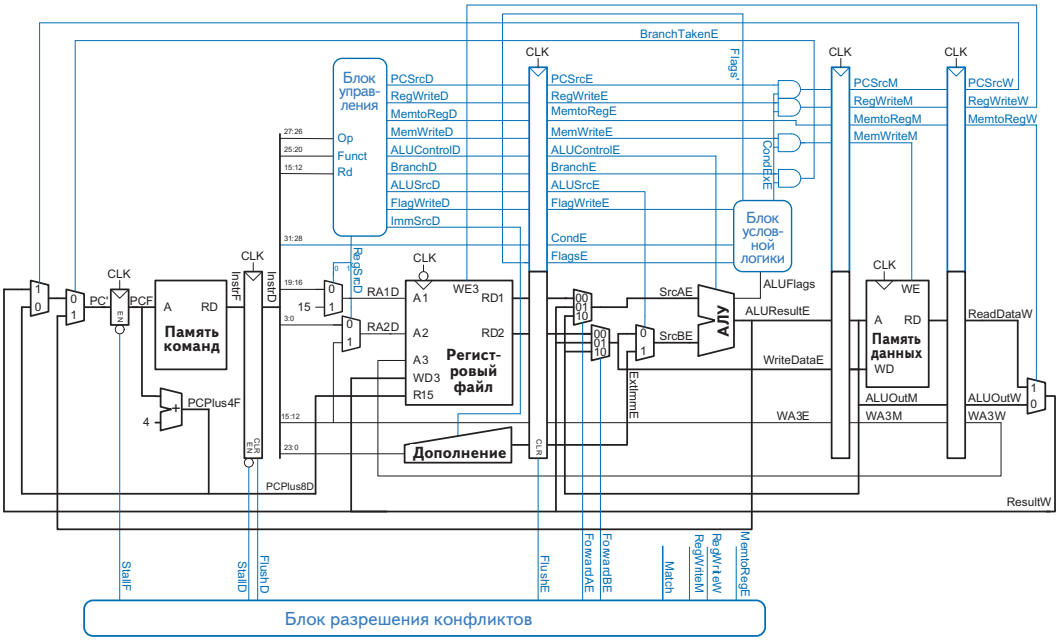


Рис. 2.58. Конвейерный процессор, разрешающий все конфликты

2.5.4. Анализ производительности

В идеальном конвейерном процессоре количество тактов на команду должно быть равно единице (CPI = 1), потому что на каждом такте процессор начинает выполнять новую команду. Однако приостановки и очистки конвейера приводят к тому, что некоторые такты выпадают, так что в реальной жизни CPI немного выше и зависит от выполняемой программы.

Пример 2.7. CPI КОНВЕЙЕРНОГО ПРОЦЕССОРА

Эталонный тест SPECINT2000, рассмотренный в **примере 2.5**, содержит примерно 25% команд загрузки, 10% команд сохранения, 13% команд перехода и 52% команд обработки данных. Предположим, что в 40% случаев за командой загрузки сразу идет команда, в которой ее результат используется, что приводит к приостановке конвейера. Также предположим, что 50% переходов производятся (т. е. предсказаны неверно), что приводит к очистке конвейера. Прочими конфликтами пренебрежем. Требуется вычислить среднее число тактов на команду (CPI) в конвейерном процессоре.

Решение. Среднее CPI можно вычислить, перемножив число тактов, требуемое для выполнения команды каждого типа, на относительное количество соответствующих команд и просуммировав полученные значения. Команды загрузки данных выполняются за один такт, если нет зависимых от них команд, и за два

такта, если конвейер должен быть приостановлен для разрешения конфликта, так что их CPI равно (0.6)(1) + (0.4)(2) = 1.4. Команды перехода выполняются за один такт, если переход предсказан корректно, и за три такта в противном случае, так что их CPI равно (0.5)(1) + (0.5)(3) = 2.0. CPI всех остальных команд равно единице. Таким образом, для этого тестового набора среднее CPI = (0.25)(1.4) + (0.1)(1) + (0.13)(2.0) + (0.52)(1) = 1.23.

Длительность такта мы можем определить, рассмотрев критический путь на каждой из пяти стадий, показанных на **Рис. 2.58**. Если принять во внимание, что запись в регистровый файл производится на стадии Writeback в первой половине такта, а чтение – на стадии Decode во второй половине такта, то получится, что длительность такта на стадиях Decode и Writeback равна удвоенному времени, необходимому для выполнения всех действий в половине такта.

$$T_{c3} = max \begin{bmatrix} t_{pcq} + t_{mem} + t_{setup} \\ 2(t_{RFread} + t_{setup}) \\ t_{pcq} + 2t_{mux} + t_{ALU} + t_{setup} \\ t_{pcq} + t_{mem} + t_{setup} \\ 2(t_{pcq} + t_{mux} + t_{RFsetup}) \end{bmatrix} \begin{matrix} Fetch \\ Decode \\ Execute \\ Memory \\ Writeback \end{matrix} \quad (2.5)$$

Пример 2.8. СРАВНЕНИЕ ПРОИЗВОДИТЕЛЬНОСТИ ПРОЦЕССОРОВ

Бен Битдидл хочет сравнить производительность конвейерного процессора с производительностью однотактного и многотактного процессоров (см. **пример 2.6**). Величины задержек элементов были приведены в **Табл. 2.5**. Помогите Бену сравнить время выполнения 100 миллиардов команд из теста SPECINT2000 на каждом из этих процессоров.

Решение. Согласно формуле **(2.5)**, длительность такта конвейерного процессора равна T_{c3} = max[40 + 200 + 50, 2(100 + 50), 40 + 2(25) + 120 + 50, 40 + 200 + 50, 2(40 + 25 + 60)] = 300 пс. Согласно формуле **(2.1)**, общее время выполнения равно T_3 = (100 × 10^9 команд)(1.23 такта/команду)(300 × 10^{-12} с/такт) = 36.9 с. Время выполнения для однотактного процессора было равно 84 с, а для многотактного – 140 с.

Таким образом, конвейерный процессор значительно быстрее прочих. Однако о пятикратном преимуществе над однотактным процессором, которое мы надеялись получить из-за пятистадийного конвейера, не может быть и речи. Конфликты в конвейере вносят свою лепту в увеличение CPI. Но важнее то, что накладные расходы на организацию конвейера (время задержки clk-to-Q и время предустановки конвейерных регистров) теперь добавляются к каждой стадии, а не к тракту данных в целом. Эти наклад-

ные расходы ограничивают преимущества конвейерной обработки. По требованиям к оборудованию конвейерный процессор близок к однотактному, но ему дополнительно требуется восемь 32-битовых конвейерных регистров, а также мультиплексоры, конвейерные регистры меньшей разрядности и логические элементы для разрешения конфликтов.

2.6. Представление на языке HDL

В этом разделе приведен HDL-код однотактного процессора, который поддерживает все рассмотренные в данной главе команды. Этот код может служить примером хорошего стиля при описании систем умеренной сложности. Написание HDL-кода для многотактного и конвейерного процессоров мы оставляем читателю в качестве **упражнений 2.25** и **2.40**.

В этом разделе мы будем считать, что память команд и память данных отделены от тракта данных и подсоединены к нему с помощью шин адреса и данных. На практике большинство процессоров выбирают команды и данные из внешних кэшей. Однако для поддержки пулов литералов развитый процессор должен уметь читать данные из памяти команд. В **главе 3** мы вернемся к подсистемам памяти и рассмотрим в том числе взаимодействие кэшей с основной памятью.

Процессор состоит из тракта данных и устройства управления. Устройство управления, в свою очередь, состоит из декодера и блока условной логики. На **Рис. 2.59** показана диаграмма однотактного процессора, соединенного с внешней памятью.

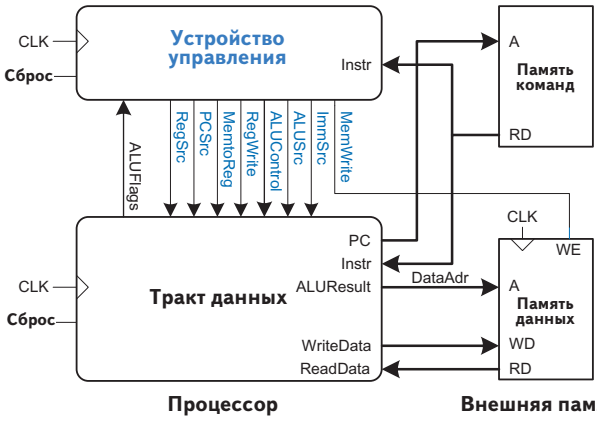


Рис. 2.59. Подключение однотактного процессора к внешней памяти

HDL-код разбит на несколько частей, каждая из которых описана в отдельном разделе. В **разделе 2.6.1** содержится описание тракта данных и устройства управления. В **разделе 2.6.2** представлены универсальные строительные блоки, в частности регистры и мультиплексоры, которые

используются в любой микроархитектуре. В **разделе 2.6.3** приведен код тестового окружения и внешней памяти. HDL-код доступен в виде файлов для скачивания на веб-сайте книги (см. **предисловие**).

2.6.1. Однотактный процессор

Основные модули однотактного процессора приведены ниже.

HDL-пример 2.1. ОДНОТАКТНЫЙ ПРОЦЕССОР

SystemVerilog

```
module arm(input  logic        clk, reset,
           output logic [31:0] PC,
           input  logic [31:0] Instr,
           output logic        MemWrite,
           output logic [31:0] ALUResult, WriteData,
           input  logic [31:0] ReadData);

logic [3:0] ALUFlags;
logic       RegWrite,
            ALUSrc, MemtoReg, PCSrc;
logic [1:0] RegSrc, ImmSrc, ALUControl;

controller c(clk, reset, Instr[31:12], ALUFlags,
             RegSrc, RegWrite, ImmSrc,
             ALUSrc, ALUControl,
             MemWrite, MemtoReg, PCSrc);
datapath dp(clk, reset,
            RegSrc, RegWrite, ImmSrc,
            ALUSrc, ALUControl,
            MemtoReg, PCSrc,
            ALUFlags, PC, Instr,
            ALUResult, WriteData, ReadData);
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity arm is -- однотактный процессор
  port(clk, reset: in STD_LOGIC;
       PC:         out STD_LOGIC_VECTOR(31 downto 0);
       Instr:       in STD_LOGIC_VECTOR(31 downto 0);
       MemWrite:    out STD_LOGIC;
       ALUResult,
       WriteData:  out STD_LOGIC_VECTOR(31 downto 0);
       ReadData:   in STD_LOGIC_VECTOR(31 downto 0));
end;

architecture struct of arm is
  component controller
  port(clk, reset: in STD_LOGIC;
       Instr:      in STD_LOGIC_VECTOR(31 downto 12);
       ALUFlags:    in STD_LOGIC_VECTOR(3 downto 0);
       RegSrc:     out STD_LOGIC_VECTOR(1 downto 0);
       RegWrite:   out STD_LOGIC;
       ImmSrc:     out STD_LOGIC_VECTOR(1 downto 0);
       ALUSrc:     out STD_LOGIC;
       ALUControl: out STD_LOGIC_VECTOR(1 downto 0);
       MemWrite:   out STD_LOGIC;
       MemtoReg:   out STD_LOGIC;
       PCSrc:      out STD_LOGIC);
  end component;
  component datapath
  port(clk, reset: in STD_LOGIC;
      RegSrc:     in STD_LOGIC_VECTOR(1 downto 0);
      RegWrite:   in STD_LOGIC;
      ImmSrc:     in STD_LOGIC_VECTOR(1 downto 0);
      ALUSrc:     in STD_LOGIC;
      ALUControl: in STD_LOGIC_VECTOR(1 downto 0);
      MemtoReg:   in STD_LOGIC;
      PCSrc:      in STD_LOGIC;
      ALUFlags:   out STD_LOGIC_VECTOR(3 downto 0);
      PC:      buffer STD_LOGIC_VECTOR(31 downto 0);
      Instr:      in STD_LOGIC_VECTOR(31 downto 0);
      ALUResult,
     WriteData:buffer STD_LOGIC_VECTOR(31 downto 0);
      ReadData:   in STD_LOGIC_VECTOR(31 downto 0));
end component;
signal RegWrite, ALUSrc, MemtoReg, PCSrc: STD_LOGIC;
signal RegSrc, ImmSrc, ALUControl: STD_LOGIC_VECTOR
                                   (1 downto 0);
signal ALUFlags: STD_LOGIC_VECTOR(3 downto 0);
begin
  cont: controller port map(clk, reset,
                            Instr(31 downto 12),
                           ALUFlags, RegSrc, RegWrite,
                           ImmSrc, ALUSrc, ALUControl,
```

```
                                MemWrite, MemtoReg, PCSrc);
  dp: datapath port map(clk, reset, RegSrc, RegWrite,
                        ImmSrc, ALUSrc, ALUControl,
                        MemtoReg, PCSrc, ALUFlags,
                        PC, Instr, ALUResult,
                        WriteData, ReadData);
end;
```

HDL-пример 2.2. УСТРОЙСТВО УПРАВЛЕНИЯ

SystemVerilog

```
module controller(input logic        clk, reset,
                  input logic [31:12] Instr,
                  input logic [3:0]   ALUFlags,
                  output logic [1:0]  RegSrc,
                  output logic        RegWrite,
                  output logic [1:0]  ImmSrc,
                  output logic        ALUSrc,
                 output logic [1:0]  ALUControl,
                  output logic        MemWrite,
                                      MemtoReg,
                  output logic        PCSrc);
  logic [1:0] FlagW;
  logic       PCS, RegW, MemW;

 decoder dec(Instr[27:26], Instr[25:20],
         Instr[15:12], FlagW, PCS, RegW, MemW,
         MemtoReg, ALUSrc, ImmSrc, RegSrc,
         ALUControl);
condlogic cl(clk, reset, Instr[31:28], ALUFlags,
             FlagW, PCS, RegW, MemW,
             PCSrc, RegWrite, MemWrite);
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity controller is -- однотактное у-во управления
  port(clk, reset: in STD_LOGIC;
       Instr:      in STD_LOGIC_VECTOR(31 downto 12);
       ALUFlags:   in STD_LOGIC_VECTOR(3 downto 0);
       RegSrc:     out STD_LOGIC_VECTOR(1 downto 0);
       RegWrite:   out STD_LOGIC;
       ImmSrc:     out STD_LOGIC_VECTOR(1 downto 0);
       ALUSrc:     out STD_LOGIC;
       ALUControl: out STD_LOGIC_VECTOR(1 downto 0);
       MemWrite:   out STD_LOGIC;
       MemtoReg:   out STD_LOGIC;
       PCSrc:      out STD_LOGIC);
end;

architecture struct of controller is
  component decoder
   port(Op:         in STD_LOGIC_VECTOR(1 downto 0);
   Funct:           in STD_LOGIC_VECTOR(5 downto 0);
   Rd:              in STD_LOGIC_VECTOR(3 downto 0);
   FlagW:           out STD_LOGIC_VECTOR(1 downto 0);
   PCS, RegW, MemW:  out STD_LOGIC;
   MemtoReg, ALUSrc: out STD_LOGIC;
   ImmSrc, RegSrc: out STD_LOGIC_VECTOR(1 downto 0);
   ALUControl:     out STD_LOGIC_VECTOR(1 downto 0));
 end component;
 component condlogic
  port(clk, reset: in STD_LOGIC;
   Cond:            in STD_LOGIC_VECTOR(3 downto 0);
   ALUFlags:        in STD_LOGIC_VECTOR(3 downto 0);
   FlagW:           in STD_LOGIC_VECTOR(1 downto 0);
   PCS, RegW, MemW: in STD_LOGIC;
   PCSrc, RegWrite: out STD_LOGIC;
   MemWrite:        out STD_LOGIC);
 end component;
 signal FlagW: STD_LOGIC_VECTOR(1 downto 0);
 signal PCS, RegW, MemW: STD_LOGIC;
begin
 dec: decoder port map(Instr(27 downto 26),
                       Instr(25 downto 20),
                       Instr(15 downto 12), FlagW,
                       PCS, RegW, MemW, MemtoReg,
                       ALUSrc, ImmSrc, RegSrc,
                       ALUControl);
 cl: condlogic port map(clk, reset,
                        Instr(31 downto 28),
                        ALUFlags, FlagW, PCS, RegW,
                        MemW, PCSrc, RegWrite,
                        MemWrite);
end;
```

HDL-пример 2.3. ДЕКОДЕР

SystemVerilog

```
module decoder(input logic  [1:0] Op,
               input logic  [5:0] Funct,
               input logic  [3:0] Rd,
               output logic [1:0] FlagW,
               output logic       PCS, RegW, MemW,
               output logic       MemtoReg, ALUSrc,
               output logic [1:0] ImmSrc, RegSrc,
                                  ALUControl);
  logic [9:0]  controls;
  logic        Branch, ALUOp;

  // Главный декодер
  always_comb
    casex(Op)
                // Обработка данных непосредственная
      2'b00: if (Funct[5]) controls = 10'b0000101001;
                 // Обработка данных регистровая
             else controls = 10'b0000001001;
                  // LDR
      2'b01: if (Funct[0]) controls = 10'b0001111000;
                // STR
             else controls = 10'b1001110100;
                  // B
      2'b10: controls = 10'b0110100010;
                // Не реализовано
      default: controls = 10'bx;
    endcase

  assign {RegSrc, ImmSrc, ALUSrc, MemtoReg,
     RegW, MemW, Branch, ALUOp} = controls;

  // Декодер АЛУ
  always_comb
  if (ALUOp) begin          // какая команда ОД?
     case(Funct[4:1])
         4'b0100: ALUControl = 2'b00; // ADD
         4'b0010: ALUControl = 2'b01; // SUB
         4'b0000: ALUControl = 2'b10; // AND
         4'b1100: ALUControl = 2'b11; // ORR
        default: ALUControl = 2'bx; // не реализовано
     endcase

     // обновить флаги, если bit S поднят
     // (C & V только для арифм. операций)
     FlagW[1] = Funct[0];
     FlagW[0] = Funct[0] &
       (ALUControl == 2'b00 | ALUControl == 2'b01);
  end else begin
     ALUControl = 2'b00; // для не ОД-команд
     FlagW = 2'b00;      // не обновлять флаги
  end

  // Логика работы с PC
  assign PCS = ((Rd == 4'b1111) & RegW) | Branch;
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity decoder is -- декодер
 port(Op:          in STD_LOGIC_VECTOR(1 downto 0);
      Funct:       in STD_LOGIC_VECTOR(5 downto 0);
      Rd:          in STD_LOGIC_VECTOR(3 downto 0);
      FlagW:       out STD_LOGIC_VECTOR(1 downto 0);
      PCS, RegW, MemW:  out STD_LOGIC;
      MemtoReg, ALUSrc: out STD_LOGIC;
      ImmSrc,
      RegSrc:      out STD_LOGIC_VECTOR(1 downto 0);
      ALUControl:  out STD_LOGIC_VECTOR(1 downto 0));
end;
architecture behave of decoder is
 signal controls: STD_LOGIC_VECTOR(9 downto 0);
 signal ALUOp, Branch: STD_LOGIC;
 signal op2: STD_LOGIC_VECTOR(3 downto 0);
begin
 op2 <= (Op, Funct(5), Funct(0));
 process(all) begin -- главный декодер
   case? (op2) is
     when "000-" => controls <= "0000001001";
     when "001-" => controls <= "0000101001";
     when "01-0" => controls <= "1001110100";
     when "01-1" => controls <= "0001111000";
     when "10--" => controls <= "0110100010";
     when others => controls <= "----------";
   end case?;
 end process;

(RegSrc, ImmSrc, ALUSrc, MemtoReg, RegW, MemW,
 Branch, ALUOp) <= controls;

 process(all) begin -- декодер АЛУ
 if (ALUOp) then
    case Funct(4 downto 1) is
      when "0100" => ALUControl <= "00"; -- ADD
      when "0010" => ALUControl <= "01"; -- SUB
      when "0000" => ALUControl <= "10"; -- AND
      when "1100" => ALUControl <= "11"; -- ORR
      when others => ALUControl <= "--"; -- не реализовано
    end case;
    FlagW(1) <= Funct(0);
    FlagW(0) <= Funct(0) and (not ALUControl(1));
  else
    ALUControl <= "00";
    FlagW <= "00";
  end if;
 end process;

 PCS <= ((and Rd) and RegW) or Branch;
end;
```

HDL-пример 2.4. БЛОК УСЛОВНОЙ ЛОГИКИ

SystemVerilog

```
module condlogic(input  logic       clk, reset,
                 input  logic [3:0] Cond,
                 input  logic [3:0] ALUFlags,
                 input  logic [1:0] FlagW,
                 input  logic       PCS, RegW, MemW,
                 output logic       PCSrc, RegWrite,
                                    MemWrite);

  logic [1:0] FlagWrite;
  logic [3:0] Flags;
  logic       CondEx;

  flopenr #(2)flagreg1(clk, reset, FlagWrite[1],
                       ALUFlags[3:2], Flags[3:2]);
  flopenr #(2)flagreg0(clk, reset, FlagWrite[0],
                       ALUFlags[1:0], Flags[1:0]);

  // управляющие сигналы записи условные
  condcheck cc(Cond, Flags, CondEx);
  assign FlagWrite = FlagW & {2{CondEx}};
  assign RegWrite  = RegW & CondEx;
  assign MemWrite  = MemW & CondEx;
  assign PCSrc     = PCS & CondEx;
endmodule

module condcheck(input  logic [3:0] Cond,
                 input  logic [3:0] Flags,
                 output logic       CondEx);

  logic neg, zero, carry, overflow, ge;

  assign {neg, zero, carry, overflow} = Flags;
  assign ge = (neg == overflow);

  always_comb
    case(Cond)
      4'b0000: CondEx = zero;             // EQ
      4'b0001: CondEx = ~zero;            // NE
      4'b0010: CondEx = carry;            // CS
      4'b0011: CondEx = ~carry;           // CC
      4'b0100: CondEx = neg;              // MI
      4'b0101: CondEx = ~neg;             // PL
      4'b0110: CondEx = overflow;         // VS
      4'b0111: CondEx = ~overflow;        // VC
      4'b1000: CondEx = carry & ~zero;    // HI
      4'b1001: CondEx = ~(carry & ~zero); // LS
      4'b1010: CondEx = ge;               // GE
      4'b1011: CondEx = ~ge;              // LT
      4'b1100: CondEx = ~zero & ge;       // GT
      4'b1101: CondEx = ~(~zero & ge);    // LE
      4'b1110: CondEx = 1'b1;             // всегда
      default: CondEx = 1'bx;       // не определено
    endcase
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity condlogic is -- Conditional logic
 port(clk, reset: in STD_LOGIC;
      Cond:       in STD_LOGIC_VECTOR(3 downto 0);
      ALUFlags:   in STD_LOGIC_VECTOR(3 downto 0);
      FlagW:      in STD_LOGIC_VECTOR(1 downto 0);
      PCS, RegW, MemW: in STD_LOGIC;
      PCSrc, RegWrite: out STD_LOGIC;
      MemWrite:        out STD_LOGIC);
end;

architecture behave of condlogic is
 component condcheck
 port(Cond: in STD_LOGIC_VECTOR(3 downto 0);
      Flags: in STD_LOGIC_VECTOR(3 downto 0);
      CondEx: out STD_LOGIC);
 end component;
 component flopenr generic(width: integer);
  port(clk, reset, en: in STD_LOGIC;
       d:   in STD_LOGIC_VECTOR (width-1 downto 0);
       q:   out STD_LOGIC_VECTOR (width-1 downto 0));
 end component;
 signal FlagWrite: STD_LOGIC_VECTOR(1 downto 0);
 signal Flags:     STD_LOGIC_VECTOR(3 downto 0);
 signal CondEx:    STD_LOGIC;
begin
  flagreg1: flopenr generic map(2)
   port map(clk, reset, FlagWrite(1),
            ALUFlags(3 downto 2), Flags(3 downto 2));
  flagreg0: flopenr generic map(2)
   port map(clk, reset, FlagWrite(0),
            ALUFlags(1 downto 0), Flags(1 downto 0));
  cc: condcheck port map(Cond, Flags, CondEx);

  FlagWrite <= FlagW and (CondEx, CondEx);
  RegWrite  <= RegW and CondEx;
  MemWrite  <= MemW and CondEx;
  PCSrc     <= PCS and CondEx;
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity condcheck is
 port(Cond: in STD_LOGIC_VECTOR(3 downto 0);
   Flags:    in STD_LOGIC_VECTOR(3 downto 0);
   CondEx:   out STD_LOGIC);
end;

architecture behave of condcheck is
 signal neg, zero, carry, overflow, ge: STD_LOGIC;
begin
 (neg, zero, carry, overflow) <= Flags;
 ge <= (neg xnor overflow);

 process(all) begin -- проверка условия
  case Cond is
    when "0000" => CondEx <= zero;
    when "0001" => CondEx <= not zero;
    when "1101" => CondEx <= not ((not zero) and    ge);
```

```
    when "0010" => CondEx <= carry;
    when "0011" => CondEx <= not carry;
    when "0100" => CondEx <= neg;
    when "0101" => CondEx <= not neg;
    when "0110" => CondEx <= overflow;
    when "0111" => CondEx <= not overflow;
    when "1000" => CondEx <= carry and (not zero);
    when "1001" => CondEx <= not(carry and (not   zero));
    when "1010" => CondEx <= ge;
    when "1011" => CondEx <= not ge;
    when "1100" => CondEx <= (not zero) and ge;
    when "1110" => CondEx <= '1';
    when others => CondEx <= '-';
    end case;
   end process;
end;
```

HDL-пример 2.5. ТРАКТ ДАННЫХ

SystemVerilog

```
module datapath(input  logic        clk, reset,
                input  logic [1:0]  RegSrc,
                input  logic        RegWrite,
                input  logic [1:0]  ImmSrc,
                input  logic        ALUSrc,
                input  logic [1:0]  ALUControl,
                input  logic        MemtoReg,
                input  logic        PCSrc,
                output logic [3:0]  ALUFlags,
                output logic [31:0] PC,
                input  logic [31:0] Instr,
                output logic [31:0] ALUResult,
                                    WriteData,
                input  logic [31:0] ReadData);

 logic [31:0] PCNext, PCPlus4, PCPlus8;
 logic [31:0] ExtImm, SrcA, SrcB, Result;
 logic [3:0] RA1, RA2;

 // логика работы с PC
 mux2 #(32) pcmux(PCPlus4, Result, PCSrc, PCNext);
 flopr #(32) pcreg(clk, reset, PCNext, PC);
 adder #(32) pcadd1(PC, 32'b100, PCPlus4);
 adder #(32) pcadd2(PCPlus4, 32'b100, PCPlus8);

 // логика работы с регистровым файлом
 mux2 #(4) ra1mux(Instr[19:16], 4'b1111, RegSrc[0],
                  RA1);
 mux2 #(4)  ra2mux(Instr[3:0], Instr[15:12],
                  RegSrc[1], RA2);
 regfile    rf(clk, RegWrite, RA1, RA2,
               Instr[15:12], Result, PCPlus8,
               SrcA, WriteData);
 mux2 #(32) resmux(ALUResult, ReadData, MemtoReg,
                  Result);
 extend     ext(Instr[23:0], ImmSrc, ExtImm);

 // логика работы с АЛУ
 mux2 #(32) srcbmux(WriteData, ExtImm, ALUSrc, SrcB);
 alu        alu(SrcA, SrcB, ALUControl, ALUResult,
                ALUFlags);
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity datapath is
 port(clk, reset: in  STD_LOGIC;
      RegSrc:     in  STD_LOGIC_VECTOR(1 downto 0);
      RegWrite:   in  STD_LOGIC;
      ImmSrc:     in  STD_LOGIC_VECTOR(1 downto 0);
      ALUSrc:     in  STD_LOGIC;
      ALUControl: in  STD_LOGIC_VECTOR(1 downto 0);
      MemtoReg:   in  STD_LOGIC;
      PCSrc:      in  STD_LOGIC;
      ALUFlags:   out STD_LOGIC_VECTOR(3 downto 0);
      PC:       buffer STD_LOGIC_VECTOR(31 downto 0);
      Instr:      in  STD_LOGIC_VECTOR(31 downto 0);
      ALUResult, WriteData:buffer STD_LOGIC_VECTOR(31
downto 0);
      ReadData:   in  STD_LOGIC_VECTOR(31 downto 0));
end;

architecture struct of datapath is
 component alu
 port(a, b:       in  STD_LOGIC_VECTOR(31 downto 0);
      ALUControl: in  STD_LOGIC_VECTOR(1 downto 0);
      Result: buffer STD_LOGIC_VECTOR(31 downto 0);
      ALUFlags:   out STD_LOGIC_VECTOR(3 downto 0));
 end component;
 component regfile
 port(clk:           in STD_LOGIC;
      we3:           in STD_LOGIC;
      ra1, ra2, wa3: in STD_LOGIC_VECTOR(3 downto 0);
      wd3, r15:      in STD_LOGIC_VECTOR(31 downto 0);
      rd1, rd2:   out STD_LOGIC_VECTOR(31 downto 0));
 end component;
 component adder
 port(a, b: in  STD_LOGIC_VECTOR(31 downto 0);
      y:    out STD_LOGIC_VECTOR(31 downto 0));
 end component;
 component extend
 port(Instr:  in  STD_LOGIC_VECTOR(23 downto 0);
      ImmSrc: in  STD_LOGIC_VECTOR(1 downto 0);
      ExtImm: out STD_LOGIC_VECTOR(31 downto 0));
 end component;
 component flopr generic(width: integer);
 port(clk, reset: in STD_LOGIC;
```

```
      d:     in  STD_LOGIC_VECTOR(width-1 downto 0);
      q:     out STD_LOGIC_VECTOR(width-1 downto 0));
 end component;
 component mux2 generic(width: integer);
 port(d0, d1: in STD_LOGIC_VECTOR(width-1 downto 0);
      s:      in STD_LOGIC;
      y:     out STD_LOGIC_VECTOR(width-1 downto 0));
 end component;
 signal PCNext, PCPlus4,
       PCPlus8:     STD_LOGIC_VECTOR(31 downto 0);
 signal ExtImm,
        Result:     STD_LOGIC_VECTOR(31 downto 0);
 signal SrcA, SrcB: STD_LOGIC_VECTOR(31 downto 0);
 signal RA1, RA2:   STD_LOGIC_VECTOR(3 downto 0);
begin
 -- логика работы с PC
 pcmux: mux2 generic map(32)
        port map(PCPlus4, Result, PCSrc, PCNext);
 pcreg: flopr generic map(32) port map(clk, reset,
           PCNext, PC);
 pcadd1: adder port map(PC, X"00000004", PCPlus4);
 pcadd2: adder port map(PCPlus4, X"00000004",
                        PCPlus8);

 -- логика работы с регистровым файлом
 ralmux: mux2 generic map (4)
   port map(Instr(19 downto 16), "1111", RegSrc(0),
            RA1);
 ra2mux: mux2 generic map (4)
   port map(Instr(3 downto 0),
            Instr(15 downto 12), RegSrc(1), RA2);
 rf: regfile
   port map(clk, RegWrite, RA1, RA2,
            Instr(15 downto 12), Result,
            PCPlus8, SrcA, WriteData);
 resmux: mux2 generic map(32)
   port map(ALUResult, ReadData, MemtoReg, Result);
 ext: extend
    port map(Instr(23 downto 0), ImmSrc, ExtImm);

 -- логика работы с АЛУ
 srcbmux: mux2 generic map(32)
   port map(WriteData, ExtImm, ALUSrc, SrcB);
 i_alu: alu
   port map(SrcA, SrcB, ALUControl, ALUResult,
            ALUFlags);
end;
```

2.6.2. Универсальные строительные блоки

Этот раздел содержит универсальные строительные блоки, которые могут быть полезны в любой цифровой системе: регистровый файл, триггеры и мультиплексор 2:1. Написание HDL-кода для АЛУ оставлено читателю в качестве **упражнений 5.11** и **5.12** (книга 1).

HDL-пример 2.6. РЕГИСТРОВЫЙ ФАЙЛ

SystemVerilog

```
module regfile(input  logic        clk,
               input  logic        we3,
               input  logic [3:0]  ra1, ra2, wa3,
               input  logic [31:0] wd3, r15,
               output logic [31:0] rd1, rd2);

 logic [31:0] rf[14:0];

 // регистровый файл с 3 портами
 // два порта читаются комбинационно
 // в третий порт производится запись
 // по положительному фронту тактового сигнала
 // при чтении из регистра 15 читается PC + 8

 always_ff @(posedge clk)
   if (we3) rf[wa3] <= wd3;

 assign rd1 = (ra1 == 4'b1111) ? r15 : rf[ra1];
 assign rd2 = (ra2 == 4'b1111) ? r15 : rf[ra2];
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;
entity regfile is – трехпортовый регистровый файл
 port(clk:           in  STD_LOGIC;
      we3:           in  STD_LOGIC;
      ra1, ra2, wa3: in STD_LOGIC_VECTOR(3 downto 0);
      wd3, r15:      in STD_LOGIC_VECTOR(31 downto 0);
      rd1, rd2:      out STD_LOGIC_VECTOR(31 downto 0));
end;

architecture behave of regfile is
  type ramtype is array (31 downto 0) of
    STD_LOGIC_VECTOR(31 downto 0);
  signal mem: ramtype;
begin
  process(clk) begin
  if rising_edge(clk) then
    if we3 = '1' then mem(to_integer(wa3)) <= wd3;
    end if;
  end if;
  end process;
  process(all) begin
    if (to_integer(ra1) = 15) then rd1 <= r15;
    else rd1 <= mem(to_integer(ra1));
    end if;
    if (to_integer(ra2) = 15) then rd2 <= r15;
    else rd2 <= mem(to_integer(ra2));
    end if;
  end process;
end;
```

HDL-пример 2.7. СУММАТОР

SystemVerilog

```
module adder #(parameter WIDTH = 8)
              (input  logic [WIDTH-1:0] a, b,
               output logic [WIDTH-1:0] y);

  assign y = a + b;
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;
entity adder is -- adder
  port(a, b: in  STD_LOGIC_VECTOR(31 downto 0);
             y: out STD_LOGIC_VECTOR(31 downto 0));
end;

architecture behave of adder is
begin
  y <= a + b;
end;
```

HDL-пример 2.8. ДОПОЛНЕНИЕ НЕПОСРЕДСТВЕННОГО ОПЕРАНДА НУЛЯМИ

SystemVerilog

```
module extend(input  logic [23:0] Instr,
              input  logic [1:0]  ImmSrc,
              output logic [31:0] ExtImm);
  always_comb
    case(ImmSrc)
          // 8-битовый непосредственый без знака
      2'b00: ExtImm = {24'b0, Instr[7:0]};
          // 12-битовый непосредственый без знака
      2'b01: ExtImm = {20'b0, Instr[11:0]};
          // 24-битовый сдвинутый в доп. коде
     2'b10: ExtImm = {{6{Instr[23]}}, Instr[23:0],
                      2'b00};
      default: ExtImm = 32'bx; // не определено
  endcase
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity extend is
  port(Instr:  in  STD_LOGIC_VECTOR(23 downto 0);
       ImmSrc: in  STD_LOGIC_VECTOR(1 downto 0);
       ExtImm: out STD_LOGIC_VECTOR(31 downto 0));
  end;

architecture behave of extend is
begin
  process(all) begin
   case ImmSrc is
     when "00" => ExtImm <= (X"000000", Instr(7
downto 0));
      when "01" => ExtImm <= (X"00000", Instr(11
downto 0));
    when "10" => ExtImm <= (Instr(23), Instr(23),
                            Instr(23), Instr(23),
                            Instr(23), Instr(23),
                        Instr(23 downto 0), "00");
    when others => ExtImm <= X"--------";
   end case;
  end process;
end;
```

HDL-пример 2.9. ТРИГГЕР СО СБРОСОМ

SystemVerilog

```
module flopr #(parameter WIDTH = 8)
              (input  logic             clk, reset,
               input  logic [WIDTH-1:0] d,
               output logic [WIDTH-1:0] q);

  always_ff @(posedge clk, posedge reset)
    if (reset) q <= 0;
    else q <= d;
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity flopr is -- flip-flop with synchronous reset
 generic(width: integer);
 port(clk, reset: in STD_LOGIC;
        d: in  STD_LOGIC_VECTOR(width-1 downto 0);
       q: out STD_LOGIC_VECTOR(width-1 downto 0));
end;

architecture asynchronous of flopr is
begin
  process(clk, reset) begin
    if reset then q <= (others => '0');
    elsif rising_edge(clk) then
         q <= d;
    end if;
  end process;
end;
```

HDL-пример 2.10. ТРИГГЕР СО СБРОСОМ И СИГНАЛОМ РАЗРЕШЕНИЯ

SystemVerilog

```
module flopenr #(parameter WIDTH = 8)
                (input logic         clk, reset, en,
                  input logic [WIDTH-1:0] d,
                  output logic [WIDTH-1:0] q);

  always_ff @(posedge clk, posedge reset)
    if (reset) q <= 0;
    else if (en) q <= d;
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity flopenr is -- триггер с сигналом разрешения и
                  -- синхронным сбросом
  generic(width: integer);
  port(clk, reset, en: in STD_LOGIC;
       d:    in  STD_LOGIC_VECTOR(width-1 downto 0);
      q:   out STD_LOGIC_VECTOR(width-1 downto 0));
end;

architecture asynchronous of flopenr is
begin
  process(clk, reset) begin
    if reset then q <= (others => '0');
    elsif rising_edge(clk) then
      if en then
         q <= d;
      end if;
    end if;
  end process;
end;
```

HDL-пример 2.11. МУЛЬТИПЛЕКСОР 2:1

SystemVerilog

```
module mux2 #(parameter WIDTH = 8)
             (input  logic [WIDTH-1:0] d0, d1,
              input  logic             s,
              output logic [WIDTH-1:0] y);

  assign y = s ? d1 : d0;
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity mux2 is -- two-input multiplexer
  generic(width: integer);
  port(d0, d1: in STD_LOGIC_VECTOR(width-1 downto 0);
       s:      in STD_LOGIC;
       y:    out STD_LOGIC_VECTOR(width-1 downto 0));
end;

architecture behave of mux2 is
begin
  y <= d1 when s else d0;
end;
```

2.6.3. Тестовое окружение

Тестовое окружение загружает программу в память. Программа, представленная на **Рис. 2.60**, тестирует все команды процессора, выполняя такие вычисления, которые приводят к правильному результату только тогда, когда все команды работают правильно. В случае успешного выполнения программа запишет значение 7 по адресу 100; маловероятно,

что это произойдет при наличии аппаратных ошибок. Это пример ситуативного тестирования (ad hoc testing).

```
АДРЕС       ПРОГРАММА            ; КОММЕНТАРИИ                    БИНАРНЫЙ МАШИННЫЙ КОД                          16-НЫЙ КОД
00  MAIN    SUB R0, R15, R15     ; R0 = 0                         1110 000 0010 0 1111 0000 0000 0000 1111       E04F000F
04          ADD R2, R0, #5       ; R2 = 5                         1110 001 0100 0 0000 0010 0000 0000 0101       E2802005
08          ADD R3, R0, #12      ; R3 = 12                        1110 001 0100 0 0000 0011 0000 0000 1100       E280300C
0C          SUB R7, R3, #9       ; R7 = 3                         1110 001 0010 0 0011 0111 0000 0000 1001       E2437009
10          ORR R4, R7, R2       ; R4 = 3 OR 5 = 7                1110 000 1100 0 0111 0100 0000 0000 0010       E1874002
14          AND R5, R3, R4       ; R5 = 12 AND 7 = 4              1110 000 0000 0 0011 0101 0000 0000 0100       E0035004
18          ADD R5, R5, R4       ; R5 = 4 + 7 = 11                1110 000 0100 0 0101 0101 0000 0000 0100       E0855004
1C          SUBS R8, R5, R7      ; R8 = 11 - 3 = 8,установить флаги 1110 000 0010 1 0101 1000 0000 0000 0111    E0558007
20          BEQ END              ; не должна выполниться          0000 1010 0000  0000 0000 0000 0000 1100       0A00000C
24          SUBS R8, R3, R4      ; R8 = 12 - 7  = 5               1110 000 0010 1 0011 1000 0000 0000 0100       E0538004
28          BGE AROUND           ; должна выполниться             1010 1010 0000  0000 0000 0000 0000 0000       AA000000
2C          ADD R5, R0, #0       ; должна быть пропущена          1110 001 0100 0 0000 0101 0000 0000 0000       E2805000
30 AROUND   SUBS R8, R7, R2      ; R8 = 3 - 5 = -2,установить флаги 1110 000 0010 1 0111 1000 0000 0000 0010   E0578002
34          ADDLT R7, R5, #1     ; R7 = 11 + 1 = 12               1011 001 0100 0 0101 0111 0000 0000 0001       B2857001
38          SUB R7, R7, R2       ; R7 = 12 - 5 = 7                1110 000 0010 0 0111 0111 0000 0000 0010       E0477002
3C          STR R7, [R3, #84]    ; mem[12+84] = 7                 1110 010 1100 0 0011 0111 0000 0101 0100       E5837054
40          LDR R2, [R0, #96]    ; R2 = mem[96] = 7               1110 010 1100 1 0000 0010 0000 0110 0000       E5902060
44          ADD R15, R15, R0     ; PC = PC+8 (пропустить следующую) 1110 000 0100 0 1111 1111 0000 0000 0000    E08FF000
48          ADD R2, R0, #14      ; не должна выполниться          1110 001 0100 0 0000 0010 0000 0000 0001       E280200E
4C          B END                ; всегда выполняется             1110 1010 0000  0000 0000 0000 0000 0001       EA000001
50          ADD R2, R0, #13      ; не должна выполниться          1110 001 0100 0 0000 0010 0000 0000 0001       E280200D
54          ADD R2, R0, #10      ; не должна выполниться          1110 001 0100 0 0000 0010 0000 0000 0001       E280200A
58 END      STR R2, [R0, #100] ; mem[100] = 7                     1110 010 1100 0 0000 0010 0000 0101 0100       E5802064
```

Рис. 2.60. Ассемблерный и машинный код тестовой программы

Машинный код хранится в шестнадцатеричном файле memfile.dat, который загружается тестовым окружением в процессе симуляции. Файл состоит из машинного кода команд, по одной команде в каждой строке. В следующих примерах приведены HDL-код тестового окружения, модуля верхнего уровня процессора ARM и блоков внешней памяти. Каждый блок памяти содержит 64 слова.

HDL-пример 2.12. ТЕСТОВОЕ ОКРУЖЕНИЕ

SystemVerilog

```
module testbench();
  logic clk;
  logic reset;
  logic [31:0] WriteData, DataAdr;
  logic MemWrite;

  // определить тестируемое устройство
 top dut(clk, reset, WriteData, DataAdr, MemWrite);

  // инициализировать тест
  initial
  begin
    reset <= 1; # 22; reset <= 0;
  end

  // генерировать такты для последовательности тестов
  always
  begin
```

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;
entity testbench is
end;

architecture test of testbench is
 component top
  port(clk, reset:           in STD_LOGIC;
       WriteData, DatAadr: out STD_LOGIC_VECTOR(31
downto 0);
       MemWrite:             out STD_LOGIC);
 end component;
  signal  WriteData,  DataAdr:  STD_LOGIC_VECTOR(31
downto 0);
 signal clk, reset, MemWrite: STD_LOGIC;
begin
 -- определить тестируемое устройство
```

```
    clk <= 1; # 5; clk <= 0; # 5;
  end

  // проверить, что в конце программы по адресу
  // 0x64 записано 7
  always @(negedge clk)
  begin
    if(MemWrite) begin
      if(DataAdr === 100 & WriteData === 7) begin
        $display("Simulation succeeded");
        $stop;
      end else if (DataAdr !== 96) begin
        $display("Simulation failed");
        $stop;
      end
    end
  end
endmodule
```

```
dut: top port map(clk, reset, WriteData, DataAdr,
                  MemWrite);

-- генерировать такты с периодом 10 нс
process begin
  clk <= '1';
  wait for 5 ns;
  clk <= '0';
  wait for 5 ns;
end process;

-- генерировать сброс на первые два такта
process begin
  reset <= '1';
  wait for 22 ns;
  reset <= '0';
  wait;
end process;

-- проверить, что в конце программы по адресу
-- 0x64 записано 7
process (clk) begin
 if (clk'event and clk = '0'
              and MemWrite = '1') then
    if (to_integer(DataAdr) = 100 and
        to_integer(WriteData) = 7) then
      report "NO ERRORS: Simulation succeeded"
           severity failure;
    elsif (DataAdr /= 96) then
      report "Simulation failed" severity failure;
    end if;
 end if;
end process;
end;
```

HDL-пример 2.13. МОДУЛЬ ВЕРХНЕГО УРОВНЯ

SystemVerilog

```
module top(input  logic        clk, reset,
           output logic [31:0] WriteData, DataAdr,
           output logic        MemWrite);

  logic [31:0] PC, Instr, ReadData;

  // определить процессор и блоки памяти
  arm arm(clk, reset, PC, Instr, MemWrite, DataAdr,
          WriteData, ReadData);
  imem imem(PC, Instr);
  dmem dmem(clk, MemWrite, DataAdr, WriteData,
            ReadData);
endmodule
```

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;
entity top is - модуль верхнего уровня для теста
 port(clk, reset: in STD_LOGIC;
      WriteData,
     DataAdr:  buffer STD_LOGIC_VECTOR(31 downto 0);
      MemWrite: buffer STD_LOGIC);
end;

architecture test of top is
 component arm
  port(clk, reset: in STD_LOGIC;
       PC:        out STD_LOGIC_VECTOR(31 downto 0);
       Instr:      in STD_LOGIC_VECTOR(31 downto 0);
```

```
      MemWrite:  out STD_LOGIC;
      ALUResult,
      WriteData: out STD_LOGIC_VECTOR(31 downto 0);
      ReadData:  in STD_LOGIC_VECTOR(31 downto 0));
 end component;
 component imem
  port(a:  in STD_LOGIC_VECTOR(31 downto 0);
       rd: out STD_LOGIC_VECTOR(31 downto 0));
 end component;
 component dmem
  port(clk, we: in STD_LOGIC;
       a, wd:   in STD_LOGIC_VECTOR(31 downto 0);
       rd:      out STD_LOGIC_VECTOR(31 downto 0));
 end component;
 signal PC, Instr,
        ReadData: STD_LOGIC_VECTOR(31 downto 0);
begin
  -- определить процессор и блоки памяти
  i_arm: arm port map(clk, reset, PC, Instr,
                      MemWrite, DataAdr,
                      WriteData, ReadData);
  i_imem: imem port map(PC, Instr);
  i_dmem: dmem port map(clk, MemWrite, DataAdr,
                        WriteData, ReadData);
end;
```

HDL-пример 2.14. ПАМЯТЬ ДАННЫХ

SystemVerilog

```
module dmem(input  logic        clk, we,
            input  logic [31:0] a, wd,
            output logic [31:0] rd);

  logic [31:0] RAM[63:0];

  assign rd = RAM[a[31:2]]; // выровнять на границу
                            // слова

  always_ff @(posedge clk)
    if (we) RAM[a[31:2]] <= wd;
endmodule
```

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all; use STD.TEXTIO.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;
entity dmem is -- память данных
  port(clk, we: in  STD_LOGIC;
       a, wd:   in  STD_LOGIC_VECTOR(31 downto 0);
       rd:      out STD_LOGIC_VECTOR(31 downto 0));
end;

architecture behave of dmem is
begin
  process is
   type ramtype is array (63 downto 0) of
                   STD_LOGIC_VECTOR(31 downto 0);
   variable mem: ramtype;
  begin - читать или писать в память
   loop
    if clk'event and clk = '1' then
     if (we = '1') then
       mem(to_integer(a(7 downto 2))) := wd;
     end if;
    end if;
    rd <= mem(to_integer(a(7 downto 2)));
    wait on clk, a;
   end loop;
  end process;
end;
```

HDL-пример 2.15. ПАМЯТЬ КОМАНД

SystemVerilog

```
module imem(input  logic [31:0] a,
            output logic [31:0] rd);

  logic [31:0] RAM[63:0];

  initial
    $readmemh("memfile.dat",RAM);

  assign rd = RAM[a[31:2]]; // выровнять
                            // на границу слова
endmodule
```

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all; use STD.TEXTIO.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;
entity imem is -- память команд
 port(a:  in  STD_LOGIC_VECTOR(31 downto 0);
      rd: out STD_LOGIC_VECTOR(31 downto 0));
end;

architecture behave of imem is -- память команд
begin
 process is
  file mem_file: TEXT;
  variable L: line;
  variable ch: character;
  variable i, index, result: integer;
  type ramtype is array (63 downto 0) of
                       STD_LOGIC_VECTOR(31 downto 0);
  variable mem: ramtype;
 begin

  -- инициализировать память из файла
  for i in 0 to 63 loop
    mem(i) := (others => '0'); -- установить все в
                               -- низкий уровень
  end loop;
  index := 0;
  FILE_OPEN(mem_file, "memfile.dat", READ_MODE);
  while not endfile(mem_file) loop
   readline(mem_file, L);
   result := 0;
   for i in 1 to 8 loop
    read(L, ch);
    if '0' <= ch and ch <= '9' then
     result := character'pos(ch) - character'pos('0');
    elsif 'a' <= ch and ch <= 'f' then
    result :=character'pos(ch) - character'pos('a') + 10;
    elsif 'A' <= ch and ch <= 'F' then
    result :=character'pos(ch) - character'pos('A') + 10;
    else report "Formaterror on line " &
                integer'image(index)
      severity error;
    end if;
    mem(index)(35-i*4 downto 32-i*4) :=
      to_std_logic_vector(result,4);
   end loop;
   index := index + 1;
  end loop;

  -- читать память
  loop
   rd <= mem(to_integer(a(7 downto 2)));
   wait on a;
  end loop;
 end process;
end
```

2.7. Улучшенные микроархитектуры

В высокопроизводительных микропроцессорах используется множество приемов для повышения скорости выполнения программ. Напомним, что время, требуемое для выполнения программы, пропорционально периоду тактового сигнала, а также среднему *количеству тактов на команду* (CPI). Таким образом, чтобы увеличить производительность, необходимо либо ускорить тактовый сигнал, либо снизить CPI. В этом разделе дается обзор некоторых способов достижения данной цели. Детали реализации довольно сложны, поэтому мы рассмотрим только общие концепции. Книга Хеннесси и Паттерсона «Архитектура компьютера» (*John L. Hennessy*, *David A. Patterson*. «Computer Architecture», Fifth Edition: A Quantitative Approach. The Morgan Kaufmann Series in Computer Architecture. Sep. 30, 2011) является наиболее авторитетным источником для тех, кто захочет вникнуть в детали.

Благодаря прогрессу в технологии производства интегральных схем размеры транзисторов неуклонно уменьшаются. С уменьшением размера транзисторы работают быстрее и, как правило, потребляют меньше электроэнергии. Таким образом, даже если не менять микроархитектуру, частота тактового сигнала может увеличиться просто потому, что все логические элементы стали быстрее. Кроме того, чем меньше по размеру транзисторы, тем больше их помещается на кристалле. Разработчики микроархитектуры используют дополнительные транзисторы, чтобы строить более сложные процессоры или помещать больше процессоров на кристалл. К сожалению, увеличение количества транзисторов и скорости их работы приводит к увеличению энергопотребления (см. **раздел 1.8**). На настоящий момент энергопотребление стало одной из главных забот разработчиков микропроцессоров. Перед ними встает непростая задача добиться компромисса между скоростью, энергопотреблением и стоимостью микросхем, некоторые из которых содержат миллиарды транзисторов и являются одними из самых сложных систем, когда-либо созданных человечеством.

2.7.1. Длинные конвейеры

Помимо улучшения технологического процесса, простейший способ увеличить тактовую частоту процессора – поделить конвейер на большее количество стадий. Каждая стадия в этом случае содержит меньше логики и, следовательно, может работать быстрее. В начале этой главы мы рассмотрели классический пятистадийный конвейер, но в наши дни нередко можно увидеть конвейеры из 10–20 стадий.

Максимальное количество стадий конвейера ограничено конфликтами в конвейере, накладными расходами на организацию конвейера и стоимостью. Чем длиннее конвейер, тем больше зависимостей. Неко-

торые зависимости могут быть разрешены при помощи пробрасывания через байпас, но другие требуют приостановки, которая увеличивает CPI. Регистры, которые находятся между стадиями конвейера, приводят к накладным расходам из-за их времени предустановки и задержки распространения, а также из-за сдвига фазы сигнала синхронизации (clock skew). Из-за этого добавление каждой последующей стадии дает все меньший прирост производительности. Наконец, добавление дополнительных стадий увеличивает стоимость разработки и производства процессора, так как приходится вводить новые конвейерные регистры и оборудование для разрешения конфликтов.

В конце 1990-х — начале 2000-х годов в основе маркетинга микропроцессоров лежала тактовая частота ($1/Tc$). Это подталкивало к использованию очень длинных конвейеров (от 20 до 31 стадий в Pentium IV), чтобы максимизировать тактовую частоту, даже если выигрыш в общей производительности оказывался под вопросом. Энергопотребление пропорционально тактовой частоте и возрастает вместе с количеством конвейерных регистров, поэтому теперь, когда энергопотребление выходит на первый план, длина конвейеров уменьшается.

Пример 2.9

Рассмотрим построение конвейерного процессора путем разделения однотактного процессора на N стадий. Задержка распространения сигнала через комбинационную логику однотактного процессора составляет 740 пс. К этому нужно добавить накладные расходы на организацию конвейера, составляющие 90 пс. Предположим, что комбинационная логика может быть поделена на произвольное число стадий, а логика обнаружения конфликтов не увеличивает задержку. CPI пятистадийного конвейера из **примера 2.7** равен 1.23. Также предположим, что каждая дополнительная стадия увеличивает CPI на 0.1 из-за неправильного предсказания переходов и прочих конфликтов. При каком числе стадий процессор будет выполнять программы быстрее всего?

Решение. Длительность такта для N-стадийного конвейера равна $T_c = (740/N + 90)$ пс. CPI равно $1.23 + 0.1(N - 5)$. Время выполнения одной команды равно произведению длительности такта на CPI. На **Рис. 2.61** приведены графики зависимости длительности такта и времени выполнения команды от числа стадий. Минимальное время выполнения команды равно 279 пс при числе стадий $N = 8$. Это лишь немногим лучше, чем время 293 пс на команду, достигаемое при использовании пятистадийного конвейера.

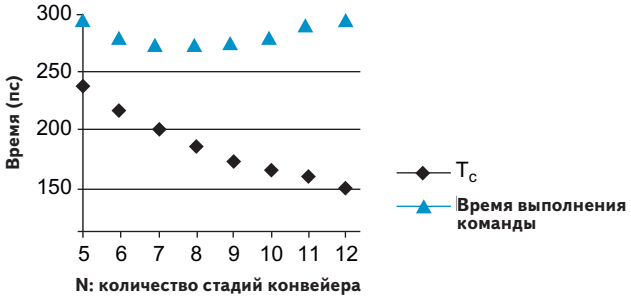


Рис. 2.61. Зависимость длительности такта и времени выполнения команды от количества стадий конвейера

2.7.2. Микрооперации

Вспомним наши принципы проектирования: «единообразие способствует простоте» и «типичный сценарий должен быть быстрым». В чистой архитектуре с сокращенным набором команд (RISC), например MIPS, имеются только простые команды, обычно такие, которые можно выполнить за один такт на простом и быстром тракте данных с трехпортовым регистровым файлом, простым АЛУ и с одним доступом к памяти данных, – такие архитектуры мы разрабатывали в этой главе. В архитектурах со сложным набором команд (CISC) обычно имеются команды, требующие больше регистров, больше операций сложения или больше одного доступа к памяти. Например, в микропроцессорах x86 команда `ADD [ESP], [EDX + 80 + EDI*2]` читает три регистра, складывает базовый адрес, смещение и масштабированный индекс, читает данные из двух ячеек памяти, складывает их значения и записывает результат обратно в память[4]. Микропроцессор, способный выполнить все эти функции сразу, будет медленнее выполнять и более простые, и часто встречающиеся команды, что совершенно нежелательно.

Разработчики компьютерной архитектуры стремятся сделать типичный случай быстрым, для чего определяют набор простых микроопераций, для выполнения которых достаточно простого такта данных. Каждая настоящая команда раскладывается на одну или несколько микроопераций. Например, если бы мы определили микрооперации, похожие на базовые команды ARM и несколько временных регистров, скажем T1 и T2, для хранения промежуточных результатов, то рассмотренную выше команду x86 можно было бы представить в виде последовательности семи микроопераций:

```
ADD T1, [EDX + 80] ; T1 <- EDX + 80
LSL T2, EDI, 2     ; T2 <- EDI*2
ADD T1, T2, T2     ; T1 <- EDX + 80 + EDI*2
LDR T1, [T1]       ; T1 <- MEM[EDX + 80 + EDI*2]
LDR T2, [ESP]      ; T2 <- MEM[ESP]
ADD T1, T2, T1     ; T1 <- MEM[ESP] + MEM[EDX + 80 + EDI*2]
STR T1, [ESP]      ; MEM[ESP]<- MEM[ESP] + MEM[EDX + 80 + EDI*2]
```

Хотя в большинстве своем команды ARM просты, некоторые все же можно разложить на несколько микроопераций. Так, для команд загрузки с постиндексной адресацией (например, `LDR R1, [R2], #4`) требуется второй порт записи в регистровом файле. Команды обработки данных в режиме адресации регистр – сдвиговый регистр (например, `ORR R3, R4, R5, LSL R6`) требуют наличия третьего порта чтения в регистровом файле. Вместо того чтобы включать пятипортовый регистровый файл, тракт

[4] Архитектура x86 не поддерживает команду ADD с двумя операндами из памяти, но для хорошей иллюстрации можно немного пренебречь достоверностью. – *Прим. перев.*

данных ARM мог бы декодировать эти сложные команды, представив каждую в виде двух более простых:

Сложная операция	Последовательность микроопераций
`LDR R1, [R2], #4`	`LDR R1, [R2]` `ADD R2, R2, #4`
`ORR R3, R4, R5 LSL R6`	`LSL T1, R5, R6` `ORR R3, R4, T1`

Конечно, программист мог бы сам написать более простые команды, и программа работала бы быстрее, но одна сложная команда занимает меньше памяти, чем две простые. На чтение команд из внешней памяти тратится значительная энергия, так что сложная команда может заодно и снизить энергопотребление. Набор команд ARM оказался столь успешным в том числе и потому, что архитекторы тщательно подошли к отбору команд, благодаря чему плотность кода оказалась лучше, чем в чистых RISC-архитектурах типа MIPS, но при этом декодирование эффективнее, чем для CISC-архитектур типа x86.

Разработчики микроархитектуры принимают решение о том, как поступить: включить оборудование для реализации сложной операции непосредственно или разложить ее на последовательность микроопераций. Подобные решения приходится принимать и по поводу других вариантов, описываемых далее в этом разделе. Каждому выбору соответствует некоторая точка в пространстве производительность — энергопотребление — стоимость.

2.7.3. Предсказание условных переходов

Теоретически CPI идеального конвейерного процессора должно быть равно 1.0. Одной из основных причин более высокого CPI является штраф за неправильно предсказанные переходы. С увеличением длины конвейера необходимость перехода выясняется на все более поздних стадиях конвейера. Таким образом, штраф становится все больше, т. к. конвейер должен быть очищен от всех команд, выбранных после неправильно предсказанного перехода. Чтобы разрешить эту проблему, в большинстве конвейерных процессоров используется *предсказатель условных переходов*, позволяющий с высокой вероятностью угадать, стоит ли осуществлять переход. Напомним, что наш конвейер из **раздела 2.5.3** всегда просто предсказывал, что переход не произойдет.

Некоторые переходы происходят, когда программа доходит до конца цикла (например, в предложениях for или while) и переходит к его началу для новой итерации. Циклы часто выполняются много раз, поэтому такие условные переходы назад, как правило, выполняются. Простейший метод предсказания переходов состоит в том, чтобы проверить направление перехода и считать, что переход назад всегда будет выполнен. Такой метод называется *статическим предсказанием переходов*, потому что он не зависит от истории выполнения программы.

Переходы вперед трудно предсказать без детального понимания конкретной программы. Поэтому в большинстве процессоров используются

динамические предсказатели переходов, которые анализируют историю выполнения программы, чтобы предсказать, нужно ли выполнить переход. Динамические предсказатели переходов содержат таблицу, содержащую несколько сотен (или тысяч) последних команд перехода, выполненных процессором. Эта таблица, которую иногда называют *буфером адресов перехода* (branch target buffer), содержит адреса переходов и информацию о том, был ли переход выполнен.

Чтобы проиллюстрировать работу динамического предсказателя переходов, рассмотрим цикл из **примера кода 1.17**. Цикл повторяется 10 раз, причем команда `BGE` для выхода из цикла выполняется только на последней итерации.

```
    MOV R1, #0
    MOV R0, #0
FOR
    CMP R0, #10
    BGE DONE
    ADD R1, R1, R0
    ADD R0, R0, #1
    B   FOR
DONE
```

Однобитовый динамический предсказатель переходов запоминает, был ли переход выполнен в прошлый раз, и предсказывает, что в следующий раз произойдет то же самое. Пока цикл повторяется, предсказатель помнит, что в прошлый раз команда `BGE` не была выполнена, и предсказывает, что она не будет выполнена и в следующий раз. Это предсказание остается правильным вплоть до последней итерации, на которой переход все-таки выполняется. К сожалению, если цикл запустить снова, то предсказатель переходов будет помнить, что в последний раз условный переход был выполнен. Поэтому на первой итерации запущенного заново цикла предсказатель ошибется – неправильно предскажет, что переход нужно выполнить. Стало быть, однобитный предсказатель переходов ошибается на первой и на последней итерации цикла.

Двухбитовый динамический предсказатель переходов решает эту проблему, используя четыре состояния: переход *точно выполнится*, переход, *скорее, выполнится*, переход, *скорее, не выполнится* и переход *точно не выполнится* (strongly taken, weakly taken, weakly not taken, strongly not taken), как показано на **Рис. 2.62**. Пока цикл повторяется, предсказатель переходит в состояние «точно не выполнится» и остается в нем, предсказывая, что условный переход не будет выполнен и в следующий раз. Это предсказание остается верным вплоть до последней итерации, на которой переход выполняется, и переводит предсказатель в состояние «скорее, не выполнится». Когда цикл начнется снова, предсказатель переходов правильно предскажет, что переход не должен быть выполнен, и снова перейдет в состояние «точно не выполнится». Короче

говоря, двухбитовый предсказатель переходов неправильно предсказывает только переход на последней итерации цикла.

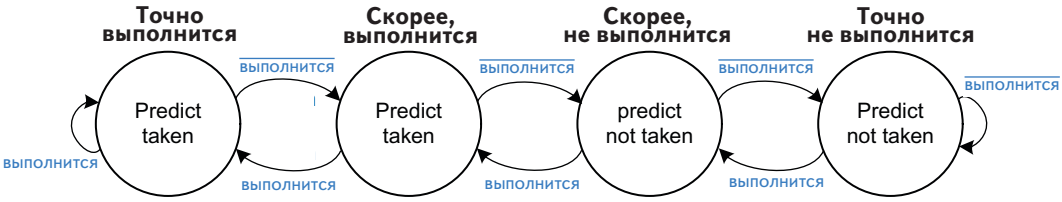


Рис. 2.62. Диаграмма состояний двухбитового предсказателя переходов

2.7.4. Суперскалярный процессор

Тракт данных *суперскалярного процессора* содержит несколько копий функциональных блоков, что позволяет ему выполнять несколько команд одновременно. На **Рис. 2.63** показана диаграмма двухканального (2-way) суперскалярного процессора, который осуществляет выборку и выполнение двух команд за один такт. Тракт данных выбирает из памяти команд две команды за раз. Он содержит регистровый файл с шестью портами, чтобы читать четыре операнда и записывать назад два результата на каждом такте. Тракт данных также содержит два АЛУ и двухпортовую память данных, чтобы выполнять две команды одновременно.

На **Рис. 2.64** показана диаграмма двухканального суперскалярного процессора, который выполняет две команды на каждом такте. В этом случае CPI процессора равно 0.5. Проектировщики зачастую используют величину, обратную CPI, – количество команд на такт (instructions per cycle, IPC), которое для этого процессора равно 2.

Выполнять много команд одновременно трудно из-за зависимостей. Рассмотрим, к примеру, **Рис. 2.65**, на котором показана диаграмма конвейера, выполняющего программу с зависимостями по данным. Зависимости в коде показаны синим цветом. Команда `ADD` зависит от регистра R8, значение которого изменяет команда `LDR`, поэтому эти команды нельзя запускать на выполнение одновременно. На самом деле команда `ADD` приостанавливается еще на один такт, чтобы `LDR` могла пробросить через байпас прочитанное из памяти значение R8 команде `ADD` на пятом такте. Другие конфликты (между `SUB` и `AND` из-за R8 и между `ORR` и `STR` из-за R11) разрешаются путем пробрасывания

Скалярный процессор в каждый момент времени осуществляет вычисления лишь над одной порцией данных. *Векторный* процессор работает над несколькими порциями данных одновременно, но использует для этого только одну команду. Суперскалярный процессор запускает на выполнение одновременно несколько команд, каждая из которых применяется к одной порции данных.

Наш конвейерный процессор ARM является скалярным. Векторные процессоры часто использовались в суперкомпьютерах 1980-х и 1990-х годов, т. к. позволяли эффективно обрабатывать длинные векторы данных, часто встречающиеся в научных расчетах. Они также широко применяются в наше время в *графических процессорах* (GPU). Современные высокопроизводительные микропроцессоры являются суперскалярными, т. к. запуск выполнения нескольких независимых команд одновременно – более гибкий подход, по сравнению с применяемым для обработки векторов.

Однако современные процессоры также включают аппаратные расширения для работы с короткими векторами данных, которые часто бывают необходимы в графических и мультимедийных приложениях. Они называются SIMD-расширениями (Single Instruction Multiple Data – один поток команд, несколько потоков данных) и обсуждаются в **разделе 1.7.5**.

на следующий такт результатов, вычисленных на предыдущем. Эта программа требует пяти тактов для запуска шести команд, ее IPC равно 1.2.

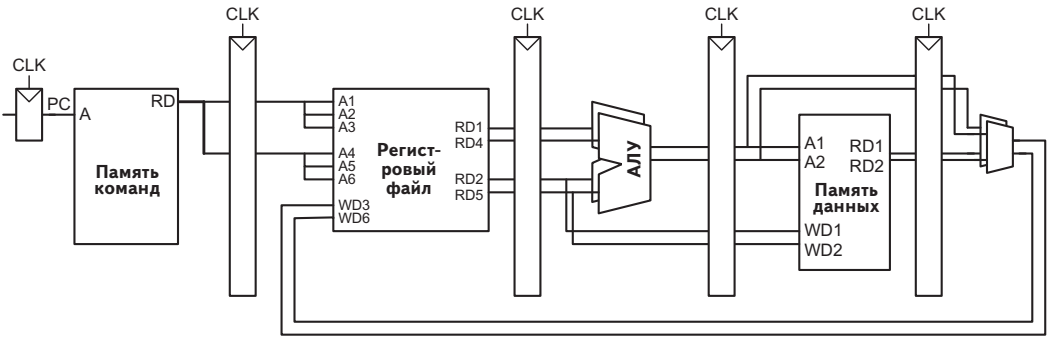


Рис. 2.63. Тракт данных суперскалярного процессора

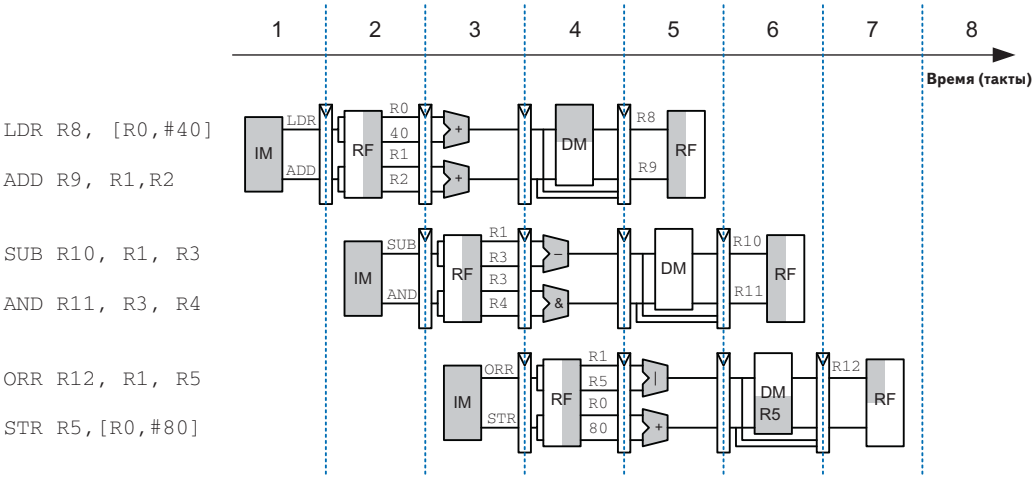


Рис. 2.64. Работающий суперскалярный конвейер

Напомним, что у параллелизма две формы – временная и пространственная. Конвейер – пример временного параллелизма, а наличие нескольких экземпляров одних и тех же исполнительных блоков – пространственного. В суперскалярных процессорах используются обе формы параллелизма во имя достижения производительности, значительно превосходящей производительность наших однотактного и многотактного процессоров.

Коммерческие суперскалярные процессоры могут быть трех-, четырех- или даже шестиканальными. Им приходится обрабатывать как конфликты управления, вызываемые, например, условными переходами, так и конфликты по данным. К сожалению, в реальных программах

встречается много зависимостей, поэтому суперскалярные процессоры с большим количеством каналов редко могут использовать все свои исполнительные блоки полностью. Более того, большое количество исполнительных блоков и сложности с организацией байпаса требуют множества дополнительных логических элементов и потребляют дополнительную электроэнергию.

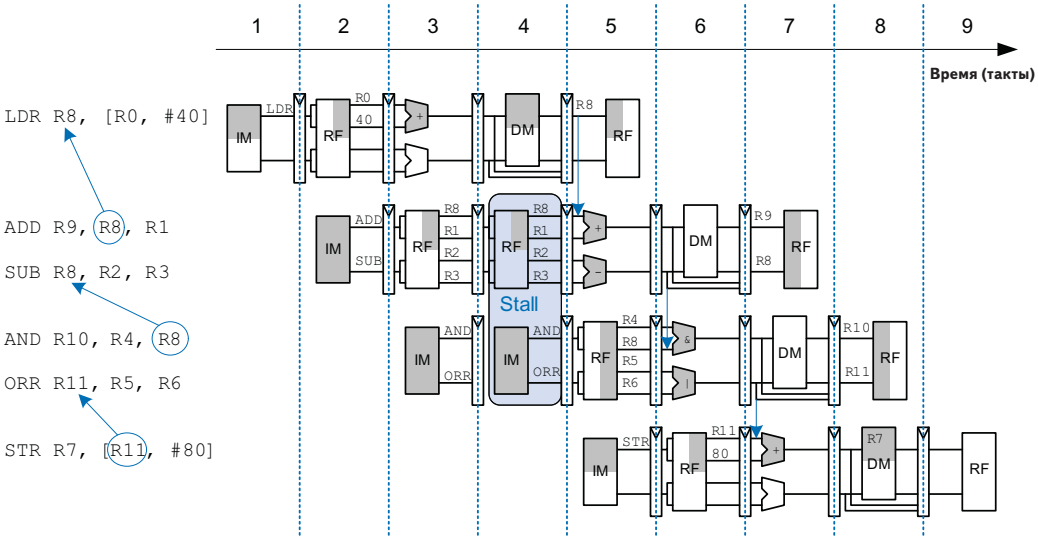


Рис. 2.65. Программа с зависимостями по данным

2.7.5. Процессор с внеочередным выполнением команд

Чтобы справиться с проблемой зависимостей, *процессор с внеочередным выполнением команд* (out-of-order processor) заранее просматривает большое количество команд, чтобы как можно быстрее обнаружить и запустить на выполнение те команды, которые не зависят друг от друга. Команды могут выполняться не в том в порядке, в котором они находятся в программе, но только при условии, что процессор учитывает все зависимости, что позволит программе выдать ожидаемый результат.

Рассмотрим выполнение той же программы, что на **Рис. 2.65**, двухканальным суперскалярным процессором с внеочередным выполнением команд. За один такт процессор может запускать до двух команд из любой части программы при условии соблюдения всех зависимостей. На **Рис. 2.66** показаны зависимости по данным и работа процессора. Чуть позже мы обсудим классификацию зависимостей (RAW и WAR). Ниже описаны ограничения на запуск команд:

- **Такт 1**
  - Команда `LDR` запускается на выполнение.
  - Команды `ADD`, `SUB` и `AND` зависят от `LDR`, так как используют R8, поэтому их пока запустить нельзя. А вот команда `ORR` не зависит от `LDR`, поэтому она тоже запускается на выполнение.
- **Такт 2**
  - Напомним, что между запуском команды `LDR` и моментом, когда ее результат может быть использован зависимой от нее командой, существует задержка в два такта. Поэтому `ADD` запустить пока нельзя, так как она зависит от R8. Команда `SUB` записывает результат в R8, поэтому ее нельзя запускать перед `ADD`, иначе `ADD` получит неверное значение R8. Команда `AND` зависит от `SUB`.
  - Запускается только команда `STR`.
- **Такт 3**
  - На третьем такте значение в R8 становится корректным, поэтому запускается `ADD`. Команда `SUB` тоже запускается на выполнение, потому что `ADD` прочитает R8 раньше, чем `SUB` изменит его.
- **Такт 4**
  - Запускается команда `AND`. Значение R8 пробрасывается от `SUB` к `AND`.

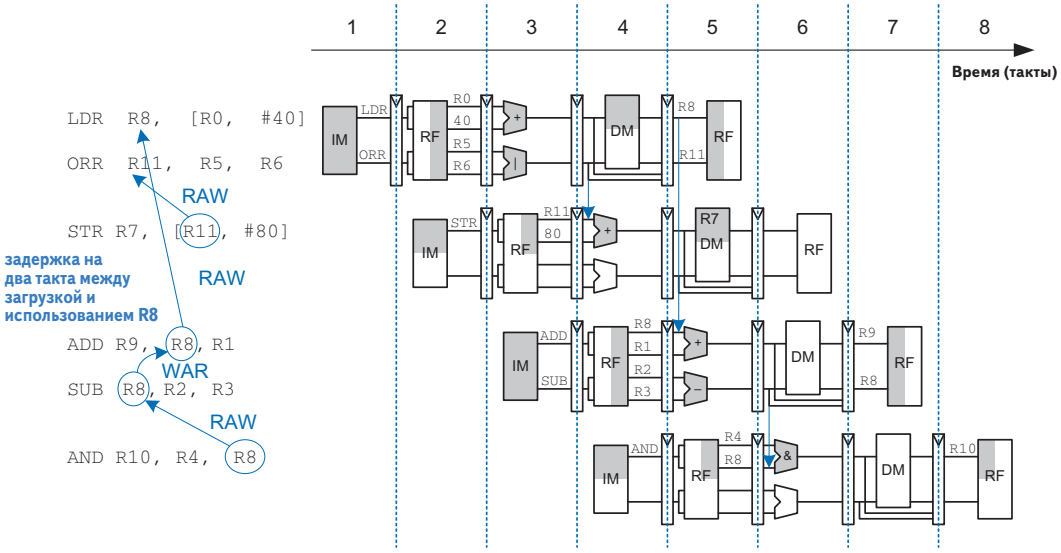


Рис. 2.66. Внеочередное выполнение команд, зависящих друг от друга

Таким образом, процессор с внеочередным выполнением команд запускает шесть команд за четыре такта, то есть его IPC равно 1.5.

Зависимость ADD от LDR из-за использования R8 называется *конфликтом чтения после записи*, или RAW-конфликтом (read-after-write, RAW). Команда ADD не имеет права читать R8, до тех пор пока LDR в него не запишет. Мы уже встречались с таким типом зависимости, когда рассматривали конвейерный процессор, и знаем, как с ней справляться. Такая зависимость по своей природе ограничивает скорость выполнения программы, даже если у процессора есть бесконечно много исполнительных блоков. Аналогично зависимость STR от ORR из-за использования R11 и зависимость AND от SUB из-за использования R8 являются RAW-зависимостями.

Зависимость между SUB и ADD из-за использования R8 называется конфликтом записи после чтения, или WAR-конфликтом (write-after-read, WAR), или *антизависимостью*. Команда SUB не имеет права писать в R8, до того как ADD прочитает его. Это необходимо, чтобы команда ADD получила правильное значение в соответствии с исходным порядком команд в программе. WAR-конфликты не могут возникнуть в простом конвейере, но могут возникнуть в процессоре с внеочередным выполнением команд, если он попытается запустить зависимую команду (в данном случае SUB) слишком рано.

WAR-конфликты не являются неотъемлемым свойством работы программы. Это просто следствие решения программиста использовать один и тот же регистр для двух не связанных друг с другом команд. Если бы команда SUB записывала результат в R12, а не в R8, то зависимость исчезла бы, и можно было бы запустить SUB перед ADD. В архитектуре ARM всего 16 регистров, поэтому иногда программист вынужден использовать регистры повторно, создавая тем самым почву для конфликта, просто потому что все остальные регистры заняты.

Третий тип конфликта, не показанный в программе, называется *конфликтом записи после записи*, или WAW-конфликтом (write-after-write, WAW). Его еще называют *зависимостью вывода* (output dependency), или *ложной зависимостью* (false dependency). WAW-конфликт случается, когда команда пытается писать в регистр, после того как в него уже записала следующая по ходу программы команда. В результате этого конфликта в регистр будет записано неверное значение. Например, в программе ниже две команды, LDR и ADD, пишут в R8. Согласно порядку команд в программе, окончательное значение в R8 должна записать ADD. Если бы процессор с внеочередным выполнением команд попытался выполнить ADD первой, то произошел бы WAW-конфликт.

```
LDR R8, [R3]
ADD R8, R1, R2
```

WAW-конфликты также не являются внутренне присущими, а возникают из-за решения программиста использовать один и тот же регистр

для двух не связанных между собой команд. Если бы команда ADD была запущена первой, то программа могла бы устранить WAW-конфликт, отбросив результат LDR, вместо того чтобы записывать его в R8. Этот прием называется «*раздавливанием*» (squashing) команды LDR[5].

В процессорах с внеочередным выполнением команд используется специальная таблица, чтобы отслеживать команды, ожидающие запуска. Эта таблица, иногда называемая *табло готовности* (scoreboard), содержит информацию о зависимостях. Размер таблицы определяет, сколько команд одновременно могут являться кандидатами на запуск. На каждом такте процессор сверяется с таблицей и запускает столько команд, сколько может, с учетом зависимостей и количества доступных исполнительных блоков (АЛУ, портов памяти и т. д.).

Параллелизм на уровне команд (instruction level parallelism, ILP) – это число команд конкретной программы, которые могут одновременно выполняться на определенной микроархитектуре. Теоретические исследования показали, что ILP для микроархитектур с внеочередным выполнением команд при условии идеального предсказания переходов и очень большого количества исполнительных блоков может быть весьма высоким. К сожалению, на практике даже шестиканальные суперскалярные процессоры с внеочередным выполнением команд редко достигают ILP, превышающего 2 или 3.

2.7.6. Переименование регистров

Для того чтобы устранить WAR- и WAW-конфликты, в процессорах с внеочередным выполнением команд используется прием, который называется *переименованием регистров* (register renaming). Суть его в том, что в процессор добавляются неархитектурные *регистры переименования* (renaming registers). Например, можно добавить 20 регистров переименования, называемых T0–T19. Программист не может использовать эти регистры напрямую, поскольку они не являются частью архитектуры. Но процессор может пользоваться ими для устранения конфликтов.

Например, в предыдущем разделе был показан WAR-конфликт между командами SUB и ADD, который случился из-за повторного использования R8. Процессор с внеочередным выполнением команд мог бы переименовать R8 в T0 для команды SUB. После этого SUB можно было бы выполнить быстрее, потому что у T0 нет зависимости от команды ADD. У процессора есть таблица с информацией о том, какие регистры были переименованы,

[5] Возникает вопрос: зачем вообще нужно запускать команду LDR? Дело в том, что процессоры с внеочередным выполнением команд обязаны гарантировать, что во время выполнения программы произойдут все те же исключения, которые произошли бы, если бы все команды выполнялись в исходном порядке. Команда LDR потенциально может вызвать исключение Data Abort, поэтому ее необходимо запустить, для того чтобы проверить, произойдет исключение или нет, даже если ее результат будет отброшен.

поэтому он может соответствующим образом переименовать регистры и в последующих зависимых командах. В этом примере R8 необходимо переименовать в T0 еще и в команде AND, потому что в ней используется результат команды SUB.

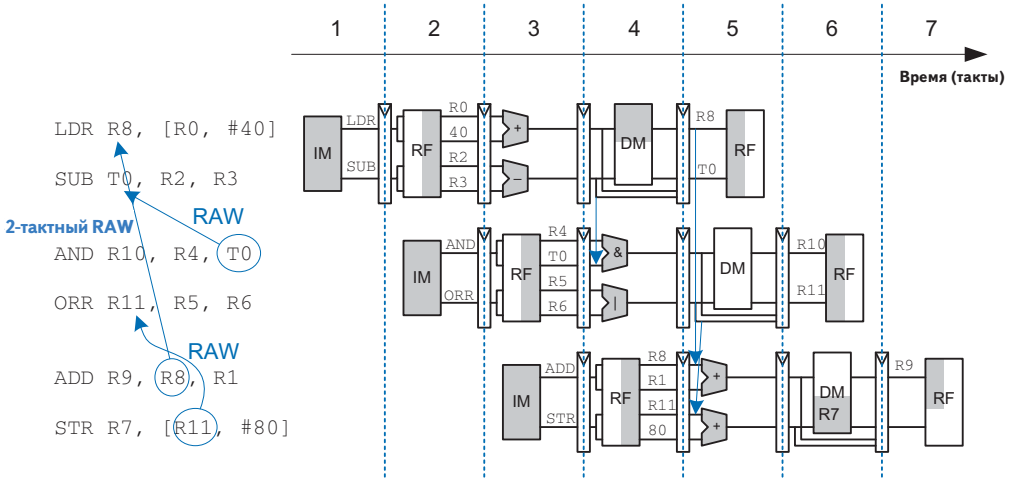


Рис. 2.67. Внеочередное выполнение команд с переименованием регистров

На **Рис. 2.67** показана та же программа, что и на **Рис. 2.65**, но выполняемая процессором с внеочередным выполнением команд с переименованием регистров. Чтобы устранить WAR-конфликт, регистр R8 переименован в T0 в командах SUB и AND. Ниже описаны ограничения на запуск команд.

- **Такт 1**
  - Команда LDR запускается на выполнение.
  - Команда ADD зависит от LDR из-за использования R8, поэтому пока что запустить ее нельзя. Однако команда SUB теперь независима, т. к. ее регистр-приемник переименован в T0, поэтому SUB также запускается на выполнение.
- **Такт 2**
  - Напомним, что между запуском команды LDR и моментом, когда ее результат может быть использован зависимой от нее командой, существует задержка в два такта. Поэтому ADD запустить пока нельзя, так как она зависит от R8.
  - Команда AND зависит от SUB, поэтому ее тоже можно запускать. T0 пробрасывается от SUB к AND через байпас.
  - У команды ORR нет зависимостей, поэтому она тоже запускается.

- **Такт 3**
  - На третьем такте значение в R8 становится корректным, поэтому запускается `ADD`.
  - Значение в R11 также становится корректным, поэтому запускается и `STR`.

Таким образом, процессор с внеочередным выполнением команд и переименованием регистров запускает шесть команд за три такта, то есть его IPC равно 2.

2.7.7. Многопоточность

Так как *параллелизм на уровне команд* (instruction level parallelism, ILP) у реальных программ, как правило, довольно низок, добавление новых и новых исполнительных блоков к суперскалярному процессору или процессору с внеочередным выполнением команд дает все меньший эффект. Еще одной проблемой является то, что основная память гораздо медленнее, чем процессор (мы рассмотрим это в главе 8). Большинство команд загрузки и сохранения данных работают со значительно более быстрой и маленькой *кэш-памятью*. К сожалению, если нужных команд или данных в кэше нет, то процессор может быть приостановлен на 100 и более тактов в ожидании получения информации из основной памяти. Многопоточность – это способ загрузить работой процессор с большим количеством исполнительных блоков, даже если у программы низкий ILP или она приостановлена на время ожидания данных из памяти.

Для того чтобы объяснить суть многопоточности, нам надо определить несколько новых терминов. Программа, которая выполняется на компьютере, называется *процессом*. Компьютеры могут выполнять несколько процессов одновременно. Например, на своем ПК вы можете слушать музыку и сидеть в Интернете, одновременно запустив антивирус. Каждый процесс состоит из одного или более *потоков выполнения* (threads), которые тоже выполняются одновременно. Например, в текстовом редакторе один поток может обрабатывать набор текста пользователем, второй в это время проверяет орфографию, а третий – печатает документ на принтере. При такой организации пользователю не нужно ждать, пока закончится печать, чтобы продолжить ввод текста. Степень, до которой процесс можно разделить на несколько одновременно выполняющихся потоков, определяет его *уровень параллелизма на уровне потоков* (thread level parallelism, TLP).

В обычном процессоре одновременная работа потоков – не более чем иллюзия. Реально потоки выполняются процессором по очереди под управлением операционной системы. Когда «смена» одного потока подходит к концу, ОС сохраняет его архитектурное состояние, загружает

из памяти архитектурное состояние следующего потока и передает ему управление. Эта процедура называется *контекстным переключением* (context switching). При условии, что процессор переключается между потоками достаточно быстро, пользователю кажется, что все потоки выполняются одновременно.

У многопоточного процессора есть несколько копий архитектурного состояния, вследствие чего несколько потоков могут быть активны одновременно. Например, если мы расширим наш процессор, так чтобы у него было четыре счетчика команд и 64 регистра, то одновременно могут быть доступны четыре потока. Если один из них приостанавливается в ожидании данных из основной памяти, то процессор немедленно переключает контекст на другой поток. Это переключение происходит безо всяких накладных расходов, так как счетчик команд и регистры уже доступны и их не надо отдельно загружать. Более того, если один из потоков не может в полной мере использовать все исполнительные блоки суперскалярного процессора из-за недостаточного уровня параллелизма, то другой поток может запустить на исполнение команды на незанятых блоках.

Многопоточность не улучшает производительность отдельного потока, потому что она не повышает ILP. Тем не менее она улучшает общую пропускную способность процессора, так как несколько потоков могут более полно использовать те ресурсы процессора, которые не использовались бы при выполнении одного-единственного потока. Многопоточность относительно легко реализовать, так как требуется добавить только копии счетчика команд и регистрового файла. Исполнительные блоки и память копировать не надо.

2.7.8. Мультипроцессоры

При участии Мэттью Уоткинса

В современных процессорах количество транзисторов огромно. Если использовать эти транзисторы только для увеличения длины конвейера или количества исполнительных блоков в суперскалярном процессоре, то существенного прироста производительности мы не получим и будем лишь понапрасну потреблять энергию. Примерно в 2005 году разработчики компьютерных архитектур склонились к другому принципиальному решению – создание нескольких копий процессора на одном кристалле. Эти копии называются ядрами.

Многопроцессорная система (multiprocessor system), или просто *мультипроцессор*, состоит из нескольких процессоров и аппаратуры для соединения их между собой. Есть три основных класса мультипроцессоров: *симметричные* (или *гомогенные*) мультипроцессоры, *гетерогенные* мультипроцессоры и *кластеры*.

Симметричные мультипроцессоры

Симметричный мультипроцессор состоит из двух или более одинаковых процессоров, подключенных к общей основной памяти. Процессоры могут быть выполнены в виде отдельных микросхем или нескольких ядер на одном кристалле.

Мультипроцессоры можно использовать, чтобы выполнять больше потоков одновременно или чтобы быстрее выполнять один конкретный поток. Выполнять больше потоков одновременно довольно просто – эти потоки можно просто распределить между процессорами. К сожалению, типичному пользователю персонального компьютера обычно нужно выполнять лишь небольшое количество потоков в каждый момент времени. Ускорение одного потока при помощи мультипроцессора – гораздо более сложная задача. Чтобы достичь этого, программист должен разделить один существующий поток на несколько, которые можно будет запустить на разных процессорах. Все еще больше усложняется, если процессорам нужно взаимодействовать между собой. Эффективное использование большого количества процессорных ядер – одна из главных проблем, стоящих перед разработчиками компьютеров и программистами.

У симметричных мультипроцессоров есть ряд преимуществ. Их сравнительно просто проектировать, поскольку процессор можно спроектировать один раз, а затем реплицировать для повышения производительности. Программировать симметричный мультипроцессор и исполнять на нем код тоже относительно легко, потому что любую программу можно запустить на любом процессоре системы и получить примерно одинаковую производительность.

Гетерогенные мультипроцессоры

К сожалению, нет никакой гарантии, что с увеличением количества симметричных ядер производительность системы продолжит улучшаться. По состоянию на 2015 год пользовательские приложения, запускаемые на домашних компьютерах, использовали небольшое число потоков в каждый момент времени. У типичного пользователя обычно запущена всего пара программ одновременно. И хотя этого достаточно, чтобы загрузить двух- или четырехъядерную систему, добавление большего числа ядер будет приводить ко все менее заметным результатам до тех пор, пока программы не начнут использовать параллелизм более широко. Кроме того, т. к. процессоры общего назначения разрабатываются с целью обеспечить хорошую среднюю производительность на широком классе задач, то они, как правило, являются далеко не самым энергоэффективным инструментом для решения конкретной задачи. Энергоэффективность особенно важна в мобильных системах, где энергопотребление сильно ограничено.

Гетерогенные мультипроцессоры (heterogeneous multiprocessors) решают эти проблемы путем использования ядер разного типа и (или) специализированной аппаратуры в одной системе. Каждое приложение использует те ресурсы, которые позволяют достичь либо наилучшей производительности, либо наилучшего соотношения производительности и энергопотребления. Так как в наши дни разработчики могут использовать сколько угодно транзисторов, то никого особенно не заботит, что не каждое приложение будет применять все имеющиеся в наличии аппаратные блоки. Гетерогенные системы могут принимать разные формы. Они могут включать ядра с различными микроархитектурами, имеющие разное соотношение энергопотребления, производительности и занимаемой на кристалле площади.

Ученые, занятые поиском признаков инопланетного разума, используют самый большой в мире кластер мультипроцессоров, пытаясь найти в данных от радиотелескопов закономерности, которые могли бы быть признаками жизни в других солнечных системах. Этот кластер, работающий с 1999 года, состоит из персональных компьютеров, принадлежащих более чем 6 миллионам добровольцев во всем мире.
Когда компьютер из кластера простаивает, он получает блок данных от центрального сервера, анализирует эти данные и отсылает результаты обратно на сервер. Вы тоже можете поделиться с кластером неиспользуемым временем своего компьютера, зайдя на сайт `setiathome.berkeley.edu`.

Одна из стратегий разработки гетерогенных систем, популярная в мире ARM, называется *big.LITTLE*. Такая система содержит как энергоэффективное (*LITTLE*), так и высокопроизводительное (*big*) ядро. Ядра типа *LITTLE*, например Cortex-A53, представляют собой процессоры с последовательным одиночным и попарным выполнением команд, обладающие высокой энергоэффективностью и пригодные для решения простых задач. Ядра типа *big*, например Cortex-A57, – это более сложные суперскалярные процессоры с внеочередным выполнением команд, гарантирующие высокую производительность при пиковых нагрузках.

Другая стратегия построения гетерогенных систем связана с ускорителями. Такая система содержит специализированное оборудование, оптимизированное ради достижения высокой производительности или энергоэффективности на задачах определенного типа. Например, современные мобильные системы на кристалле (system-on-chip, SoC) могут содержать специализированные ускорители для обработки графики, видео, беспроводной связи, задач реального времени и криптографических задач. Эти ускорители могут быть в 10–100 раз эффективнее процессора общего назначения, но только для конкретной задачи. Еще одним классом ускорителей являются процессоры цифровой обработки сигналов. В этих процессорах реализован специализированный набор команд, оптимизированный для математических расчетов.

У гетерогенных систем есть и недостатки. Они сложнее и в разработке, и в программировании, так как требуется не только спроектировать разнообразные аппаратные блоки, но и решить, когда и как наилучшим образом использовать различные ресурсы системы. В конечном итоге и у симметричных, и у гетерогенных систем есть свои ниши. Симметричные мультипроцессоры подходят, например, для больших центров обработки данных, где нет недостатка в задачах с высоким параллелизмом на

уровне потоков. Гетерогенные системы хороши в случае более разнообразной или специализированной вычислительной нагрузки.

Кластеры

В кластерных мультипроцессорах у каждого процессора имеется собственная подсистема памяти. Одним из типов кластеров является группа персональных компьютеров, подключенных к сети и совместно решающих какую-то большую задачу. Важное значение приобрели кластеры другого типа – центры обработки данных (ЦОД), в которых стойки с серверами и дисками объединены в общую сеть, подключенную к общей системе энергоснабжения и охлаждения. Крупные интернет-компании, в т. ч. Google, Amazon и Facebook, стали движущей силой быстрого развития ЦОДов, поскольку должны поддерживать миллионы пользователей во всем мире.

2.8. Живой пример: эволюция микроархитектуры ARM

DMIPS (Dhrystone millions of instructions per second — миллионов команд в секунду по тесту Dhrystone) — мера производительности.

В этом разделе мы проследим эволюцию архитектуры и микроархитектуры ARM с момента ее выхода на сцену в 1985 году. В **Табл. 2.7** приведены основные вехи на пути увеличения IPC в 10 раз и тактовой частоты в 250 раз за тридцать лет. За это время сменилось восемь версий архитектуры. Частота, площадь на кристалле и энергопотребление меняются в зависимости от техпроцесса, целей, графика разработки и возможностей коллектива проектировщиков. В таблице показаны частоты, репрезентативные для процесса производства на момент вывода изделия на рынок. В основном прирост частоты обусловлен прогрессом в области технологии транзисторов, а не микроархитектуры. Относительный размер нормирован на размер транзисторов и может изменяться в широких пределах в зависимости от размера кэша и других факторов.

На **Рис. 2.68** показана фотография процессора ARM1, содержавшего 25 000 транзисторов и имевшего трехстадийный конвейер. Посчитав, можно обнаружить 32 бита тракта данных внизу. Регистровый файл расположен слева, АЛУ – справа. К левому краю примыкает счетчик команд; обратите внимание, что два младших бита в нижней части пусты (привязаны к 0), а шесть бит наверху различаются, потому что содержат биты состояния. Устройство управления расположено над трактом данных. Часть прямоугольных блоков – это ПЛМ, реализующие логику управления. Прямоугольники, примыкающие к краям, – это контактные площадки ввода-вывода, от которых отходят крохотные золотые проволочные соединения, ведущие за пределы рисунка.

Таблица 2.7. Эволюция процессоров ARM

Микроархитектура	Год	Архитектура	Длина конвейера	DMIPS МГц	Репрезентативная частота (МГц)	Кэш L1	Относительный размер
ARM1	1985	v1	3	0.33	8	N/A	0.1
ARM6	1992	v3	3	0.65	30	4 КБ объединенный	0.6
ARM7	1994	v4T	3	0.9	100	0–8 КБ объединенный	1
ARM9E	1999	v5TE	5	1.1	300	0–16 КБ К + Д	3
ARM11	2002	v6	8	1.25	700	4–64 КБ К + Д	30
Cortex-A9	2009	v7	8	2.5	1000	16–64 КБ К + Д	100
Cortex-A7	2011	v7	8	1.9	1500	8–64 КБ К + Д	40
Cortex-A15	2011	v7	15	3.5	2000	32 КБ К + Д	240
Cortex-M0 +	2012	v7M	2	0.93	60–250	Нет	0.3
Cortex-A53	2012	v8	8	2.3	1500	8–64 КБ К + Д	50
Cortex-A57	2012	v8	15	4.1	2000	48 КБ К + 32 КБ D	300

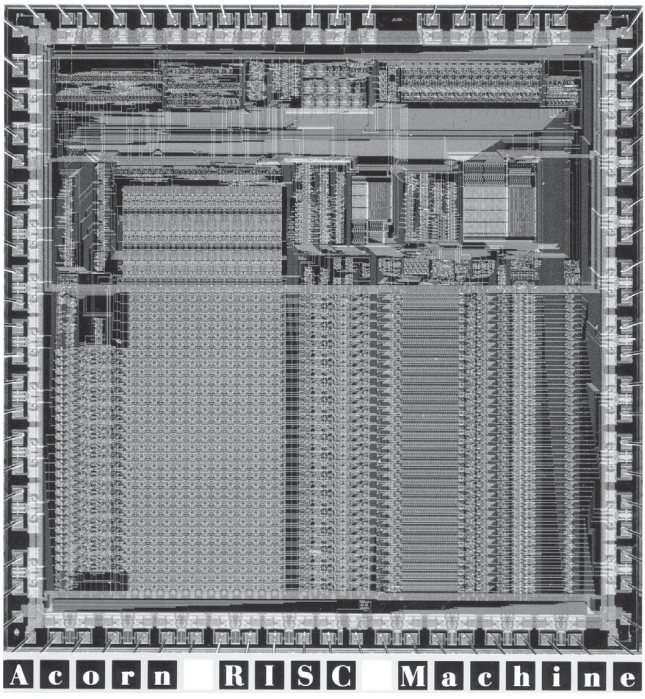


Рис. 2.68. Фотография кристалла ARM1
(печатается с разрешения корпорации ARM, © 1985 ARM Ltd.)

Процессор ARM1 — совместное детище Софи Уилсон и Стива Фэрбера.

Софи Уилсон (1957—) родилась в английском городе Йоркшире и изучала информатику в Кэмбриджском университете. Она спроектировала операционную систему и написала интерпретатор языка BBC Basic для компании Acorn Computer, а затем принимала участие в проектировании ARM1 и последующих процессоров, вплоть до ARM7. К 1999 г. она спроектировала процессор цифровой обработки сигналов Firepath SIMD и создала для его продвижения новую компанию, приобретенную Broadcom в 2001 году. В настоящее время занимает должность старшего управляющего в компании Broadcom Corporation и является действительным членом Королевского общества, Королевской инженерной академии наук, Британского компьютерного общества и Общества женщин-инженеров.

Софи Уилсон
(фотография печатается с разрешения правообладателя)

В 1990 году из компании Acorn выделилась группа проектирования процессоров, которая образовала отдельную компанию, Advanced RISC Machines (позже переименованную в ARM Holdings), и приступила к лицензированию архитектуры ARMv3. В этой архитектуре биты состояния были вынесены из счетчика команд в регистр текущего состояния программы (CPSR), а разрядность PC увеличена до 32. Компания Apple приобрела крупную долю в ARM и использовала процессор ARM 610 в компьютере Newton, первом в мире компьютере класса PDA (Personal Digital Assistant — персональный цифровой помощник) и одном из первых коммерческих приложений распознавания рукописного текста. Newton опередил свое время, но заложил фундамент для более успешных PDA, на смену которым позже пришли смартфоны и планшеты.

ARM добилась оглушительного успеха после выхода линейки ARM7 в 1994 году, а особенно ARM7TDMI, ставшего одним из самых широко распространенных RISC-процессоров, применяемых во встраиваемых системах на протяжении последующих 15 лет. В ARM7TDMI использовался набор команд ARMv4T, в котором впервые был реализован набор команд Thumb для повышения плотности кода и определены команды загрузки и сохранения полуслов и байтов со знаком. Акроним TDMI означает **T**humb, JTAG **D**ebug, fast **M**ultiply, and **I**nCircuit Debug (**T**humb, отладчик JTAG, быстрое умножение и внутрисхемная отладка). Различные средства отладки помогают программистам писать код, размещаемый в оборудовании, и тестировать его на ПК, подключенном простым кабелем, — в то время это стало важным техническим достижением. В ARM7 использовался трехстадийный конвейер со стадиями выборки, декодирования и выполнения. У процессора был объединенный кэш команд и данных. Поскольку в конвейерном процессоре кэш обычно занят на каждом такте выборкой команд, ARM7 приостанавливал команды доступа к памяти на стадии выполнения, чтобы дать кэшу время для обращения к данным. На **Рис. 2.69** приведена блок-схема процессора. Компания ARM не производила процессор сама, а продавала лицензии другим компаниям, которые включали его в состав более крупных систем на кристалле (SoC). Заказчики могли приобрести процессор в формате Hard Macro (полный и эффективный, но негибкий макет, который можно сразу перенести на кристалл) или в формате Soft Macro (код на языке Verilog, который заказчик мог синтезировать своими силами). Процессор ARM7 использо-

вался в самых разных изделиях: мобильных телефонах, Apple iPod, Lego Mindstorms NXT, игровых автоматах Nintendo и автомобилях. С тех пор практически во всех мобильных телефонах применяются процессоры ARM.

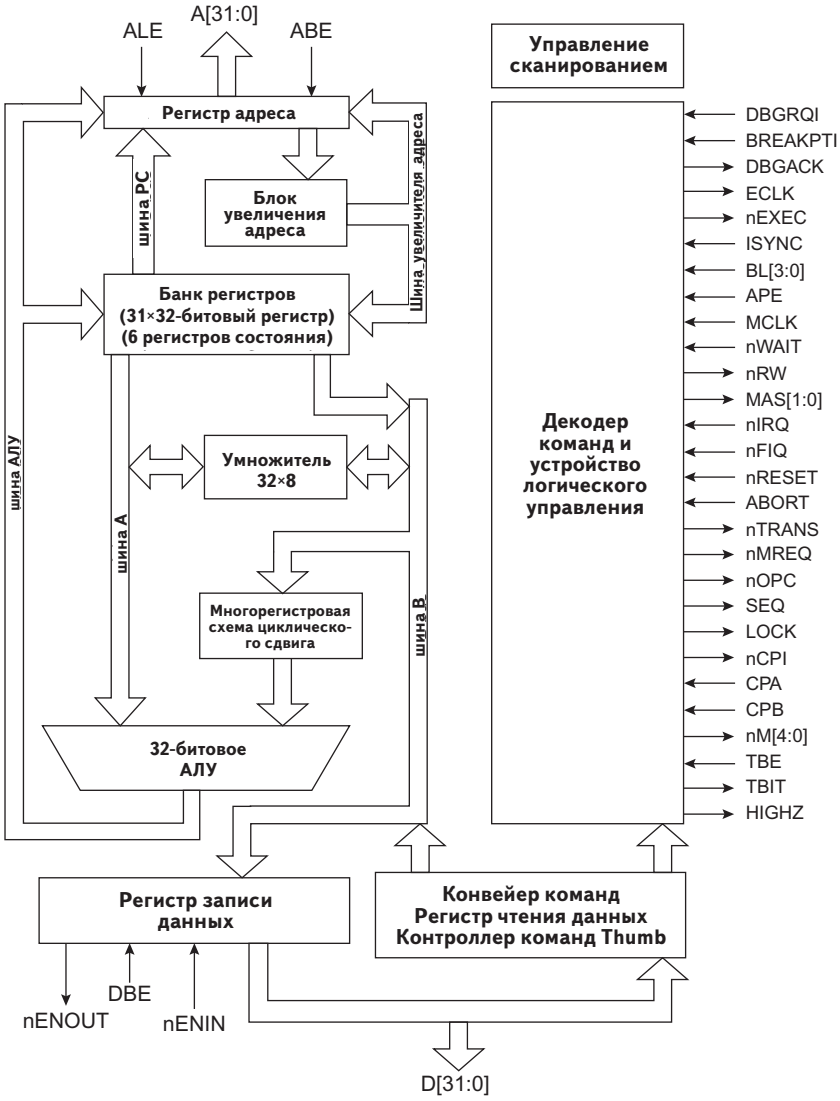


Рис. 2.69. Блок-схема процессора ARM7
(печатается с разрешения ARM. © 1998 ARM Ltd.)

Линейка ARM9E стала развитием ARM7: были реализованы пятистадийный конвейер, похожий на описанный в этой главе, отдельные кэши

команд и данных, а также новый набор команд Thumb и команды для цифровой обработки сигналов в архитектуре ARMv5TE. На **Рис. 2.70** приведена блок-схема процессора ARM9, содержавшего многие из компонентов, встречавшихся в этой главе, а также умножитель и сдвиговый регистр. Сигналы IA/ID/DA/DD – это шина адреса команд, шина адреса данных, шина команд и шина данных (соединяющие процессор с подсистемой памяти), а сигнал IAreg – это счетчик команд. В процессоре следующего поколения ARM11 число стадий конвейера было увеличено до восьми, чтобы поднять тактовую частоту, и, кроме того, были определены наборы команд Thumb2 и SIMD.

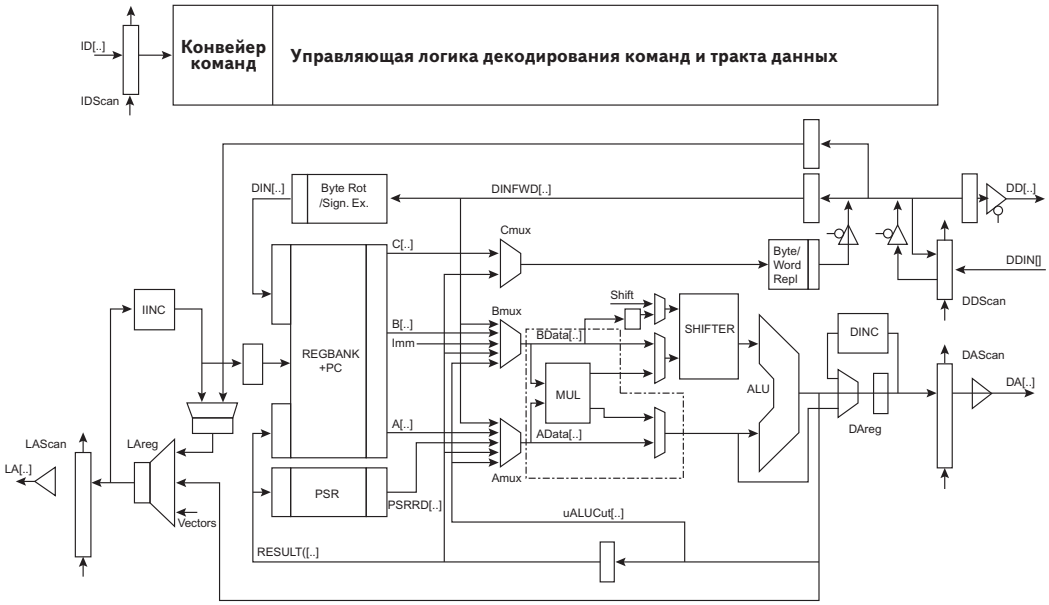


Рис. 2.70. Блок-схема процессора ARM9
(печатается с разрешения ARM. © 1999 ARM Ltd.)

В набор команд ARMv7 были добавлены команды Advanced SIMD, работающие с регистрами двойных и четверных слов. Был также определен вариант v7-M, поддерживающий только команды Thumb. ARM разработала семейства процессоров Cortex-A и Cortex-M. Высокопроизводительные процессоры семейства Cortex-A сейчас используются практически во всех смартфонах и планшетах. Семейство Cortex-M с набором команд Thumb – это очень небольшие и недорогие микроконтроллеры, применяемые во встраиваемых системах. Например, в изделии Cortex-M0+ реализован двухстадийный конвейер и имеется всего 12 000 логических вентилей, а не сотни тысяч, как в процессорах серии A. В виде автономной микросхемы он стоит меньше одного доллара, а при интеграции в объем-

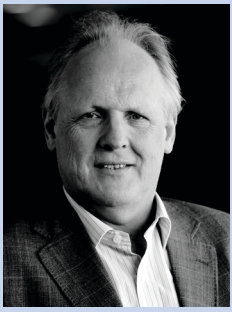
лющую систему на кристалле (SoC) обходится дешевле одного пенса. Энергопотребление составляет приблизительно 3 мкВт/МГц, поэтому процессор, питающийся от батарейки для часов, может непрерывно работать примерно год на частоте 10 МГц.

Процессоры ARMv7 более высокого ценового уровня заняли рынок сотовых телефонов и планшетов. Cortex-A9 широко использовался в мобильных телефонах, часто в составе двухъядерной SoC-системы, содержавшей два процессора Cortex-A9, графический ускоритель, сотовый модем и другие периферийные устройства. На **Рис. 2.71** показана блок-схема Cortex-A9. Процессор декодирует две команды в одном такте, производит переименование регистров и запускает команды на выполнение исполнительными блоками с внеочередным выполнением.

Для мобильных устройств критичны как энергоэффективность, так и производительность, поэтому компания ARM активно продвигала архитектуру big.LITTLE, сочетающую высокую производительность «больших» ядер, работающих в режиме пиковой нагрузки, с энергоэффективностью «малых» ядер, занятых большинством рутинных процессов. Например, микросхема Samsung Exynos 5 Octa, установленная в телефоне Galaxy S5, содержит четыре больших ядра Cortex-A15, работающих на частоте до 2.1 ГГц, и четыре малых ядра Cortex-A7, работающих на частоте до 1.5 ГГц. На **Рис. 2.72** показаны конвейерные диаграммы ядер того и другого типа. Cortex-A7 – процессор с последовательным выполнением команд, способный декодировать и запускать не более одной команды доступа к памяти и еще одной команды в каждом такте. Cortex-A15 – гораздо более сложный процессор с внеочередным выполнением команд, способный декодировать до трех команд в каждом такте. Чтобы справиться с такой сложностью и повысить таковую частоту, длина конвейера почти удвоена, поэтому необходим более точный предсказатель переходов, который мог бы компенсировать увеличение штрафа за неправильное предсказание перехода. Cortex-A15 приблизительно в 2.5 раза производительнее Cortex-A7, но ценой шестикратного увеличения энергопотребления. Смартфон может задействовать большие ядра только кратковременно, а затем кристалл начинает перегреваться и возвращается в более щадящий режим.

ARMv8 – модернизированная 64-битовая архитектура. В процессорах Cortex-A53 и A57 конвейеры похожи на реализованные в Cortex-A7 и A15 соответственно, но регистры и тракт данных расширены до 64 бит. Компания Apple популяризировала 64-битовую архитектуру в 2013 году, когда положила ее в основу собственных реализаций в iPhone и iPad.

Стив Фэрбер (1953–) родился в Манчестере, Англия, получил докторскую степень по аэродинамике в Кэмбриджском университете. Поступил на работу в компанию Acorn Computer, где принимал участие в проектировании компьютера BBC Micro и микропроцессора ARM1. В 1990 г. перешел на работу в Манчестерский университет, где занимался исследованиями в области асинхронных вычислений и нейронных систем.



Фотография © Манчестерского университета (печатается с разрешения правообладателя)

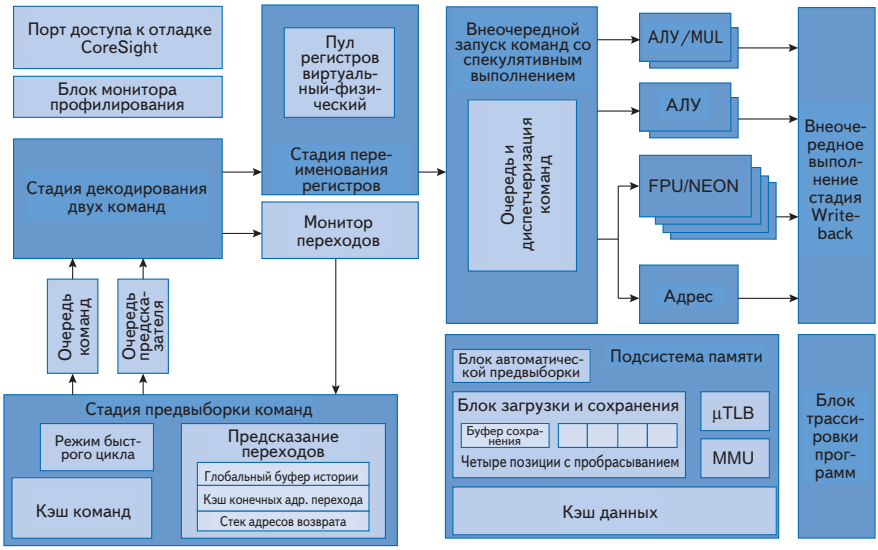


Рис. 2.71. Блок-схема процессора Cortex-A9
(изображение подготовлено авторами)

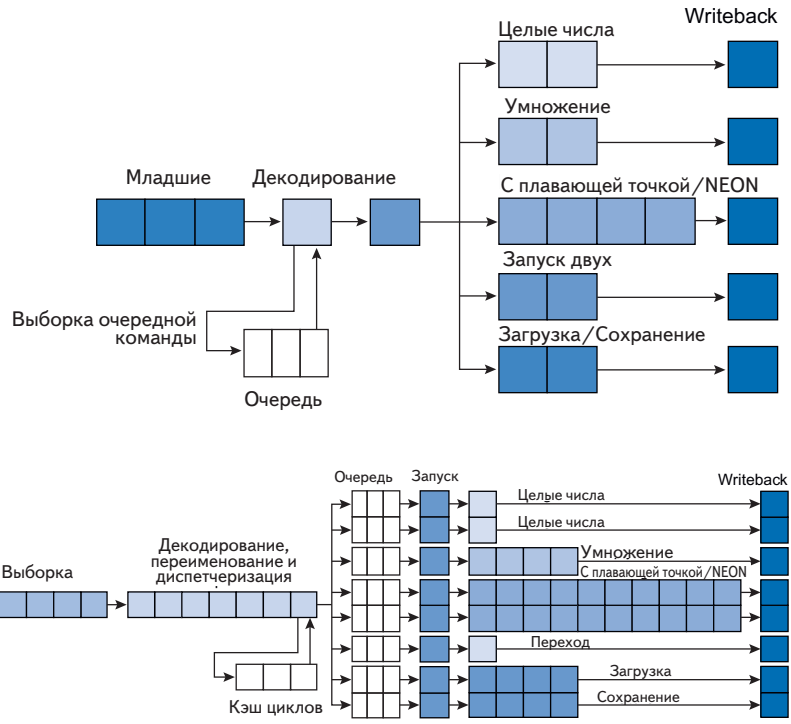


Рис. 2.72. Блок-схемы процессоров Cortex-A7 и Cortex-A15
(изображения подготовлены авторами)

2.9. Резюме

В этой главе мы рассмотрели три способа построения процессоров, отличающихся разным соотношением цены и стоимости. Мы считаем, что тут есть какая-то магия – как может столь сложное на вид устройство, как микропроцессор, оказаться настолько простым, что его схема занимает всего полстраницы? Более того, принцип его работы, такой таинственный для непосвященных, на поверку оказывается довольно очевидным.

В микроархитектурах собрались воедино почти все темы, которые мы рассматривали до сих пор. Чтобы собрать микроархитектурный пазл, нам понадобились знания почти из всех предыдущих глав. Мы проектировали комбинационные и последовательные схемы (**главы 2** и **3** (книга 1)), применяли строительные блоки (**глава 5** (книга 1)) и воплощали в жизнь архитектуру ARM, описанную здесь в **главе 1**. Используя методы, описанные в **главе 4** (книга 1), мы смогли описать микроархитектуру всего на нескольких страницах HDL-кода.

Разработка различных вариантов микроархитектур также потребовала от нас применения принципов управления сложностью. Абстракция микроархитектуры перекидывает мостик между логическим и архитектурным уровнями абстракции и представляет собой квинтэссенцию этой книги о проектировании цифровых систем и компьютерной архитектуре. Мы также использовали абстракции уровня блочных диаграмм и языков описания аппаратуры, чтобы в сжатой форме представить организацию компонентов. При разработке микроархитектур мы широко применяли принципы повторяемости и модульности, повторно используя библиотеки таких часто используемых строительных блоков, как АЛУ, блоки памяти, мультиплексоры и регистры. Мы активно использовали иерархическую организацию, разделив микроархитектуру на тракт данных и устройство управления, которые создали из функциональных блоков, а те, в свою очередь, – из логических элементов, а логические элементы – из транзисторов, как было описано в первых пяти главах.

В этой главе мы сравнили однотактную, многотактную и конвейерную микроархитектуры процессора ARM. Все они реализуют одно и то же подмножество набора команд ARM и имеют одинаковое архитектурное состояние. Однотактный процессор самый простой, каждая его команда выполняется за один такт, т. е. CPI для него равен 1.

Многотактный процессор разбивает выполнение команд на более короткие этапы, число которых переменно. Таким образом, он может использовать одно-единственное АЛУ вместо нескольких сумматоров. Однако ему требуется несколько неархитектурных регистров для хранения промежуточных результатов вычислений между этапами. В теории многотактный процессор мог бы быть быстрее, т. к. не все команды выполняются за одинаковое время. На практике, однако, он обычно медленнее, так как длительность его такта ограничена длительностью самого

медленного этапа, на которую, в свою очередь, негативно влияют накладные расходы, связанные с использованием неархитектурных регистров.

Конвейерный процессор разделяет однотактный процессор на пять относительно быстрых стадий. Для этого между его стадиями добавляют конвейерные регистры, что позволяет изолировать друг от друга пять одновременно выполняющихся команд. В идеальном мире его CPI был бы равен 1, но конфликты в конвейере приводят к необходимости периодически приостанавливать и очищать его, что увеличивает CPI. Логика, необходимая для разрешения конфликтов, также увеличивает сложность процессора. Период тактового сигнала мог бы быть в пять раз меньше, чем у однотактного процессора, но на практике он далеко не так мал, потому что ограничен скоростью работы самой медленной стадии, а также накладными расходами на организацию конвейерных регистров. Тем не менее конвейерная обработка обеспечивает существенное увеличение производительности, поэтому она используется во всех современных высокопроизводительных микропроцессорах.

Хотя микроархитектуры, рассмотренные в этой главе, реализуют только ограниченное подмножество архитектуры ARM, мы показали, что добавление новых команд требует внесения весьма простых и понятных изменений в тракт данных и устройство управления.

Существенным ограничением этой главы является то, что мы считали подсистему памяти идеальной, обеспечивающей быстрый доступ и достаточно большой для хранения всей программы и данных. В реальности же большая и быстрая память чрезмерно дорога. В следующей главе мы покажем, как получить большинство преимуществ, характерных для большой и быстрой памяти, имея только небольшую, но быструю память, в которой хранится лишь самая часто используемая информация, а также медленную, но большую память, в которой хранится все остальное.

Упражнения

Упражнение 2.1. Предположим, что один из перечисленных ниже управляющих сигналов в однотактном процессоре ARM неисправен и постоянно равен нулю, даже когда должен быть равен единице (stuck-at-0 fault). Какие команды перестанут корректно работать? Почему?

a) $RegW$
b) $ALUOp$
c) $MemW$

Упражнение 2.2. Повторите **упражнение 2.1** для случая, когда неисправный сигнал постоянно равен единице (stuck-at-1 fault).

Упражнение 2.3. Модифицируйте однотактный процессор ARM, так чтобы он поддерживал одну из перечисленных ниже команд. Описа-

ние команд см. в **приложении А**. Сделайте копию **Рис. 2.13** и отметьте необходимые изменения в тракте данных. Придумайте названия для новых управляющих сигналов. Скопируйте **Табл. 2.2** и **Табл. 2.3** и покажите все необходимые изменения в основном декодере и декодере АЛУ. Опишите другие необходимые изменения.

a) `TST`
b) `LSL` (с непосредственно заданным значением сдвига)
c) `CMN`
d) `ADC`

Упражнение 2.4. Повторите **упражнение 2.3** для следующих команд:

a) `EOR`
b) `LSR` (с непосредственно заданным значением сдвига)
c) `TEQ`
d) `RSB`

Упражнение 2.5. В ARM имеется команда `LDR` с постиндексацией, которая обновляет базовый регистр после завершения загрузки. Команда `LDR Rd,[Rn],Rm` эквивалентна двум командам:

```
LDR Rd,[Rn]
ADD Rn,Rn,Rm
```

Повторите **упражнение 2.3** для команды `LDR` с постиндексацией. Возможно ли добавить эту команду, не внося изменений в регистровый файл?

Упражнение 2.6. В ARM имеется команда `LDR` с предындексацией, которая обновляет базовый регистр после завершения загрузки. Команда `LDR Rd,[Rn,Rm]!` эквивалентна двум командам:

```
LDR Rd,[Rn, Rm]
ADD Rn,Rn,Rm
```

Повторите **упражнение 2.3** для команды `LDR` с предындексацией. Возможно ли добавить эту команду, не внося изменений в регистровый файл?

Упражнение 2.7. Ваша подруга – гуру схемотехники – предложила переделать один из блоков однотактного процессора ARM, так чтобы задержка этого блока уменьшилась вдвое. Используя значения задержек из **Табл. 2.5**, определите, какой блок ей стоит улучшить, чтобы эффект, оказанный на производительность процессора, оказался наибольшим. Какова в этом случае будет длительность такта процессора?

Упражнение 2.8. Взгляните на задержки в **Табл. 2.5**. Бен Битдиддл разрабатывает префиксный сумматор, уменьшающий задержку АЛУ на 20 пс. Считая, что задержки всех остальных элементов остаются неиз-

менными, определите новую длительность такта однотактного процессора ARM. Определите, сколько времени займет выполнение эталонного теста, содержащего 100 миллиардов команд.

Упражнение 2.9. Измените HDL-код однотактного процессора ARM, приведенный в **разделе 2.6.1**, добавив поддержку одной из команд из **упражнения 2.3**. Дополните тестовое окружение, приведенное в **разделе 2.6.3**, чтобы убедиться, что новая команда работает корректно.

Упражнение 2.10. Повторите **упражнение 2.9** для команд из **упражнения 2.4**.

Упражнение 2.11. Предположим, что один из перечисленных ниже управляющих сигналов в многотактном процессоре ARM неисправен и постоянно равен нулю, даже когда должен быть равен единице (stuck-at-0 fault). Какие команды перестанут корректно работать? Почему?

a) *RegSrc1*
b) *AdrSrc*
c) *NextPC*

Упражнение 2.12. Повторите **упражнение 2.11** для случая, когда неисправный сигнал постоянно равен единице (stuck-at-1 fault).

Упражнение 2.13. Модифицируйте многотактный процессор ARM, так чтобы он поддерживал одну из перечисленных ниже команд. Описание команд см. в **приложении А**. Сделайте копию **Рис. 2.30** и отметьте необходимые изменения в тракте данных. Придумайте названия для новых управляющих сигналов. Сделайте копию **Рис. 2.41** и покажите все необходимые изменения в управляющем конечном автомате. Опишите другие необходимые изменения.

a) `ASR` (с непосредственно заданным значением сдвига)
b) `TST`
c) `SBC`
d) `ROR` (с непосредственно заданным значением сдвига)

Упражнение 2.14. Повторите **упражнение 2.13** для следующих команд.

a) `BL`
b) `LDR` (с положительным или отрицательным непосредственным смещением)
c) `LDRB` (только с положительным непосредственным смещением)
d) `BIC`

Упражнение 2.15. Повторите **упражнение 2.5** для многотактного процессора ARM. Опишите все необходимые изменения в многотактном тракте данных и управляющем конечном автомате. Возможно ли добавить эту команду, не внося изменений в регистровый файл?

Упражнение 2.16. Повторите **упражнение 2.6** для многотактного процессора ARM. Опишите все необходимые изменения в многотактном тракте данных и управляющем конечном автомате. Возможно ли добавить эту команду, не внося изменений в регистровый файл?

Упражнение 2.17. Повторите **упражнение 2.7** для многотактного процессора ARM. Считайте, что процентное соотношение разных типов команд такое же, как в **примере 2.5**.

Упражнение 2.18. Повторите **упражнение 2.8** для многотактного процессора ARM. Считайте, что процентное соотношение разных типов команд такое же, как в **примере 2.5**.

Упражнение 2.19. Ваша подруга – гуру схемотехники – предложила переделать один из блоков многотактного процессора ARM, так чтобы задержка этого блока существенно уменьшилась. Используя значения задержек из **Табл. 2.5**, определите, какой блок ей стоит улучшить, чтобы эффект, оказанный на производительность процессора, оказался наибольшим. Как быстро должен работать новый блок? Учтите, что попытки сделать его быстрее, чем необходимо, – напрасная трата времени вашей подруги. Какова будет длительность такта процессора?

Упражнение 2.20. Корпорация «Голиаф» заявила, что запатентовала трехпортовый регистровый файл. Вместо того чтобы судиться с ней, Бен Битдиддл разработал новый регистровый файл, у которого всего один порт чтения/записи (как у объединенной памяти команд и данных). Переделайте многотактовый тракт данных и устройство управления, так чтобы они использовали новый регистровый файл.

Упражнение 2.21. Предположим, что задержки компонентов многотактного процессора ARM такие, как в **Табл. 2.5**. Алиса П. Хакер разработала новый регистровый файл, который потребляет на 40% меньше электроэнергии, но при этом работает в два раза медленнее. Стоит ли ей для своего многотактного процессора выбрать более медленный, но потребляющий меньше энергии регистровый файл?

Упражнение 2.22. Чему равно CPI переделанного многотактного процессора ARM из **упражнения 2.20**? Считайте, что процентное соотношение разных типов команд такое же, как в **примере 2.5**.

Упражнение 2.23. Сколько тактов потребуется, чтобы выполнить следующую программу на многотактном процессоре ARM? Чему равно CPI для этой программы?

```
    MOV R0, #5      ; result = 5
    MOV R1, #0      ; R1 = 0
L1
    CMP R0, R1
    BEQ DONE        ; если result > 0, повторить цикл
```

```
    SUB R0, R0, #1 ; result = result-1
    B   L1
DONE
```

Упражнение 2.24. Повторите **упражнение 2.23** для следующей программы:

```
    MOV R0, #0     ; i = 0
    MOV R1, #0     ; sum = 0
    MOV R2, #10    ; R2 = 10
LOOP
    CMP R2, R0     ; R2 == R0?
    BEQ L2
    ADD R1, R1, R0 ; sum = sum + i
    ADD R0, R0, #1 ; увеличить i
    B   LOOP
L2
```

Упражнение 2.25. Напишите HDL-код многотактного процессора ARM. Процессор должен быть совместим с модулем верхнего уровня, приведенным ниже. Модуль `mem` используется для хранения и команд, и данных. Протестируйте ваш процессор, используя тестовое окружение из **раздела 2.6.3**.

```
module top(input  logic        clk, reset,
           output logic [31:0] WriteData, Adr,
           output logic        MemWrite);
  logic [31:0] ReadData;

  // определить процессор и общую память
  arm arm(clk, reset, MemWrite, Adr,
          WriteData, ReadData);
  mem mem(clk, MemWrite, Adr, WriteData, ReadData);
endmodule

module mem(input  logic        clk, we,
           input  logic [31:0] a, wd,
           output logic [31:0] rd);
  logic [31:0] RAM[63:0];
  initial
    $readmemh("memfile.dat",RAM);
    assign rd = RAM[a[31:2]]; // выровнять на границу слова

    always_ff @(posedge clk)
      if (we) RAM[a[31:2]] <= wd;
endmodule
```

Упражнение 2.26. Добавьте в HDL-код многотактного процессора ARM из **упражнения 2.25** поддержку одной из новых команд из

упражнения 2.14. Дополните тестовое окружение, чтобы убедиться, что новая команда работает корректно.

Упражнение 2.27. Повторите **упражнение 2.26** для одной из команд из **упражнения 2.13**.

Упражнение 2.28. Конвейерный процессор ARM выполняет приведенную ниже программу. Какие регистры он читает и в какие регистры пишет на пятом такте? Не забудьте, что в конвейерном процессоре ARM имеется блок обнаружения и разрешения конфликтов.

```
MOV R1, #42
SUB R0, R1, #5
LDR R3, [R0, #18]
STR R4, [R1, #63]
ORR R2, R0, R3
```

Упражнение 2.29. Повторите **упражнение 2.28** для приведенной ниже программы.

```
ADD R0, R4, R5
SUB R1, R6, R7
AND R2, R0, R1
ORR R3, R2, R5
LSL R4, R2, R3
```

Упражнение 2.30. Используя такую же диаграмму, как на **Рис. 2.53**, покажите пробрасывания через байпас и приостановки конвейера, необходимые для выполнения следующих команд конвейерным процессором ARM.

```
ADD R0, R4, R9
SUB R0, R0, R2
LDR R1, [R0, #60]
AND R2, R1, R0
```

Упражнение 2.31. Повторите **упражнение 2.30** для следующих команд:

```
ADD R0, R11, R5
LDR R2, [R1, #45]
SUB R5, R0, R2
AND R5, R2, R5
```

Упражнение 2.32. Сколько тактов потребуется конвейерному процессору ARM, чтобы запустить на выполнение все команды программы из **упражнения 2.23**? Чему равно CPI процессора для этой программы?

Упражнение 2.33. Повторите **упражнение 2.32** для программы из **упражнения 2.23**.

Упражнение 2.34. Объясните, как добавить в конвейерный процессор ARM поддержку команды `EOR`.

Упражнение 2.35. Объясните, как добавить в конвейерный процессор ARM поддержку команды `CMN`.

Упражнение 2.36. В **разделе 2.5.3** отмечено, что производительность конвейерного процессора ARM могла бы быть выше, если бы вычисление условия перехода выполнялось бы на стадии *Decode*, а не *Execute*. Покажите, как модифицировать конвейерный процессор, показанный на **Рис. 2.58**, чтобы вычисление условия перехода выполнялось на стадии *Decode*. Как изменятся сигналы приостановки, очистки и пробрасывания? Вычислите новое значение CPI, длительность такта процессора и общее время выполнения программы в **примерах 2.7** и **2.8**.

Упражнение 2.37. Ваша подруга – гуру схемотехники – предложила переделать один из блоков конвейерного процессора ARM, так чтобы задержка этого блока существенно уменьшилась. Используя значения задержек из **Табл. 2.5**, определите, какой блок ей стоит улучшить, чтобы эффект, оказанный на производительность процессора, оказался наибольшим. Как быстро должен работать новый блок? Учтите, что попытки сделать его быстрее, чем необходимо, – напрасная трата времени вашей подруги. Какова будет длительность такта улучшенного процессора?

Упражнение 2.38. Взгляните на задержки в **Табл. 2.5**. Изменилась бы длительность такта конвейерного процессора ARM, если бы АЛУ работало на 20% быстрее? А если на 20% медленнее?

Упражнение 2.39. Предположим, что конвейерный процессор ARM поделен на 10 стадий длительностью 400 пс каждая, включая все накладные расходы на конвейеризацию. Будем считать, что процентное соотношение команд разных типов такое, как в **примере 2.7**, при этом в половине случаев результат команд загрузки требуется немедленно, что приводит к шести приостановкам конвейера. Также будем считать, что 30% условных переходов предсказано неверно, а адрес перехода в командах перехода вычисляется в конце второго такта. Вычислите для этого десятистадийного процессора среднее значение CPI и время выполнения 100 миллиардов команд из теста SPECINT2000.

Упражнение 2.40. Напишите HDL-код конвейерного процессора ARM. Процессор должен быть совместим с модулем верхнего уровня из HDL в **примере 2.13**. Он должен поддерживать все команды, рассмотренные в этой главе: `ADD`, `SUB`, `AND`, `ORR` (с регистровой и непосредственной адресацией, но без сдвига), `LDR`, `STR` (с положительным непосредствен-

ным смещением) и B. Протестируйте ваш процессор, используя тестовое окружение из **примера 2.12**.

Упражнение 2.41. Разработайте для конвейерного процессора ARM блок обнаружения и разрешения конфликтов, показанный на **Рис. 2.58**. Используйте язык описания аппаратуры. Изобразите в общих чертах схему, которую мог бы сгенерировать из вашего кода HDL-синтезатор.

Вопросы для собеседования

Приведенные ниже вопросы задавали на собеседованиях с кандидатами на вакансии разработчиков цифровой аппаратуры.

Вопрос 2.1. Объясните преимущества конвейерных микропроцессоров.

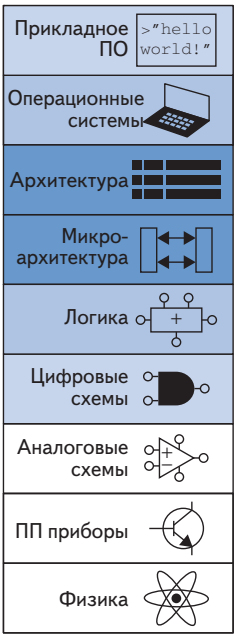
Вопрос 2.2. Если большее количество стадий конвейера позволяет процессору работать быстрее, почему нет процессоров с сотней стадий?

Вопрос 2.3. Объясните, что такое конфликты в микропроцессоре и каковы пути их разрешения. Каковы преимущества и недостатки каждого способа?

Вопрос 2.4. Расскажите, что такое суперскалярный процессор, каковы его достоинства и недостатки.

Глава 3

Подсистема памяти



3.1. Введение

Производительность компьютерной системы зависит от ее подсистемы памяти так же сильно, как и от микроархитектуры процессора. В **главе 2** мы считали, что память идеальна и к ней можно обратиться всего за один такт. Однако это было бы правдой только для очень маленькой памяти (или для очень медленного процессора)! Ранние процессоры были сравнительно медленные, так что память успевала за процессором. Но скорость процессоров росла быстрее, чем скорость памяти. В настоящее время оперативная память типа DRAM (Dynamic Random Access Memory, динамическое запоминающее устройство с произвольным доступом, ДЗУПВ) медленнее процессора в 10–100 раз. Увеличивающийся разрыв требует все более и более изощренных подсистем памяти, чтобы попытаться приблизить скорость работы памяти к скорости процессора. В этой главе мы расскажем о подсистемах памяти и проанализируем различные компромиссы между их скоростью, емкостью и стоимостью.

Процессор работает с памятью через *интерфейс памяти*. На **Рис. 3.1** показан простой интерфейс к памяти, использованный в нашем многотактном процессоре ARM. Процессор помещает адреса на шину адреса (*Address*), идущую к подсистеме памяти. Для чтения управляющий

сигнал записи (*MemWrite*) устанавливается в 0, а память возвращает данные по шине чтения данных (*ReadData*). Для записи *MemWrite* устанавливается в 1, и процессор посылает данные в память по шине записи данных (*WriteData*).

Основные проблемы при разработке подсистемы памяти можно описать, рассматривая в качестве метафоры книги в библиотеке. На библиотечных полках стоит много книг. Собравшись написать курсовую работу по толкованию снов, вы можете пойти в библиотеку[1], взять с полки книгу Фрейда «Толкование сновидений» и принести ее к себе в комнату. Просмотрев книгу, вы можете вернуть ее обратно и взять работу Юнга «Психология бессознательного». Затем вы могли бы пойти обратно за «Толкованием сновидений», чтобы использовать еще одну цитату из нее. А потом – за книгой Фрейда «Я и Оно». Очень скоро вы устанете бегать в библиотеку, и если не обделены разумом, то будете просто держать нужные книги у себя в комнате, вместо того чтобы таскаться за ними взад-вперед. Более того, взяв книгу Фрейда, вы можете захватить еще несколько его книг с той же полки (на всякий случай).

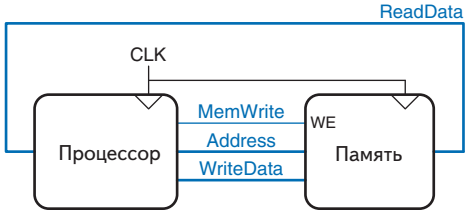


Рис. 3.1. Интерфейс памяти

Эта метафора подчеркивает принцип, изложенный в **разделе 1.2.1**, – «типичный сценарий должен быть быстрым». Оставляя в своей комнате книги, которые вы только что использовали или которые, возможно, понадобятся в ближайшем будущем, вы уменьшаете количество отнимающих много времени походов в библиотеку. В частности, вы используете принципы *временной* (temporal) и *пространственной* (spatial) *локальности*. Временная локальность означает, что если вы только что использовали книгу, то, вероятно, она вам снова скоро понадобится. Пространственная локальность означает, что если вам понадобилась определенная книга, то, вероятно, вас заинтересуют и другие книги с той же полки.

Библиотека сама старается ускорить типичный сценарий, используя принцип локальности. У нее нет ни места на полках, ни денег, чтобы собрать все книги в мире. Вместо этого редко спрашиваемые книги хранят-

[1] Мы понимаем, что в век Интернета студенты все реже пользуются библиотеками. Однако мы верим, что в библиотеках хранятся сокровища знаний, которые достались человечеству тяжелым трудом, и не все они доступны в электронном виде. Мы надеемся, что искусство поиска знаний в книгах не будет полностью вытеснено запросами к Всемирной паутине.

ся в подвале. Также используется система межбиблиотечного обмена с соседними библиотеками, так что библиотека может предложить вам больше книг, чем физически имеется в наличии.

В итоге вы получаете выгоду как от большого собрания книг, так и от быстрого доступа к наиболее популярным книгам – благодаря иерархической системе хранения книг. Книги, с которыми вы работаете, находятся у вас на столе. Большое собрание – на полках местной библиотеки. Еще большее собрание имеется в хранилище и в других библиотеках – оно доступно по предварительному запросу. Точно так же в подсистемах памяти используется иерархия хранилищ, которая обеспечивает быстрый доступ к наиболее часто используемым данным и одновременно возможность хранения больших объемов данных. Подсистемы памяти, применяемые для создания такой иерархии, были описаны в **разделе 5.5** (книга 1). Компьютерная память в основном построена на базе динамической (DRAM) и статической (SRAM) памяти. В идеале память должна быть быстрой, большой и дешевой. Однако на практике любой тип памяти обладает только двумя из этих свойств; память либо медленная, либо дорогая, либо маленького объема. Несмотря на это, компьютерные системы могут приближаться к идеалу, сочетая дешевую, быструю и маленькую память с дешевой, медленной и большой. Быстрая память используется для хранения часто используемых данных и команд, так что создается впечатление, что подсистема памяти всегда работает быстро. Остальные данные и команды хранятся в большой памяти, которая работает медленнее, но располагает большой емкостью. Комбинация двух дешевых типов памяти – это куда менее дорогой вариант, чем одна большая и быстрая память. Этот принцип распространяется на всю иерархию памяти, так как с увеличением объема памяти уменьшается ее скорость работы.

Память компьютера обычно строится на микросхемах динамической памяти (DRAM). В 2015 году типичный персональный компьютер имел *оперативную память* (main memory) объемом от 8 до 16 ГБ, и эта DRAM-память стоила около семи долларов за гигабайт. Цены на DRAM падали в среднем на 25% в год на протяжении последних тридцати лет, при этом емкость памяти росла примерно с такой же скоростью, так что общая цена памяти в персональном компьютере оставалась приблизительно одинаковой. К сожалению, скорость работы самих микросхем DRAM возрастала только на 7% в год, в то время как производительность процессоров – на 25–50% в год. На **Рис. 3.2** показан график увеличения скорости работы оперативной памяти и процессоров с 1980 года по настоящее время. В начале 1980-х годов скорость процессоров и памяти была примерно одинаковой, но затем разрыв в производительности сильно увеличился и память серьезно отстала[2].

[2] Как видно по Рис. 3.2, производительность одного процессорного ядра оставалась примерно одинаковой с 2005 по 2010 год, но переход на многоядерные системы (на рисунке не показан) только усугубляет разрыв в производительности процессоров и памяти.

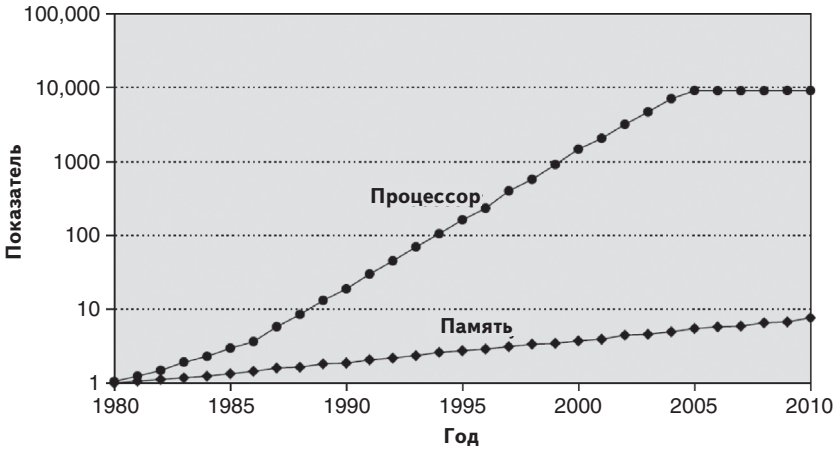


Рис. 3.2. Разрыв в производительности процессоров и памяти
График заимствован из книги Hennessy and Patterson «Computer Architecture: A Quantitative Approach», издание 5, Morgan Kaufmann, 2012 (печатается с разрешения авторов)

Память DRAM могла успешно идти в ногу с процессорами в 1970-х и в начале 1980-х годов, но сейчас она чудовищно медленная. Время доступа к DRAM на порядок или два больше длительности такта процессора (десятки наносекунд против долей наносекунды).

Чтобы справиться с этой проблемой, компьютеры хранят наиболее часто используемые команды и данные в быстрой, но небольшой по объему *кэш-памяти*, или просто *кэше*. Кэш-память обычно построена на базе статической памяти (SRAM) и находится на той же микросхеме, что и процессор. Скорость кэша сравнима со скоростью процессора, так как, во-первых, память SRAM работает быстрее, чем DRAM, а во-вторых, расположение на одном кристалле с процессором позволяет избавиться от задержек распространения сигналов по пути к внешним микросхемам памяти. В 2015 году стоимость SRAM, расположенной на кристалле процессора, составляла порядка 5 тысяч долларов за ГБ, но так как размер кэша сравнительно мал (от нескольких килобайт до нескольких мегабайт), то общая стоимость кэша не столь уж велика. В кэш-памяти можно хранить как данные, так и команды, но для краткости мы будем говорить, что она содержит просто «данные».

Если процессор запрашивает данные, которые уже находятся в кэше, то он получает их очень быстро. Это называется *попаданием в кэш* (cache hit). В противном случае процессор вынужден читать данные из оперативной памяти (DRAM). Это называется *промахом кэша*, кэш-промахом или промахом доступа в кэш (cache miss). Если процессор попадает в кэш бóльшую часть времени, то он редко простаивает в ожидании доступа к медленной оперативной памяти, и среднее время доступа мало.

Третий уровень в иерархии памяти – жесткий диск. По аналогии с использованием библиотекой подвала для хранения книг, не умещающихся на полках, компьютерные системы используют жесткий диск для хранения данных, которые не помещаются в оперативной памяти. В 2015 году жесткий диск, выполненный по технологии магнитной записи (Hard Disk Drive, HDD), стоил менее пяти центов за ГБ и имел время доступа около 5 миллисекунд. Цены на жесткие диски падают на 60% в год, но время доступа почти не улучшается. Твердотельные диски (Solid State Drive, SSD), которые сделаны на основе флеш-памяти, становятся все более популярной альтернативой HDD. SSD использовались для специальных применений на протяжении двух десятилетий, но на потребительский рынок вышли только в 2007 году. Они не подвержены механическим отказам, но и стоят в десять раз больше, чем HDD, – порядка сорока центов за ГБ .

Жесткий диск обеспечивает иллюзию наличия бóльшего объема памяти, чем реально доступно в оперативной памяти. Это называется виртуальной памятью. Как и доступ к книгам в хранилище, доступ к данным в виртуальной памяти занимает длительное время. Оперативная память, также называемая физической памятью, содержит только часть данных, находящихся в виртуальной памяти, остальные находятся на жестком диске. Следовательно, оперативная память может рассматриваться как кэш-память для наиболее часто используемых данных с жесткого диска.

В этой главе мы рассмотрим иерархию памяти компьютерной системы, показанную на **Рис. 3.3**. Процессор сначала ищет данные в маленькой, но быстрой кэш-памяти, обычно расположенной на том же кристалле. Если данные в кэше отсутствуют, процессор обращается к оперативной памяти. Если данных нет и там, то процессор читает данные с большого, хотя и медленного, жесткого диска, применяя механизм виртуальной памяти. **Рисунок 3.4** иллюстрирует соотношение емкости и скорости в многоуровневой иерархии памяти компьютерной системы и показывает типичную стоимость, время доступа и пропускную способность для технологий памяти по состоянию на 2015 год. Как видите, с уменьшением времени доступа скорость возрастает.

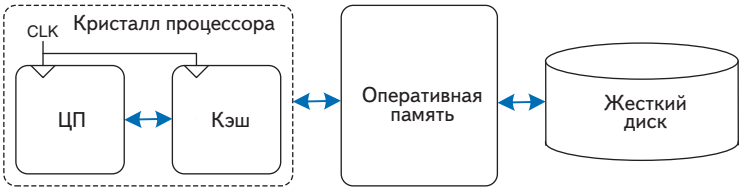
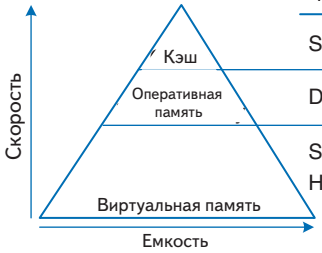


Рис. 3.3. Типичная иерархия памяти



Технология	Цена за ГБ	Время доступа (нс)	Пропускная способность (ГБ/с)
SRAM	$5,000	0.5	25+
DRAM	$7	10–50	10
SSD	$0.40	20,000	0.5
HDD	$0.05	5,000,000	0.75

Рис. 3.4. Компоненты иерархии памяти и их характеристики по состоянию на 2015 год

В **разделе 3.2** мы покажем, как анализировать производительность систем памяти. В **разделе 3.3** рассматривается несколько методов организации кэш-памяти, а **раздел 3.4** посвящен виртуальной памяти.

3.2. Анализ производительности подсистемы памяти

Чтобы оценить соотношение цены и производительности для разных вариантов подсистемы памяти, разработчикам (и покупателям) компьютеров нужны количественные способы измерения производительности. Мерами измерения производительности систем памяти являются *процент попаданий* (hit rate) или *промахов* (miss rate), а также *среднее время доступа*. Процент попаданий и промахов вычисляется по формулам:

$$\textit{Доля промахов} = \frac{\textit{Число промахов}}{\textit{Общее число доступов к памяти}} = 1 - \textit{Доля попаданий},$$
$$\textit{Доля попаданий} = \frac{\textit{Число попаданий}}{\textit{Общее число доступов к памяти}} = 1 - \textit{Доля промахов}. \quad (3.1)$$

Пример 3.1. ВЫЧИСЛЕНИЕ ПРОИЗВОДИТЕЛЬНОСТИ КЭШ-ПАМЯТИ

Предположим, что в программе имеется 2000 команд обращения к данным (загрузки и сохранения), но только 1250 из этих команд нашли запрошенные ими данные в кэш-памяти. Остальным 750 командам пришлось получать данные из оперативной памяти или с диска. Чему равен процент промахов и попаданий в кэш-память в этом случае?

Решение. Процент промахов находится как $750/2000 = 0.375 = 37.5\%$. Процент попаданий равен $1250/2000 = 0.625 = 1 \times 0.375 = 62.5\%$.

Среднее время доступа (average memory access time, AMAT) определяется как среднее время, которое процессор тратит в ожидании доступа к памяти при выполнении команд загрузки или сохранения данных. В типичной компьютерной системе, показанной на **Рис. 3.3**, процессор сначала ищет данные в кэше. Если данных в кэше нет, то процессор обращается к оперативной памяти. Если же данных нет и там, то процессор выполняет обращение к виртуальной памяти на диске. Следовательно, AMAT вычисляется по формуле:

$$AMAT = t_{\text{cache}} + MR_{\text{cache}}(t_{MM} + MR_{MM}\, t_{VM}), \qquad (3.2)$$

где t_{cache}, t_{MM} и t_{VM} – времена доступа к кэшу, оперативной памяти и диску соответственно, а MR_{cache} и MR_{MM} – процент промахов кэша и оперативной памяти.

Пример 3.2. ВЫЧИСЛЕНИЕ СРЕДНЕГО ВРЕМЕНИ ДОСТУПА К ПАМЯТИ

Предположим, что в компьютерной системе имеется память всего с двумя уровнями иерархии: кэшем и оперативной памятью. Чему равно среднее время доступа, если время доступа и процент промахов заданы в **Табл. 3.1**?

Решение. Среднее время доступа будет равно 1 + 0.1(100) = 11 тактов.

Таблица 3.1. Время доступа и процент промахов

Уровень памяти	Время доступа в тактах	Процент промахов
Кэш-память	1	10%
Оперативная память	100	0%

Пример 3.3. УЛУЧШЕНИЕ ВРЕМЕНИ ДОСТУПА

Среднее время доступа к памяти в 11 тактов означает, что процессор тратит 10 тактов на ожидание данных и один такт на их использование. Какой процент промахов в кэш необходим для уменьшения среднего времени доступа к памяти до 1.5 такта при заданном в **Табл. 3.1** времени доступа к памяти?

Решение. Обозначим процент промахов в кэш как m, тогда среднее время доступа будет равно 1 + 100m. Вычислим m, приравняв это выражение к 1.5. Ответ: требуемый процент промахов должен быть равен 0.5%.

Джин Амдал (1922–2015)

Джин Амдал известен прежде всего как автор «закона Амдала» – наблюдения, которое он сделал в 1965 году. Будучи аспирантом, Амдал начал в свободное время разрабатывать компьютеры. Эта работа принесла ему докторскую степень по теоретической физике в 1952 году. Сразу после окончания аспирантуры Амдал устроился на работу в IBM, а позже основал три компании, одну из которых в 1970 году назвал Amdahl Corporation.

Cache (кэш) – тайник для хранения орудий труда и продовольствия (*словарь Merriam Webster. Online Dictionary, 2012*, www.merriam-webster.com).

Увы, улучшение производительности в реальности может оказаться не таким радужным, как на бумаге. Например, увеличение быстродействия памяти в десять раз не обязательно сделает компьютерную программу в десять раз быстрее. Если 50% команд в программе – команды загрузки и сохранения данных, то десятикратное увеличение быстродействия памяти приведет к ускорению программы всего лишь в 1.82 раза. Этот общий принцип называется законом Амдала и гласит, что усилия, потраченные на улучшение производительности подсистемы, оправдываются только тогда, когда она оказывает значительное влияние на общую производительность системы.

3.3. Кэш-память

Кэш содержит часто используемые данные из памяти. Число слов данных, которое он может хранить, называется *емкостью* (capacity) кэша. Поскольку емкость кэша меньше, чем емкость оперативной памяти, то проектировщик компьютерной системы должен решить, какую часть оперативной памяти хранить в кэше.

Когда процессор пытается получить доступ к данным, он сначала ищет их в кэше. Если данные там есть, т. е. произошло попадание в кэш, то процессор получает их немедленно. Если же их там нет, то есть произошел промах кэша, то процессор выбирает данные из оперативной памяти и помещает их в кэш для последующего использования. Для этого кэш должен *заместить* какие-то старые данные новыми. В этом разделе мы рассмотрим разработку кэшей и ответим на следующие вопросы: (1) Какие данные хранятся в кэш-памяти? (2) Как найти данные в кэш-памяти? и (3) Какие данные заместить в кэш-памяти, когда требуется место для новых данных, а кэш заполнен?

При чтении последующих разделов помните, что в основе ответов на эти вопросы лежит присущая большинству программ пространственная и временная локальность при обращении к данным. Эту локальность кэш использует для предсказания того, какие данные понадобятся в следующий раз. Если программа обращается к памяти случайным образом, то она не получит никакого выигрыша от использования кэша.

Как мы увидим в следующих разделах, кэш-память характеризуется емкостью C, числом наборов S, длиной строки, иногда называемой размером блока b, количеством строк или блоков B и степенью ассоциативности N.

Хотя мы акцентируем внимание на чтении из кэша данных, те же самые принципы применимы и к чтению из кэша команд. Запись в кэш данных похожа на чтение и будет рассмотрена в **разделе 3.3.4**.

3.3.1. Какие данные хранятся в кэш-памяти?

Идеальный кэш должен предугадывать, какие данные понадобятся процессору, и выбирать их из оперативной памяти заранее таким образом, чтобы процент промахов был равен нулю. Но поскольку точно предсказать будущее невозможно, кэш должен угадывать, какие данные понадобятся, основываясь на предыдущих обращениях к памяти. В частности, чтобы уменьшить процент непопадания в кэш, используется временная и пространственная локальность.

Напомним, что временная локальность означает, что процессор с большой вероятностью еще раз обратится к тем данным, которые недавно использовались. Поэтому, когда процессор читает или записывает данные, отсутствующие в кэше, то эти данные копируются из оперативной памяти в кэш, так что последующие обращения к ним уже не вызовут промаха кэша.

Напомним также, что пространственная локальность означает, что когда процессор обращается к каким-либо данным, то, вероятно, ему понадобятся и расположенные рядом данные. Поэтому, когда кэш читает одно слово данных из памяти, он заодно читает и несколько соседних слов. Эта группа слов называется *строкой кэша* (cache line), иногда также используют термин «*блок кэша*» (cache block). Число слов в строке b называется *длиной строки*. Кэш емкостью C содержит $B = C / b$ строк.

Принципы пространственной и временной локальности данных были экспериментально подтверждены на реальных программах. Если некоторая переменная используется в программе, то она, скорее всего, будет использована снова, что приводит к временной локальности. Если используется какой-либо элемент массива, то, скорее всего, и другие элементы этого массива тоже будут использованы, что ведет к пространственной локальности.

3.3.2. Как найти данные в кэш-памяти?

Кэш состоит из S *наборов*, каждый из которых содержит одну или несколько строк данных. Взаимосвязь между адресом данных в оперативной памяти и расположением этих данных в кэше называется *отображением*. Любой адрес памяти всегда отображается в один и тот же набор кэша. Несколько бит адреса используются, чтобы определить, какой именно набор кэша содержит искомые данные. Если в наборе больше одной строки, то данные могут находиться в любой из них.

Кэш-память классифицируется по числу строк в наборе. В *кэше прямого отображения* (direct mapped cache) каждый набор содержит только одну строку, так что кэш содержит $S = B$ наборов. Таким образом, каждый адрес в оперативной памяти отображается в единственную строку кэша. В случае же *наборно-ассоциативного кэша с N секциями* (N-way set associative cache) каждый набор состоит из N строк. Каждый адрес памяти по-прежнему отображается в единственный набор, но число наборов в этом случае равно $S = B / N$, а данные могут оказаться в любой из N строк этого набора. В отличие от кэша прямого отображения и наборно-ассоциативного кэша, *полностью ассоциативный кэш* (fully associative cache) имеет только один набор ($S = 1$), и данные могут оказаться в любой из B строк этого набора. Таким образом, полностью ассоциативный кэш – то же самое, что и наборно-ассоциативный кэш с B секциями (количество секций совпадает с количеством строк во всем кэше).

Для иллюстрации этих вариантов организации кэша мы рассмотрим подсистему памяти процессора ARM с 32-битовыми адресами и 32-битовыми словами. В наших примерах память адресуется побайтово, а каждое слово состоит из четырех байт, так что память содержит 2^{30} слов, выровненных по границе слова. Для простоты будем рассматривать кэши емкостью $C = 8$ слов. Начнем с длины строки (b), равной одному слову, после чего перейдем к большим по размеру строкам.

Кэш-память прямого отображения

В кэш-памяти прямого отображения каждый набор содержит только одну строку, так что в кэше $S = B$ наборов и строк. Чтобы понять способ отображения адресов памяти на строки такого кэша, представьте, что оперативная память поделена на блоки по b слов так же, как кэш поделен на строки по b слов. Адрес слова, находящегося в блоке 0 оперативной памяти, отображается в набор 0 кэша. Адрес слова из блока 1 оперативной памяти отображается в набор 1 кэша, и так далее вплоть до блока $B - 1$ оперативной памяти, который отображается в строку $B - 1$ кэша. Больше строк в кэше нет, так что следующий блок оперативной памяти (блок B) снова отображается в строку 0 кэша, и т. д.

Отображение для такого кэша емкостью 8 слов и размером строки, равным одному слову, показано на **Рис. 3.5**. В кэше 8 наборов, каждый из которых содержит по одной строке, длина каждой строки равна одному слову. Младшие два бита адреса всегда равны нулю, потому что все адреса выровнены на границу слова. Следующие $\log_2 8 = 3$ бита адресуют набор, в который будет отображен этот адрес памяти. Таким образом, данные по адресам 0x00000004, 0x00000024, ..., 0xFFFFFFE4 отображаются в набор 1, как показано синим цветом. Аналогично данные по адресам 0x00000010, ..., 0xFFFFFFF0 отображаются в набор 4 и т. д. Каждый адрес оперативной памяти отображается строго в один набор кэша.

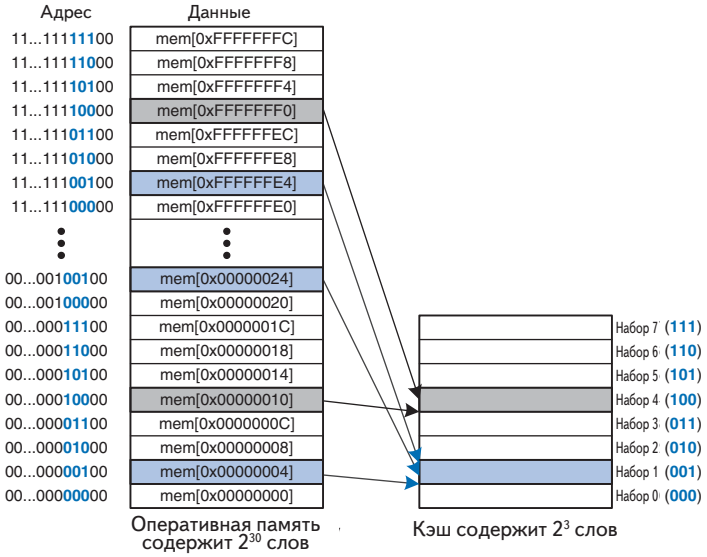


Рис. 3.5. Отображение оперативной памяти на кэш прямого отображения

Пример 3.4. ПОЛЯ АДРЕСА ПРИ ОТОБРАЖЕНИИ В КЭШ

В какой набор кэша на **Рис. 3.5** будет отображено слово с адресом 0x00000014? Назовите другой адрес, который отображается в этот же самый набор.

Решение. Два младших бита адреса всегда равны нулю, потому что адрес выровнен по границе слова. Следующие 3 бита равны 101, так что слово будет отображено в набор 5. Слова с адресами 0x34, 0x54, 0x74, ..., 0xFFFFFFF4 тоже будут отображены в этот набор.

Так как в один набор кэша отображается множество адресов, то кэш должен отслеживать адреса данных, находящихся в каждом наборе в текущий момент времени. Младшие биты адреса определяют набор, в котором хранятся данные. Оставшиеся биты адреса называются *тегом* (tag) и указывают, какой именно из всех возможных адресов сейчас находится в этом наборе.

В наших предыдущих примерах два младших бита адреса называются *байтовым смещением* (byte offset), поскольку указывают на номер байта внутри слова. Следующие три бита называются *индексом* (cache index), или *номером набора* (set bits), так как указывают на номер набора, в который отображается этот адрес (в общем случае номер набора состоит из $\log_2 S$ бит, где S – число наборов). Оставшиеся 27 бит тега указывают на адрес слова, которое в текущий момент находится в этом наборе кэша. На **Рис. 3.6** показано, на какие поля делится адрес 0xFFFFFFE4. Он отображается в набор 1, а его тег содержит одни единицы.

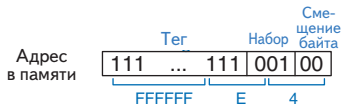


Рис. 3.6. Поля адреса при отображении в кэш

Пример 3.5. ПОЛЯ АДРЕСА ПРИ ОТОБРАЖЕНИИ В КЭШ

Найти число битов тега и номера набора для кэш-памяти прямого отображения с 1024 (2^{10}) наборами и длиной строки, равной одному слову. Размер адреса равен 32 битам.

Решение. Для кэш-памяти с 2^{10} наборами требуется $\log_2(2^{10}) = 10$ бит для хранения номера набора. В двух младших битах адреса хранится байтовое смещение, а оставшиеся $32 - 10 - 2 = 20$ бит используются для тега.

Иногда, особенно когда компьютер только включили, наборы кэша еще не содержат никаких данных. Для каждого набора в кэше есть бит достоверности (valid bit), который равен единице, если в нем находятся корректные данные, и нулю, если находящееся в нем значение бессмысленно.

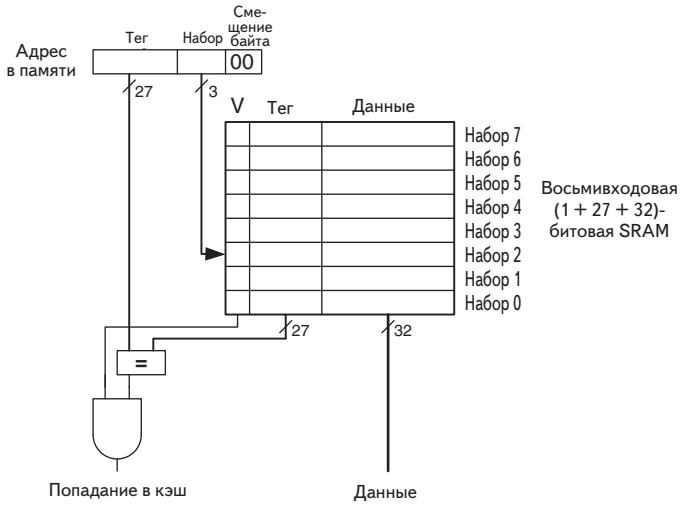


Рис. 3.7. Кэш прямого отображения с восемью наборами

На **Рис. 3.7** изображена блок-схема аппаратной реализации кэша прямого отображения, показанного на **Рис. 3.5**. Кэш представляет собой блок статической памяти SRAM с восемью ячейками. Каждая ячейка, или набор, содержит 32 бита данных, 27 бит тега и 1 бит достоверности (V). К кэшу обращаются, используя 32-битовые адреса. Два младших бита адреса – байтовое смещение – игнорируются при обращении к словам. Следующие 3 бита указывают на ячейку, или набор, кэш-памяти.

Команда загрузки читает эту ячейку из кэш-памяти и проверяет тег и бит достоверности. Если тег совпадает со старшими 27 битами адреса и бит достоверности равен 1, то фиксируется попадание в кэш и данные передаются процессору. В противном случае имеет место промах кэша, и подсистема памяти должна прочитать запрошенные данные из оперативной памяти.

Пример 3.6. ВРЕМЕННАЯ ЛОКАЛЬНОСТЬ С КЭШ-ПАМЯТЬЮ ПРЯМОГО ОТОБРАЖЕНИЯ

Циклы – типичный источник временной и пространственной локальности данных в приложениях. Используя кэш с восемью ячейками, изображенный на **Рис. 3.7**, покажите, каким будет содержимое кэша после выполнения следующего небольшого цикла на языке ассемблера ARM. Считайте, что первоначально кэш пуст. Каким будет процент промахов?

```
      MOV R0, #5
      MOV R1, #0
LOOP  CMP R0, #0
      BEQ DONE
      LDR R2, [R1, #4]
      LDR R3, [R1, #12]
      LDR R4, [R1, #8]
      SUB R0, R0, #1
      B   LOOP
DONE
```

Решение. Эта программа содержит цикл, повторяющийся пять раз. Каждая итерация содержит три обращения в память (три команды загрузки), всего 15 обращений. Когда цикл выполняется в первый раз, кэш пуст, и данные, расположенные в оперативной памяти по адресам 0x4, 0xC и 0x8, должны быть загружены в наборы кэша 1, 3 и 2 соответственно. Однако на следующих четырех итерациях цикла данные будут получены уже из кэша. На **Рис. 3.8** показано содержимое кэша при последнем обращении по адресу 0x4. Все теги равны нулю, потому что старшие 27 бит всех адресов равны нулю. Процент промахов кэша составляет 3/15 = 20%.

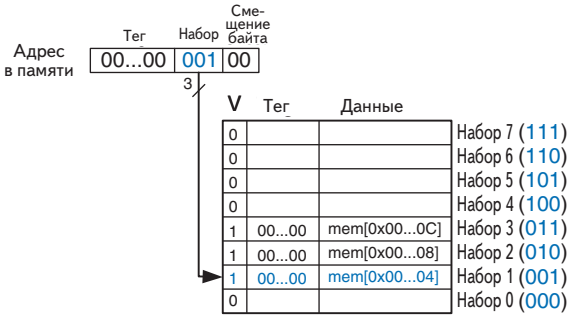


Рис. 3.8. Содержимое кэша прямого отображения

Когда два обращения к памяти по разным адресам отображаются в одну и ту же строку кэша, возникает *конфликт*, и данные, загруженные во время последнего обращения, *вытесняют* (evict) из кэша данные, загруженные раньше. В кэше прямого отображения в каждом наборе есть только одна строка, так что два адреса, отображаемых в одну строку, всегда вызывают конфликт. Один из таких конфликтов рассмотрен в **примере 3.7**.

Пример 3.7. КОНФЛИКТЫ ПРИ ОБРАЩЕНИИ В КЭШ-ПАМЯТЬ

Чему будет равен процент промахов при выполнении следующего цикла, если имеется кэш прямого отображения емкостью 8 слов, показанный на **Рис. 3.8**? Считайте, что первоначально кэш пуст.

```
        MOV R0, #5
        MOV R1, #0
LOOP    CMP R0, #0
        BEQ DONE
        LDR R2, [R1, #0x4]
        LDR R3, [R1, #0x24]
        SUB R0, R0, #1
        B   LOOP
DONE
```

Решение. Оба адреса памяти (0x4 и 0x24) отображаются в набор 1. На первой итерации цикла данные по адресу 0x4 будут загружены в набор 1. Затем в тот же набор загружаются данные по адресу 0x24, вытесняя данные по адресу 0x4. На второй итерации все повторяется: кэш должен повторно прочитать данные по адресу 0x4, вытеснив данные по адресу 0x24. Эти два адреса конфликтуют, так что процент промахов кэша равен 100%.

Многосекционный наборно-ассоциативный кэш

N-секционный наборно-ассоциативный кэш уменьшает число конфликтов за счет расширения набора до *N* строк. Каждый адрес памяти по-прежнему отображается в строго определенный набор, но теперь он может быть отображен в любую из *N* строк этого набора. Можно сказать, что кэш прямого отображения – это односекционный наборно-ассоциативный кэш. Число *N* называют *степенью ассоциативности* кэша.

На **Рис. 3.9** показана блок-схема аппаратной реализации наборно-ассоциативного кэша емкостью $C = 8$ слов с $N = 2$ секциями. Теперь в кэше только $S = 4$ набора вместо 8. Таким образом, только $\log_2 4 = 2$ бита, а не 3, как раньше, используются для выбора нужного набора. Соответственно, размер тега увеличивается с 27 до 28 бит. Каждый набор теперь содержит две секции. Каждая секция состоит из строки данных, тега и бита достоверности. Кэш читает теги и биты достоверности одновременно из

обеих секций выбранного набора, после чего сравнивает их с адресом для определения попадания или промаха. Если имеет место попадание в одну из секций кэша, то мультиплексор выбирает данные из этой секции.

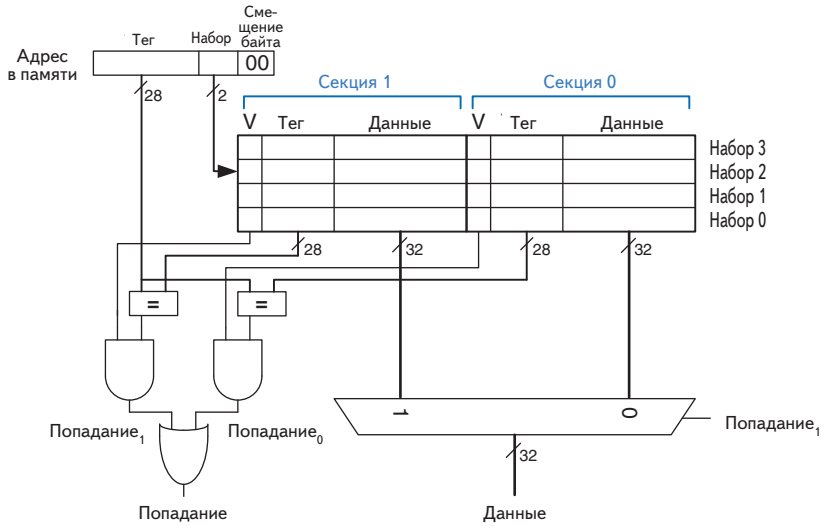


Рис. 3.9. Двухсекционный наборно-ассоциативный кэш

Наборно-ассоциативные кэши, как правило, имеют меньший процент промахов, чем кэши прямого отображения той же емкости, т. к. в них происходит меньше конфликтов. Однако они обычно медленнее и дороже в реализации, поскольку необходимо использовать дополнительные компараторы и выходной мультиплексор. Кроме того, в таких кэшах возникает вопрос о том, какую именно секцию замещать, когда все они заняты; мы рассмотрим эту проблему в **разделе 3.3.3**. В большинстве коммерческих систем используют наборно-ассоциативные кэши.

Пример 3.8. ПРОЦЕНТ ПРОМАХОВ НАБОРНО-АССОЦИАТИВНОГО КЭША

Повторите **пример 3.7**, используя двухсекционный кэш, показанный на **Рис. 3.9**, емкость которого равна восьми словам.

Решение. Оба обращения в память (по адресу 0x4 и по адресу 0x24) отображаются в набор 1. Однако теперь кэш имеет две секции, так что он может разместить данные для этих обращений в одном наборе. На первой итерации цикла пустой кэш приводит к двум промахам, после чего загружается два слова данных в две секции строки 1, как показано на **Рис. 3.10**. На следующих четырех итерациях данные будут прочитаны из кэша. В результате процент промахов равен $2/10 = 20\%$. Напомним, что для кэша прямого отображения того же размера из **примера 3.7** процент промахов был равен 100%.

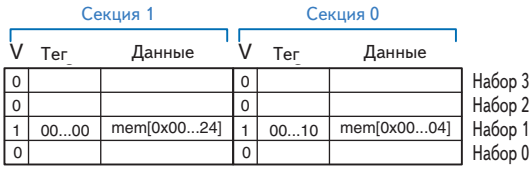


Рис. 3.10. Содержимое двухсекционного наборно-ассоциативного кэша

Полностью ассоциативный кэш

Полностью ассоциативный кэш состоит из одного набора с B секциями, где B – число строк. Адрес памяти может быть отображен в строку любой из этих секций. Можно сказать, что полностью ассоциативный кэш – это B-секционный наборно-ассоциативный кэш с одним набором.

На **Рис. 3.11** показан массив памяти SRAM полностью ассоциативного кэша, содержащего 8 строк. При запросе данных необходимо сделать восемь сравнений адреса с тегом, т. к. данные могут находиться в любой строке. Восьмивходовый мультиплексор (на рисунке не показан) выбирает соответствующую строку и подает ее на выход, если произошло попадание. Полностью ассоциативные кэши обеспечивают при прочих равных условиях минимально возможное количество конфликтов, но требуют еще больше аппаратуры для дополнительных сравнений тегов. Из-за этого они применяются лишь в относительно маленьких кэшах.

Рис. 3.11. Полностью ассоциативный кэш с восемью строками

Длина строки

В предыдущих примерах мы использовали преимущества исключительно временной локальности данных, т. к. длина строки кэша была равна одному слову. Чтобы воспользоваться пространственной локальностью, в кэш-памяти используют более длинные строки, в которых можно хранить несколько последовательных слов.

Преимущества строк с длиной, превышающей одно слово, заключается в том, что когда случается промах кэша и требуется прочитать слово данных из памяти, то в эту строку заодно загружаются и соседние слова. Таким образом, последующие обращения с большей вероятностью приведут к попаданию в кэш из-за пространственной локальности данных. Однако из-за увеличения длины строки кэш того же размера будет содержать меньше строк. Это может привести к росту числа конфликтов

и соответственно увеличить вероятность промахов кэша. Более того, потребуется больше времени на чтение данных в строку после промаха, т. к. из памяти необходимо будет прочитать не одно, а несколько слов. Время, требуемое для загрузки данных в строку кэша после промаха, называется *ценой промаха* (miss penalty). Если соседние слова данных в строке не будут востребованы в дальнейшем, то усилия на их загрузку будут потрачены зря. Тем не менее большинству реальных программ увеличение длины строки приносит пользу.

На **Рис. 3.12** показана блок-схема аппаратной реализации кэша прямого отображения емкостью 8 слов с длиной строки $b = 4$ слова. В кэше теперь только $B = C / b = 2$ строки. Так как в кэше прямого отображения число строк и наборов совпадает, то в данном случае кэш содержит два набора. Следовательно, только $\log_2 2 = 1$ бит адреса используется для определения номера набора. Теперь понадобится новый мультиплексор для выбора того из слов строки, которое будет передано процессору. Этот мультиплексор управляется $\log_2 4 = 2$ битами адреса, которые называются *битами смещения в строке*. Оставшиеся 27 старших бит адреса образуют тег. На всю строку нужен только один тег, т. к. слова в ней размещены по последовательным адресам.

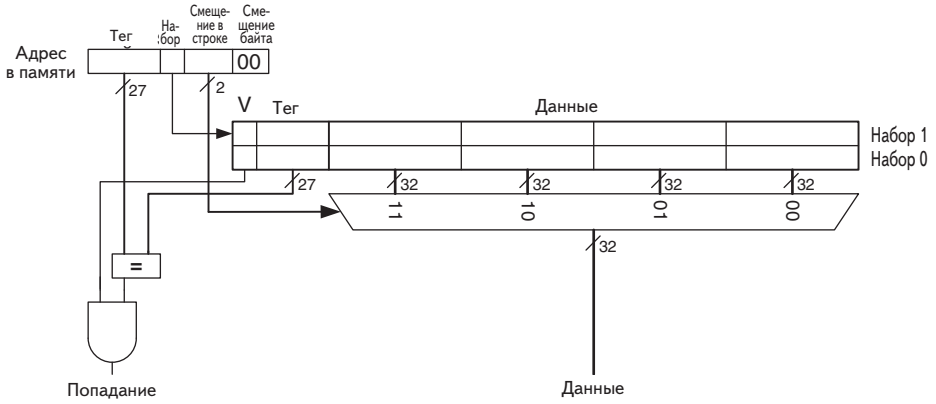


Рис. 3.12. Кэш прямого отображения со строкой длиной 4 слова

На **Рис. 3.13** показано, на какие поля разбивается адрес 0x8000009C при отображении в кэш прямого отображения, показанный на **Рис. 3.12**. Биты байтового смещения всегда равны нулю при доступе к словам. Следующие $\log_2 b = 2$ бита определяют смещение в строке. Следующий бит выбирает один из двух наборов, а оставшиеся 27 бит образуют тег. Следовательно, слово по адресу 0x8000009C отображается в третье слово первого набора кэша. Принцип использования более длинных строк для эксплуатации свойства пространственной локальности данных применяется и к ассоциативным кэшам.

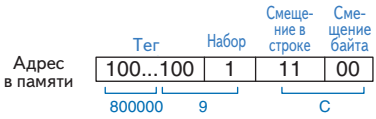


Рис. 3.13. Поля адреса 0x8000009C при отображении в кэш, показанный на Рис. 3.12

Пример 3.9. ПРОСТРАНСТВЕННАЯ ЛОКАЛЬНОСТЬ С КЭШ-ПАМЯТЬЮ ПРЯМОГО ОТОБРАЖЕНИЯ

Повторите **пример 3.6** для кэша прямого отображения емкостью 8 слов, длина строки которого равна четырем словам.

Решение. На **Рис. 3.14** показано содержимое кэша после первого обращения к памяти. На первой итерации цикла происходит промах кэша при обращении в память по адресу 0x4, после чего в строку кэша загружаются данные с адреса 0x0 по адрес 0xC. Все последующие обращения (как показано на рисунке для адреса 0xC) попадают в кэш. Следовательно, процент промахов будет равен $1/15 = 6.67\%$.

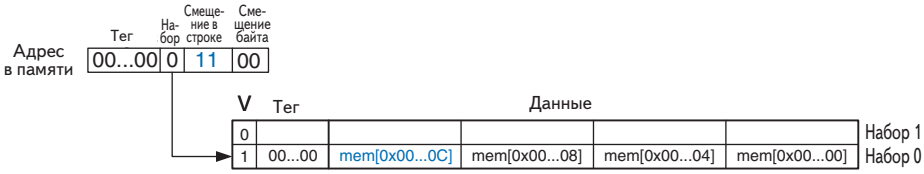


Рис. 3.14. Содержимое кэша со строкой длиной 4 слова

Подводя итоги

Кэш представляет собой двумерный массив. Строки этого массива называют наборами, а столбцы – секциями. Каждый элемент массива содержит строку и ассоциированные с ней тег и бит достоверности. Кэш характеризуется:

- емкостью C;
- длиной строки b (и числом строк $B = C/b$);
- числом строк в наборе (N).

В **Табл. 3.2** перечислены различные способы организации кэш-памяти. Любой адрес в памяти отображается только на один набор, но соответствующие этому адресу данные могут оказаться в любой секции этого набора.

Таблица 3.2. Способы организации кэш-памяти

Способ организации	Количество секций (N)	Количество наборов (S)
Прямого отображения	1	B
Наборно-ассоциативный	$1 < N < B$	B / N
Полностью ассоциативный	B	1

Емкость кэша, степень ассоциативности, число наборов и длина строки обычно являются степенями двойки. Это позволяет однозначно соотносить определенные биты адреса с битами тега, номера набора и смещения в строке.

Увеличение степени ассоциативности N обычно уменьшает процент промахов кэша из-за конфликтов. При этом чем больше степень ассоциативности, тем больше требуется компараторов для сравнения адреса с тегами. Увеличение длины строки b позволяет использовать пространственную локальность данных для уменьшения процента промахов, но тогда в кэше того же размера станет меньше наборов, что может привести к увеличению числа конфликтов. К тому же увеличивается цена промаха.

3.3.3. Какие данные заместить в кэш-памяти?

В кэш-памяти прямого отображения любому адресу всегда соответствует одна и та же строка одного и того же набора. Поэтому когда нужно загрузить новые данные в уже заполненный набор, строка в наборе просто замещается новыми данными. В наборно-ассоциативной и полностью ассоциативной кэш-памяти нужно решить, какую именно из нескольких строк набора вытеснить. Учитывая принцип временной локальности, наилучшим вариантом было бы заменить ту строку, которая дольше всего не использовалась, потому что маловероятно, что она будет использована снова. Именно поэтому в большинстве кэшей используется стратегия вытеснения по давности использования (least recently used, LRU).

В двухсекционном наборно-ассоциативном кэше *бит использования U* (от англ. *used*) помечает ту секцию в наборе, которая дольше не использовалась. При любом доступе к одной из секций набора бит U корректируется. Для наборно-ассоциативных кэшей с большим количеством секций отслеживать самые редко используемые строки становится сложно. Чтобы упростить реализацию, секции часто делят на две группы, а бит использования указывает на ту *группу*, которая дольше не использовалась. При необходимости заместить строку вытесняется случайным образом выбранная строка из той группы, которая дольше не использовалась. Такая стратегия называется *псевдо-LRU* и на практике достаточно хорошо работает.

Пример 3.10. СТРАТЕГИЯ ЗАМЕЩЕНИЯ LRU

Покажите содержимое двухсекционного наборно-ассоциативного кэша емкостью 8 слов после выполнения следующего кода. Используйте стратегию замещения LRU и считайте, что длина строки равна одному слову, а кэш первоначально пуст.

```
MOV R0, #0
LDR R1, [R0, #4]
LDR R2, [R0, #0x24]
LDR R3, [R0, #0x54]
```

Решение. Первые две команды загружают данные из памяти по адресам 0x4 и 0x24 в набор 1 кэша, как показано на **Рис. 3.15 (a)**. Бит использования $U = 0$ показывает, что дольше всего не использовались данные в секции 0. Следующее обращение в память по адресу 0x54 также отображается в набор 1 и вытесняет дольше всего не использовавшиеся данные из секции 0, как показано на **Рис. 3.15 (b)**. Бит использования при этом устанавливается в 1, указывая, что теперь секцией с дольше всего не использовавшимися данными стала секция 1.

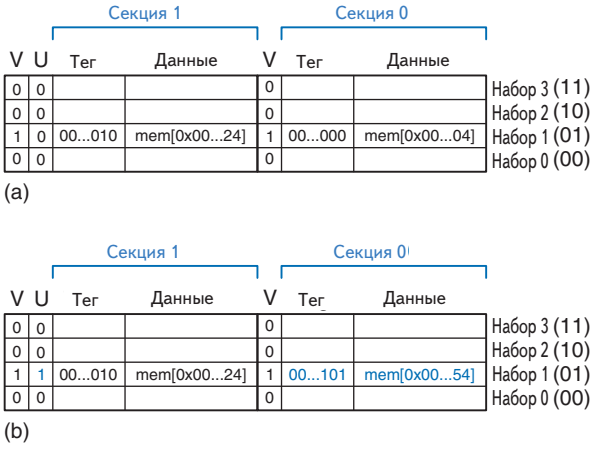


Рис. 3.15. Двухсекционный кэш со стратегией замещения LRU

3.3.4. Улучшенная кэш-память

В современных системах для сокращения времени доступа к памяти используется несколько уровней кэша. В этом разделе мы рассмотрим производительность двухуровневой системы кэширования и выясним, как длина строки, ассоциативность и емкость кэша влияют на частоту промахов. Также мы рассмотрим, как работает кэширование с использованием стратегий сквозной (write-though) и отложенной (write-back) записи в память.

Многоуровневые кэши

Чем больше размер кэша, тем больше вероятность, что интересующие нас данные в нем найдутся, и, следовательно, тем меньше у него будет частота промахов. Но большой кэш обычно медленнее, чем маленький, поэтому в современных системах используют как минимум два уровня кэша, как показано на **Рис. 3.16**. Кэш первого уровня (L1) достаточно мал, чтобы обеспечить время доступа в один или два такта. Кэш второго уровня (L2) тоже выполнен по технологии SRAM, но больше по размеру и потому медленнее, чем кэш L1. Сначала процессор ищет данные в кэше L1, а если происходит промах, то в кэше L2. Если и там происходит промах, то процессор обращается за данными к оперативной памяти. Во многих современных системах используется еще больше уровней кэша в иерархии памяти, т. к. доступ к оперативной памяти чрезвычайно медленный.

Рис. 3.16. Иерархия памяти с двумя уровнями кэша

Пример 3.11. СИСТЕМА С КЭШЕМ L2

Предположим, что в системе, показанной на **Рис. 3.16**, времена доступа к кэшу L1, кэшу L2 и оперативной памяти равны 1, 10 и 100 тактам соответственно. Чему равно среднее время доступа к памяти при условии, что доля промахов для кэша L1 равна 5%, причем в 20% из них имеет место также промах для кэша L2?

Решение. При каждом обращении к памяти процессор сначала ищет запрошенные данные в кэше L1. Когда происходит промах (5% случаев), процессор ищет их в кэше L2. Если снова возникает промах (20% случаев), то процессор обращается за данными в оперативную память. Применяя формулу **(3.2)**, мы можем вычислить среднее время доступа к памяти как 1 такт + 0.05 (10 тактов + 0.2 (100 тактов)) = 2.5 такта.

Процент промахов кэша L2 выше, поскольку до него доходят лишь «трудные» обращения к памяти – те, которые уже привели к промаху кэша L1. Если бы все обращения шли непосредственно в кэш L2, доля его промахов была бы около 1%.

Сокращение частоты промахов

Процент промахов можно сократить, изменяя емкость кэша, длину строки и (или) ассоциативность. Для этого сначала необходимо разобраться с причинами промахов. Промахи кэша делятся на *неизбежные промахи* (compulsory misses), *промахи из-за недостаточной емкости* (capacity misses) и *промахи из-за конфликтов* (conflict misses). Первое обращение к строке кэша всегда приводит к неизбежному промаху, т. к. эту строку нужно прочитать из оперативной памяти хотя бы один раз независимо от архитектуры кэша. Промахи из-за недостаточной емкости происходят, когда кэш слишком мал для хранения всех одновременно используемых данных. Промахи из-за конфликтов случаются, если несколько адресов памяти отображаются на один и тот же набор кэша и выталкивают из него данные, которые все еще нужны.

Изменение параметров кэша может повлиять на частоту одного или нескольких типов промахов. Например, увеличение размера кэша может сократить промахи из-за конфликтов и промахи из-за недостатка емкости, но никак не повлияет на число неизбежных промахов. С другой стороны, увеличение длины строки может сократить число неизбежных промахов (благодаря пространственной локальности), но одновременно увеличить частоту промахов из-за конфликтов, поскольку большее число адресов будет отображаться на один и тот же набор, повышая вероятность конфликтов.

Системы памяти настолько сложны, что лучший способ оценивать их производительность – запускать тестовые программы, варьируя параметры кэша. На **Рис. 3.17** изображен график зависимости частоты промахов от размера кэша и степени ассоциативности для теста SPEC2000. Небольшое число неизбежных промахов показано темным цветом вдоль оси X и не зависит от емкости кэша. С другой стороны, как и ожидалось, с увеличением емкости кэша частота промахов из-за недостатка емкости сокращается. Увеличение ассоциативности, особенно для кэшей небольшого размера, сокращает количество промахов из-за конфликтов, показанных вдоль верхней части кривой. При этом ассоциативность свыше четырех или восьми секций приводит лишь к незначительному сокращению частоты промахов.

Как уже отмечалось, частоту промахов можно уменьшить, взяв более длинные строки и тем самым воспользовавшись пространственной локальностью. Однако если размер кэша фиксирован, то с увеличением длины строки уменьшается количество наборов, а значит, повышается вероятность конфликтов. На **Рис. 3.18** показана зависимость частоты промахов от длины строки в байтах для кэшей разной емкости. Для небольших кэшей, например емкостью 4 КБ, при длине строки свыше 64 байтов частота промахов из-за конфликтов *возрастает*. Для кэшей большей емкости увеличение длины свыше 64 байтов не влияет на ча-

стоту промахов. Однако большая длина строки все же может вызвать увеличение времени выполнения из-за более высокой *цены промаха* – времени, требуемого для выборки отсутствующей строки кэша из оперативной памяти.

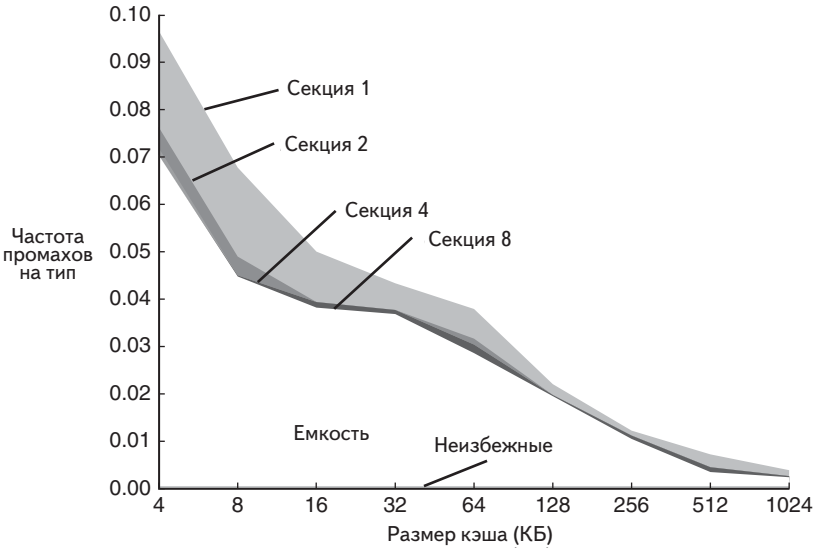


Рис. 3.17. Зависимость частоты промахов от размера и ассоциативности кэша на тесте SPEC2000
График заимствован из книги: *Hennessy and Patterson.* Computer Architecture: A Quantitative Approach. 5-е изд. 5. Morgan Kaufmann, 2012 (печатается с разрешения авторов)

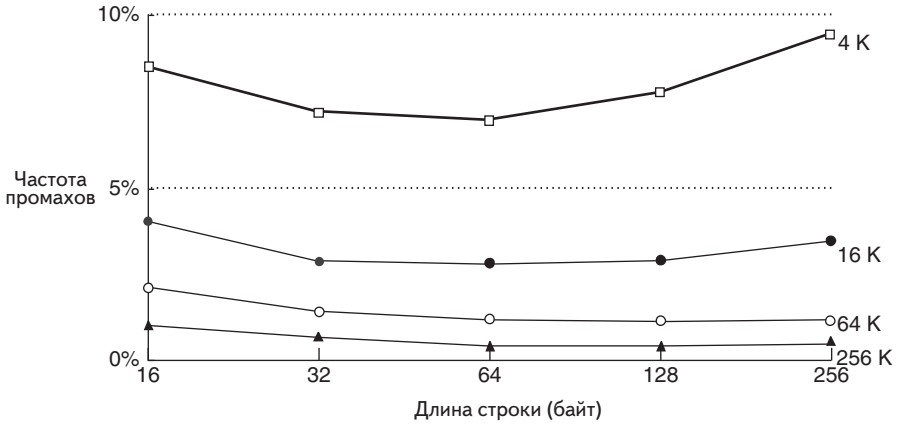


Рис. 3.18. Зависимость частоты промахов от длины строки и размера кэша на тесте SPEC92
График заимствован из книги: *Hennessy and Patterson.* Computer Architecture: A Quantitative Approach. 5-е изд. 5. Morgan Kaufmann, 2012 (печатается с разрешения авторов)

Стратегии записи

В предыдущих разделах мы рассматривали чтение из памяти. Запись в память выполняется примерно так же, как чтение. При выполнении команды сохранения данных процессор сначала проверяет кэш. В случае промаха соответствующая строка выбирается из оперативной памяти в кэш, а затем в нее записывается нужное слово. В случае попадания слово просто записывается в строку кэша.

Кэши делятся на два типа: со *сквозной записью* (write-through) и с *отложенной записью* (write-back). В кэше со сквозной записью данные, записываемые в кэш, одновременно записываются и в оперативную память. В кэше с отложенной записью с каждой строкой ассоциирован бит изменения *D* (от англ. *dirty*). Если в строку производилась запись, то этот бит равен 1, в противном случае он равен 0. Измененные строки записываются обратно в оперативную память только тогда, когда они вытесняются из кэша. В кэше со сквозной записью биты изменения не нужны, но такой кэш обычно приводит к большему количеству операций записи в память, чем кэш с отложенной записью. Из-за того, что время обращения к оперативной памяти очень велико, в современных системах обычно используют кэши с отложенной записью.

Пример 3.12. СКВОЗНАЯ И ОТЛОЖЕННАЯ ЗАПИСЬ

Допустим, что длина строки кэша – четыре слова. Сколько обращений к оперативной памяти потребуется при выполнении приведенного ниже кода, если используется стратегия сквозной записи, и сколько, если используется отложенная запись?

```
MOV R5, #0
STR R1, [R5]
STR R2, [R5, #12]
STR R3, [R5, #8]
STR R4, [R5, #4]
```

Решение. Все четыре команды сохранения пишут в одну и ту же строку кэша. При сквозной записи каждая команда сохраняет слово в оперативной памяти, следовательно, потребуется четыре обращения к памяти. При отложенной записи потребуется только одно обращение – тогда, когда эта строка будет вытеснена из кэша.

3.3.5. Эволюция кэш-памяти процессоров ARM

В **Табл. 3.3** прослежена эволюция организации кэш-памяти в процессорах ARM с 1985 по 2012 год. Основные тенденции – введение нескольких уровней кэша, увеличение емкости и разделение кэшей L1 для команд и данных. Движущими силами этих изменений явились увеличивающийся

разрыв между частотой процессора и скоростью оперативной памяти, а также снижение стоимости транзисторов. Разрыв между частотой процессора и скоростью памяти вынуждает сокращать частоту промахов во избежание появления узкого места при обращениях к памяти, а снижение стоимости транзисторов позволяет увеличивать размеры кэшей.

Таблица 3.3. Эволюция кэш-памяти процессоров ARM

Год	CPU	МГц	Кэш L1	Кэш L2
1985	ARM1	8	Нет	Нет
1992	ARM6	30	4 КБ, объединенный	Нет
1994	ARM7	100	8 КБ, объединенный	Нет
1999	ARM9E	300	0—128 КБ, К/Д	Нет
2002	ARM11	700	4—64 КБ, К/Д	0—128 КБ, вне кристалла
2009	Cortex-A9	100	16—64 КБ, К/Д	0—8 МБ
2011	Cortex-A7	1500	32 КБ, К/Д	0—4 МБ
2011	Cortex-A15	2000	32 КБ, К/Д	0—4 МБ
2012	Cortex-M0+	60—250	Нет	Нет
2012	Cortex-A53	1500	8—64 КБ, К/Д	128 КБ—2 МБ
2012	Cortex-A57	2000	48 КБ К/32 КБ D	512 КБ—2 МБ

3.4. Виртуальная память

В большинстве современных вычислительных систем в качестве нижнего уровня в иерархии памяти используются жесткие диски, представляющие собой магнитные или твердотельные запоминающие устройства (см. **Рис. 3.4**). По сравнению с идеальной памятью, которая должна быть быстрой, дешевой и большой, жесткий диск имеет большой объем и недорого стоит, однако работает невероятно медленно. Жесткий диск располагает куда большей емкостью, чем недорогая оперативная память (DRAM). Однако если доступ к памяти в значительной мере состоит из обращений к диску, то скорость работы сильно снижается. Вы могли сталкиваться с этой проблемой на персональном компьютере, если одновременно запускали слишком много программ.

На **Рис. 3.19** изображен магнитный жесткий диск со снятой крышкой. Как следует из названия, жесткий диск состоит из одной или нескольких твердых пластин, каждой из которых соответствует головка считывания-записи, расположенная на конце длинного треугольного

рычага. Головка перемещается в правильное положение относительно диска и считывает или записывает информацию при помощи магнитного поля в момент, когда диск проходит под ней. Чтобы занять правильное положение, головке требуется несколько миллисекунд – быстро с точки зрения человека, но в миллионы раз медленнее, чем скорость работы процессора. Жесткие диски постепенно вытесняются твердотельными, поскольку те читают на несколько порядков быстрее (см. **Рис. 3.4**) и не подвержены механическим отказам.

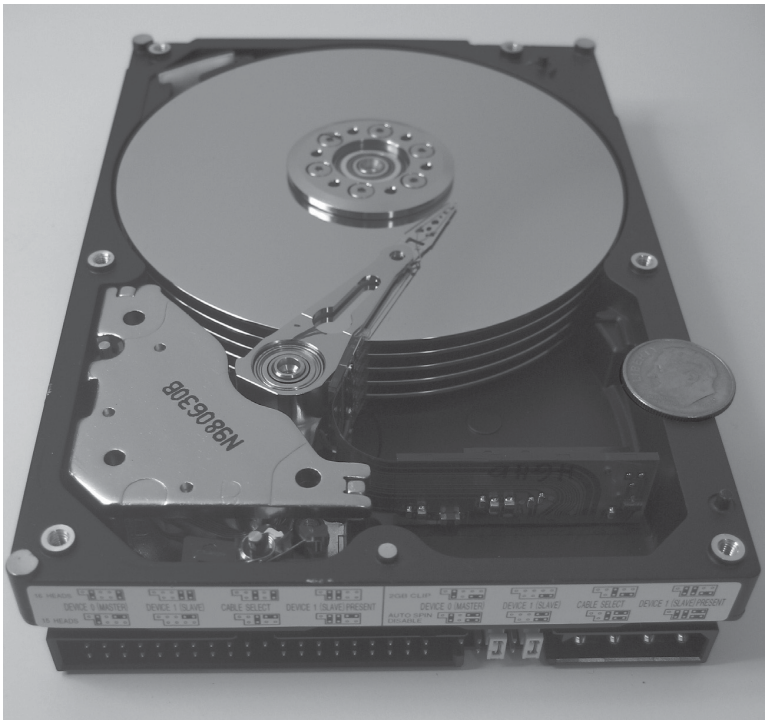


Рис. 3.19. Жесткий диск

Компьютер с 32-битовой адресацией имеет доступ к 232 байтам = 4 ГБ памяти. Эта одна из причин перехода к 64-битовым компьютерам, которым доступна память гораздо большего объема.

Цель включения жесткого диска в иерархию памяти – за небольшую плату создать иллюзию памяти большого объема, обеспечив в то же время скорость доступа, характерную для более быстрых типов памяти, на большинстве обращений к памяти. Например, компьютер с оперативной памятью 128 МБ может обеспечить видимость наличия 2 ГБ оперативной памяти, используя для этого жесткий диск. В этом случае большая память, объемом 2 ГБ, называется *виртуальной памятью*, а меньшая память, объемом 128 МБ, – *физической памятью*. В этом разделе мы будем использовать термин физическая память, подразумевая оперативную память.

Программы могут обращаться к данным в любом месте виртуальной памяти, поэтому должны использовать *виртуальные адреса*, определяющие расположение в виртуальной памяти. Физическая память хранит последние запрошенные из виртуальной памяти блоки данных. Таким образом, физическая память выступает в роли кэша для виртуальной памяти, то есть большинство обращений происходит к быстрой физической памяти (DRAM), и в то же время программа имеет доступ к большей по объему виртуальной памяти.

В подсистемах виртуальной памяти используется другая терминология для тех же самых принципов кэширования, которые были рассмотрены в **разделе 3.3**. В **Табл. 3.4** устанавливается соответствие между терминами. Виртуальная память разделена на *виртуальные страницы*, обычно размером 4 КБ. Аналогично физическая память разделена на *физические страницы*. Размеры виртуальных и физических страниц одинаковы. Виртуальная страница может располагаться в физической памяти (DRAM) или на жестком диске. Например, на **Рис. 3.20** показана виртуальная память, которая больше физической памяти. Прямоугольники обозначают страницы. Одни виртуальные страницы расположены в физической памяти, другие – на жестком диске. Процесс преобразования виртуального адреса в физический называется *трансляцией адреса*. Если процессор обращается к виртуальному адресу, которого нет в физической памяти, происходит *страничный отказ* (page fault), и операционная система подгружает соответствующую страницу с жесткого диска в физическую память.

Таблица 3.4. Соответствие терминологии кэша и виртуальной памяти

Кэш	Виртуальная память
Строка	Страница
Длина строки	Размер страницы
Смещение относительно начала строки	Смещение относительно начала страницы
Промах	Страничный отказ
Тег	Номер виртуальной страницы

Чтобы избежать страничных отказов, вызванных конфликтами, любая виртуальная страница может отображаться на любую физическую. Другими словами, физическая память работает как полностью ассоциативный кэш для виртуальной памяти. В традиционном полностью ассоциативном кэше каждая секция содержит компаратор, который проверяет старшие биты адреса на соответствие тегу, чтобы определить, находятся ли там нужные данные. В аналогичной системе виртуальной памяти каждой физической странице нужен был бы компаратор, чтобы сверять старшие биты виртуальных адресов с тегом и определять, отображается ли виртуальная страница на эту физическую страницу.

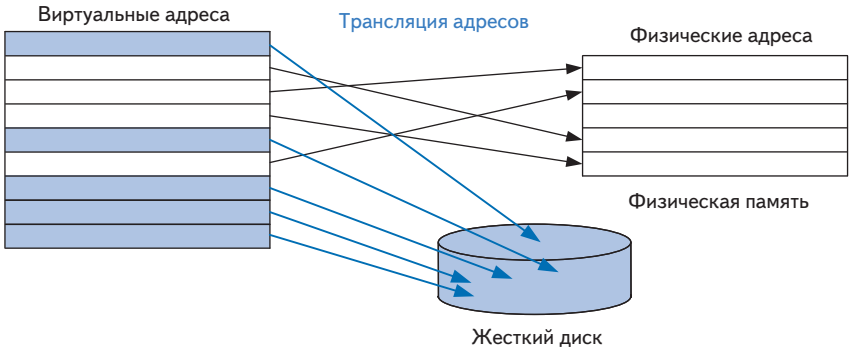


Рис. 3.20. Виртуальные и физические страницы

На практике в подсистеме виртуальной памяти так много физических страниц, что обеспечивать компаратором каждую страницу было бы чересчур дорого. Вместо этого в подсистемах виртуальной памяти используется трансляция адресов при помощи *таблицы страниц*. Таблица страниц содержит запись для каждой виртуальной страницы, в которой хранится либо ее местоположение в физической памяти, либо признак нахождения на жестком диске. Каждая команда загрузки или сохранения требует доступа к таблице страниц с последующим доступом к физической памяти. Обращение к таблице страниц позволяет транслировать виртуальный адрес, используемый программой, в физический адрес. Затем физический адрес используется для фактического чтения или записи данных.

Таблица страниц обычно настолько велика, что сама находится в физической памяти. Таким образом, каждая команда загрузки или сохранения данных включает два обращения к физической памяти: обращение к таблице страниц и собственно обращение к данным. Чтобы ускорить трансляцию адреса, используется *буфер ассоциативной трансляции* (translation lookaside buffer, TLB), который содержит наиболее часто используемые записи таблицы страниц.

Далее в этом разделе мы более подробно рассмотрим трансляцию адресов, таблицы страниц и TLB.

3.4.1. Трансляция адресов

В системах с виртуальной памятью программы используют виртуальные адреса и потому имеют доступ к памяти большого объема. Компьютер должен транслировать эти виртуальные адреса, чтобы либо найти соответствующий адрес в физической памяти, либо получить страничный отказ и загрузить данные с жесткого диска.

Напомним, что виртуальная память и физическая память разделены на страницы. Старшие биты виртуального и физического адресов опреде-

ляют *номер* виртуальной и физической страницы соответственно. Младшие биты определяют положение слова внутри страницы и называются *смещением относительно начала страницы*.

На **Рис. 3.21** показана страничная организация системы с виртуальной памятью, в которой объем виртуальной памяти равен 2 ГБ, физической памяти – 128 МБ, и та и другая разделены на страницы по 4 КБ. В ARM используется 32-битовая адресация. Так как виртуальная память имеет объем 2 ГБ = 2^{31} байт, то используются только младшие 31 бит виртуального адреса, а старший бит всегда равен нулю. Аналогично, поскольку объем физической памяти 128 МБ = 2^{27} байт, то используются только младшие 27 бит физического адреса, а старшие 5 бит всегда равны нулю.

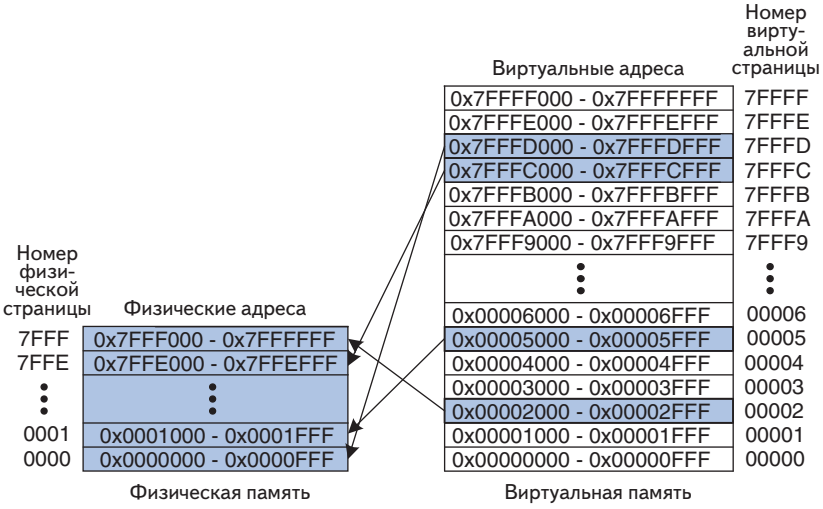


Рис. 3.21. Физические и виртуальные страницы

Поскольку размер страницы составляет 4 КБ = 2^{12} байт, существует $2^{31} / 2^{12} = 2^{19}$ виртуальных страниц и $2^{27} / 2^{12} = 2^{15}$ физических страниц. Таким образом, номера страниц виртуальной и физической памяти состоят из 19 и 15 бит соответственно. В любой момент времени в физической памяти может находиться максимум 1/16 часть страниц виртуальной памяти. Остальные виртуальные страницы хранятся на жестком диске.

На **Рис. 3.21** показано, как виртуальная страница 5 отображается на физическую страницу 1, виртуальная страница 0x7FFFC – на физическую страницу 0x7FFE и т. д. Например, виртуальный адрес 0x53F8 (смещение 0x3F8 от начала виртуальной страницы номер 5) отображается на физический адрес 0x13F8 (смещение 0x3F8 от начала физической страницы номер 1). Младшие 12 бит виртуального и физического адресов одинаковы (0x3F8) и определяют смещение от начала виртуальной и

физической страниц. Таким образом, чтобы получить физический адрес из виртуального, необходимо транслировать только номер страницы.

На **Рис. 3.22** показан процесс трансляции виртуального адреса в физический. Младшие 12 бит определяют смещение от начала страницы и не нуждаются в трансляции. Старшие 19 бит виртуального адреса определяют номер виртуальной страницы (virtual page number, VPN) и транслируются в 15-битный номер физической страницы (physical page number, PPN). В следующих двух разделах рассказывается, как для трансляции адресов используются таблицы страниц и TLB.

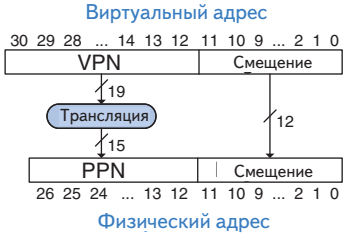


Рис. 3.22. Трансляция виртуального адреса в физический

Пример 3.13. ТРАНСЛЯЦИЯ ВИРТУАЛЬНОГО АДРЕСА В ФИЗИЧЕСКИЙ

Найдите физический адрес, соответствующий виртуальному адресу 0x247C, используя подсистему виртуальной памяти, показанную на **Рис. 3.21**.

Решение. 12 бит, обозначающие смещение от начала страницы (0x47C), не нуждаются в трансляции. Оставшиеся 19 бит виртуального адреса определяют номер виртуальной страницы. Это означает, что виртуальный адрес 0x247C находится внутри виртуальной страницы 0x2. Согласно **Рис. 3.21**, виртуальная страница 0x2 отображается на физическую страницу 0x7FFF. Таким образом, виртуальный адрес 0x247C отображается на физический адрес 0x7FFF47C.

3.4.2. Таблица страниц

Для трансляции виртуальных адресов в физические процессор использует *таблицу страниц*. В таблице страниц имеется по одной записи для каждой виртуальной страницы. Эта запись содержит номер физической страницы и бит достоверности (valid bit). Если бит достоверности равен 1, то виртуальная страница отображается на физическую страницу, номер которой указан в записи. В противном случае виртуальная страница находится на жестком диске.

Поскольку таблица страниц велика, она хранится в физической памяти. Предположим, что она хранится в виде непрерывного массива, как показано на **Рис. 3.23**.

Пример 3.14. ИСПОЛЬЗОВАНИЕ ТАБЛИЦЫ СТРАНИЦ ДЛЯ ТРАНСЛЯЦИИ АДРЕСА

Найти физический адрес, соответствующий виртуальному адресу 0x247C, используя таблицу страниц, показанную на **Рис. 3.23**.

Решение. На **Рис. 3.24** показана трансляция виртуального адреса 0x247C в физический. 12 бит, обозначающие смещение от начала страницы, не нуждаются в трансляции. Оставшиеся 19 бит виртуального адреса – это номер виртуальной страницы, 0x2, они дают индекс в таблице страниц. Таблица страниц содержит информацию о том, что виртуальная страница 0x2 отображается на физическую страницу 0x7FFF. Таким образом, виртуальный адрес 0x247C отображается на физический адрес 0x7FFF47C. Младшие 12 бит одинаковы для физического и виртуального адресов.

V	Номер физической страницы	Номер виртуальной страницы
0		7FFFF
0		7FFFE
1	0x0000	7FFFD
1	0x7FFE	7FFFC
0		7FFFB
0		7FFFA
	⋮	⋮
0		00007
0		00006
1	0x0001	00005
0		00004
0		00003
1	0x7FFF	00002
0		00001
0		00000

Таблица страниц

Рис. 3.23. Таблица страниц для случая, показанного на Рис. 3.21

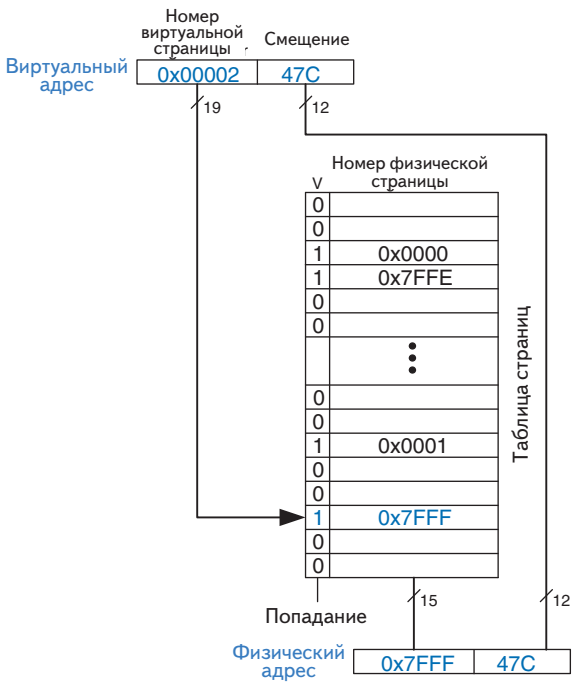


Рис. 3.24. Трансляция адреса с использованием таблицы страниц

Таблица страниц может храниться в любом месте физической памяти по усмотрению операционной системы. Процессор обычно использует специальный регистр, называемый *регистром таблицы страниц*, для хранения ее базового адреса.

Чтобы выполнить операцию загрузки или сохранения данных, процессор должен сначала транслировать виртуальный адрес в физический, а затем обратиться к физической памяти по этому физическому адресу. Процессор извлекает номер виртуальной страницы из виртуального адреса и прибавляет его к содержимому регистра таблицы страниц, чтобы найти физический адрес записи в таблице страниц. Затем процессор считывает эту запись из физической памяти и получает номер физической страницы. Если запись действительна, т. е. бит достоверности равен 1, то процессор объединяет номер физической страницы и смещение и получает физический адрес. Наконец, он читает или записывает данные по этому физическому адресу. Поскольку таблица страниц хранится в физической памяти, каждая команда загрузки или сохранения требует двух обращений к физической памяти.

3.4.3. Буфер ассоциативной трансляции

Виртуальная память оказывала бы огромное негативное влияние на производительность, если бы нужно было обращаться к таблице страниц при выполнении каждой команды загрузки или сохранения – тогда время выполнения этих команд удваивалось бы. К счастью, обращения к таблице страниц имеют высокую временную локальность. Временная и пространственная локальность обращений к данным и большой размер страницы означают, что с большой вероятностью многие последовательные команды загрузки или сохранения обращаются к одной и той же странице. Поэтому, если процессор запомнит последнюю прочитанную запись таблицы страниц, то, скорее всего, сможет повторно использовать результат трансляции, не выполняя повторного чтения таблицы страниц. В общем случае процессор может хранить несколько последних записей, прочитанных из таблицы страниц, в небольшом кэше, называемом *буфером ассоциативной трансляции* (TLB). Процессор «заглядывает» в TLB в поисках информации о трансляции, прежде чем обратиться к таблице страниц в физической памяти. В реальных программах большинство обращений находит в TLB нужную информацию (в этом случае говорят, что произошло попадание в TLB), что избавляет от затрат на повторное чтение таблицы страниц из физической памяти.

TLB организован как полностью ассоциативный кэш и обычно содержит от 16 до 512 записей. Каждая запись в TLB хранит номер виртуальной страницы и соответствующий ей номер физической страницы. Обращение к TLB происходит по номеру виртуальной страницы. Если происходит попадание в TLB, то возвращается номер соответствующей физической страницы. В противном случае процессор должен прочитать нужную запись из таблицы страниц в физической памяти. Буфер ассоциативной трансляции проектируют настолько маленьким, чтобы

доступ к нему занимал менее одного такта. Даже при этом частота попаданий в него обычно превышает 99%. TLB уменьшает число обращений к памяти, требуемое для большинства команд загрузки и сохранения, с двух до одного.

Пример 3.15. ИСПОЛЬЗОВАНИЕ TLB ДЛЯ ТРАНСЛЯЦИИ АДРЕСА

Рассмотрим подсистему виртуальной памяти, показанную на **Рис. 3.21**. Используйте TLB с двумя записями или объясните, почему необходимо обращение к таблице страниц, чтобы осуществить трансляцию виртуальных адресов 0x247C и 0x5FB0 в физические адреса. Предполагается, что TLB хранит корректные записи для виртуальных страниц 0x2 и 0x7FFFD.

Решение. На **Рис. 3.25** показаны TLB с двумя записями и запрос на трансляцию виртуального адреса 0x247C. Из поступившего на вход TLB адреса извлекается номер виртуальной страницы, равный 0x2, после чего этот номер сравнивается с номерами виртуальных страниц, находящимися в каждой записи. Совпадение номеров обнаружено для записи 0, при этом запись действительна (V = 1), т. е. имеет место попадание в TLB. Транслированный физический адрес представляет собой номер физической страницы из найденной записи, равный 0x7FFF, объединенный со смещением, скопированным из виртуального адреса. Как всегда, смещение не требует трансляции.

Запрос на трансляцию виртуального адреса 0x5FB0 приводит к промаху TLB, поэтому для трансляции необходимо использовать таблицу страниц.

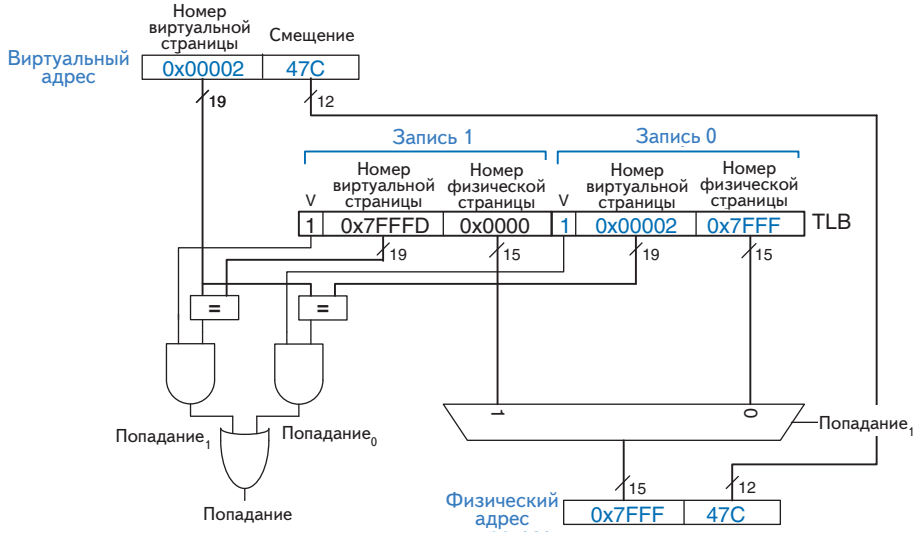


Рис. 3.25. Трансляция адреса с использованием TLB

3.4.4. Защита памяти

До сих пор в этом разделе мы рассматривали применение виртуальной памяти для создания видимости наличия быстрой, недорогой и большой по объему памяти. Не менее важной причиной использования виртуальной памяти является обеспечение защиты нескольких одновременно выполняемых программ друг от друга.

Как вы, возможно, знаете, современные компьютеры обычно выполняют несколько программ или процессов одновременно. Все эти программы одновременно присутствуют в физической памяти. В хорошо спроектированной компьютерной системе программы должны быть защищены друг от друга, чтобы работа одной программы не могла нарушить работу другой. Точнее, ни одна программа не должна иметь доступа к памяти другой программы без разрешения. Это называется *защитой памяти*.

Системы виртуальной памяти обеспечивают защиту памяти, предоставляя каждой программе персональное виртуальное адресное пространство. Каждая программа может использовать столько памяти в пределах виртуального пространства, сколько необходимо, но только часть виртуального адресного пространства находится в физической памяти в каждый момент времени. Каждая программа может полностью использовать свое виртуальное адресное пространство, не беспокоясь о том, где расположены остальные программы. Однако программа имеет доступ только к тем физическим страницам, информация о которых находится в ее таблице страниц. Таким образом, программа не может случайно или преднамеренно получить доступ к физическим страницам другой программы, поскольку они не отображены в ее таблице страниц. Иногда несколько программ должны иметь доступ к общим командам или данным. Для этого операционная система добавляет к каждой записи в таблице страниц несколько служебных битов, позволяющих определить, какие именно программы могут писать в общие физические страницы.

3.4.5. Стратегии замещения страниц

В подсистеме виртуальной памяти используются стратегия отложенной записи (write-back) и стратегия вытеснения редко используемых страниц (least recently used, LRU) для замещения страниц в физической памяти. Стратегия сквозной записи (write-though), при которой каждая запись в физическую память приводит заодно и к записи на жесткий диск, была бы непрактична, потому что команды сохранения выполнялись бы со скоростью жесткого диска, а не со скоростью процессора (миллисекунды вместо наносекунд). Стратегия отложенной записи подразумевает, что физическая страница записывается обратно на жесткий диск только тогда, когда она вытесняется из физической памяти. Процесс записи физической страницы обратно на жесткий диск и размещения на ее месте дру-

гой виртуальной страницы называется *замещением страниц* (paging), а жесткий диск в подсистемах виртуальной памяти иногда называется *пространством подкачки* (swap). Когда происходит страничный отказ, процессор замещает одну из редко используемых физических страниц отсутствующей виртуальной страницей, вызвавшей страничный отказ. Для такой реализации требуется, чтобы в каждой записи таблицы страниц присутствовали два дополнительных служебных бита: бит изменения D (dirty bit) и бит доступа U (use bit).

Бит изменения равен 1, если какая-либо команда сохранения изменила содержимое физической страницы после ее считывания с жесткого диска. Когда физическая страница с битом изменения, равным 1, замещается новой, ее необходимо записать обратно на жесткий диск. Если же бит изменения замещаемой страницы равен 0, то точная копия этой страницы и так уже находится на жестком диске, поэтому записывать ее туда не имеет смысла.

Бит доступа равен 1, если к физической странице недавно обращались. Как и в случае кэш-памяти, точный алгоритм LRU было бы слишком сложно реализовать. Вместо этого операционная система использует приближенный алгоритм вытеснения редко используемых страниц, периодически сбрасывая все биты доступа в таблице страниц. В момент обращения к странице бит доступа устанавливается в 1. В случае страничного отказа операционная система находит страницу с $U = 0$ и вытесняет ее из физической памяти. Таким образом, замещается необязательно самая редко используемая страница, но одна из нескольких относительно редко используемых страниц.

3.4.6. Многоуровневые таблицы страниц

Таблицы страниц могут занимать большой объем физической памяти. Например, для рассмотренной выше таблицы страниц при размере виртуальной памяти 2 ГБ и размере страниц 4 КБ потребуется 219 записей. Если размер каждой записи равен 4 байтам, то размер таблицы страниц будет равен $2^{19} \times 2^2$ байт = 2^{21} байт = 2 МБ.

Чтобы сэкономить физическую память, таблицы страниц можно организовать в виде многоуровневой иерархии (обычно двухуровневой). Таблица страниц первого уровня всегда хранится в физической памяти. Она указывает, в каком месте виртуальной памяти хранятся маленькие таблицы страниц второго уровня. Каждая таблица страниц второго уровня хранит информацию о некотором диапазоне виртуальных страниц. Если какой-то диапазон виртуальных адресов не используется, то соответствующая таблица страниц второго уровня может быть выгружена на жесткий диск, чтобы не занимать место в физической памяти.

В двухуровневой таблице страниц номер виртуальной страницы разделен на две части: *номер таблицы страниц* (page table number) и *сме-*

щение в таблице страниц (page table offset), как показано на **Рис. 3.26**. Номер таблицы страниц – это индекс строки в таблице первого уровня, которая должна всегда находиться в физической памяти. Запись в таблице страниц первого уровня содержит базовый адрес таблицы страниц второго уровня или, если $V = 0$, указывает, что ее необходимо загрузить с жесткого диска. Смещение в таблице страниц – это индекс строки в таблице второго уровня. Оставшиеся 12 бит виртуального адреса, как и раньше, определяют смещение от начала страницы размером $2^{12} = 4$ КБ.

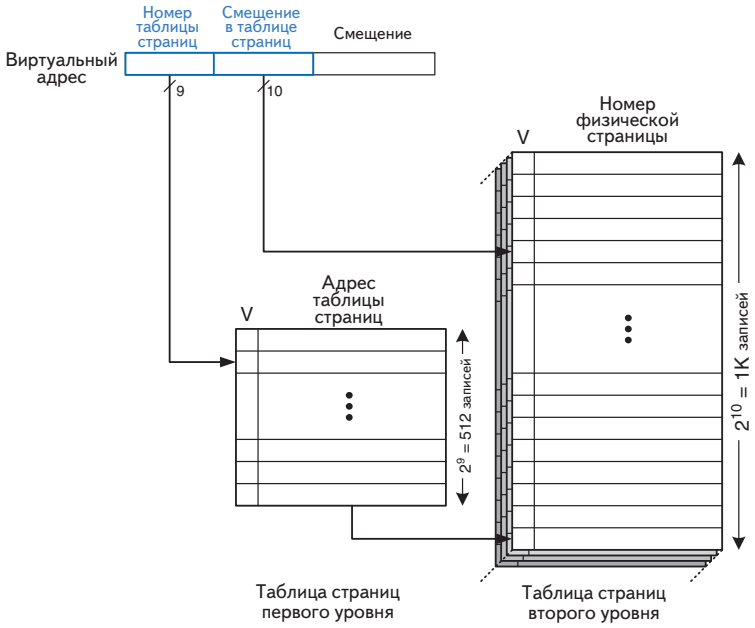


Рис. 3.26. Иерархические таблицы страниц

На **Рис. 3.26** 19-битовый номер виртуальной страницы разделен на две части по 9 и 10 бит, определяющие номер таблицы страниц и смещение в таблице страниц соответственно. Таким образом, таблица страниц первого уровня содержит $2^9 = 512$ записей. Каждая из 512 таблиц страниц второго уровня содержит $2^{10} = 1024$ записи. Если размер каждой записи в таблицах страниц первого и второго уровня равен 32 бит (4 байта) и только две таблицы страниц второго уровня присутствуют в физической памяти одновременно, то иерархическая таблица страниц занимает всего (512 × 4 байт) + 2 × (1024 × 4 байт) = 10 КБ физической памяти. Очевидно, что двухуровневая таблица страниц занимает лишь малую часть физической памяти, необходимой для хранения всей одноуровневой таблицы страниц (2 МБ). Недостаток двухуровневой таблицы состоит в том, что при промахе TLB необходимо будет выполнить на одно обращение к памяти больше.

Пример 3.16. ИСПОЛЬЗОВАНИЕ МНОГОУРОВНЕВОЙ ТАБЛИЦЫ СТРАНИЦ

На **Рис. 3.27** показано возможное содержимое двухуровневой таблицы страниц, изображенной на **Рис. 3.26**. Показано содержимое только одной таблицы страниц второго уровня. Используя эту двухуровневую таблицу, опишите, что происходит при обращении к виртуальному адресу 0x003FEFB0.

Решение. Как всегда, требуется транслировать только номер виртуальной страницы. Старшие девять бит виртуального адреса, 0x0, содержат номер таблицы страниц, равный индексу записи в таблице первого уровня. Соответствующая запись в таблице первого уровня указывает, что нужная таблица страниц второго уровня уже располагается в памяти ($V = 1$), а ее физический адрес равен 0x2375000.

Следующие десять бит виртуального адреса, 0x3FE, определяют смещение в таблице страниц, которое дает индекс записи в таблице второго уровня. Запись 0 расположена внизу таблицы второго уровня, а запись 0x3FE наверху. Запись 0x3FE показывает, что виртуальная страница тоже находится в физической памяти ($V = 1$), а номер физической страницы равен 0x23F1. Физический адрес получается путем объединения номера физической страницы и смещения от начала страницы и равен 0x23F1FB0.

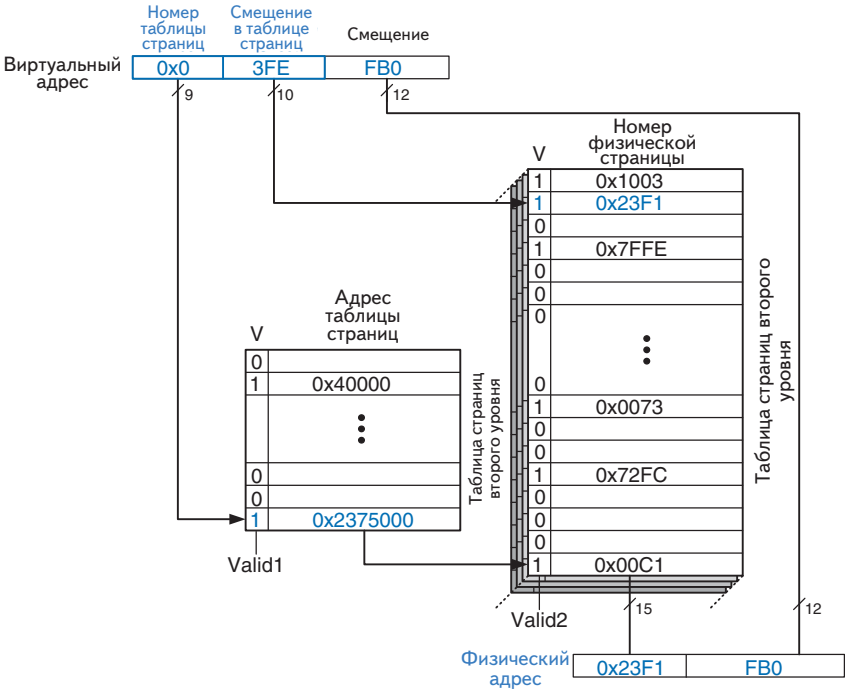


Рис. 3.27. Трансляция адреса с использованием двухуровневой таблицы страниц

3.5. Резюме

Организация системы памяти – важный фактор, влияющий на производительность компьютера. Различные технологии производства памяти, такие как SRAM, DRAM и жесткие диски, позволяют найти компромисс между емкостью, скоростью работы и стоимостью. В этой главе мы рассмотрели организацию иерархии памяти, включающую кэш-память и виртуальную память, которая позволяет разработчикам приблизиться к идеалу – большой, быстрой и дешевой памяти. Оперативная память обычно строится из модулей динамической памяти (DRAM) и работает существенно медленнее, чем процессор. Кэш, в котором часто используемые данные хранятся в гораздо более быстрой статической памяти SRAM, служит для уменьшения времени доступа к оперативной памяти. Виртуальная память позволяет увеличить доступный объем памяти за счет использования жесткого диска, на котором располагаются данные, не помещающиеся в оперативную память. Кэш и виртуальная память требуют дополнительной аппаратуры и усложняют компьютерную систему, но обычно их преимущества перевешивают недостатки. Во всех современных персональных компьютерах используются кэш и виртуальная память.

Упражнения

Упражнение 3.1. В пределах одной страницы опишите четыре повседневных занятия, обладающих временной или пространственной локальностью. Приведите два конкретных примера для каждого типа локальности.

Упражнение 3.2. Одним абзацем опишите два коротких компьютерных приложения, обладающих временной или пространственной локальностью. Опишите, как именно это происходит.

Упражнение 3.3. Придумайте последовательность адресов, для которых кэш с прямым отображением емкостью 16 слов и длиной страницы, равной четырем словам, будет более производительным, чем полностью ассоциативный кэш с такой же емкостью и длиной строки, использующий стратегию вытеснения редко используемых данных (LRU).

Упражнение 3.4. Повторите **упражнение 3.3** для случая, когда полностью ассоциативный кэш более производителен, чем кэш с прямым отображением.

Упражнение 3.5. Опишите компромиссы, связанные с увеличением каждого из следующих параметров кэша при сохранении остальных параметров неизменными:

a) длина строки;
b) степень ассоциативности;
c) емкость кэша.

Упражнение 3.6. Процент промахов у двухсекционного ассоциативного кэша всегда меньше, обычно меньше, иногда меньше или никогда не меньше, чем у кэша с прямым отображением с такой же емкостью и длиной строки? Поясните ответ.

Упражнение 3.7. Утверждения ниже относятся к проценту промахов кэша. Укажите, истинно ли каждое из утверждений. Кратко поясните ход рассуждений; приведите контрпример, если утверждение ложно.

a) Процент промахов у двухсекционного ассоциативного кэша всегда ниже, чем у кэша прямого отображения с такой же емкостью и длиной строки.
b) Процент промахов у 16-килобайтного кэша прямого отображения всегда ниже, чем у 8-килобайтного кэша прямого отображения с той же длиной строки.
c) Процент промахов у кэша команд с 32-байтной строкой обычно ниже, чем у кэша команд с 8-байтной строкой при той же емкости и степени ассоциативности.

Упражнение 3.8. Дан кэш со следующими параметрами: b, длина строки в словах; S, количество наборов; N, количество секций; A, число битов адреса.

a) Выразите через перечисленные параметры емкость кэша C.
b) Выразите через перечисленные параметры число бит, необходимое для хранения тегов.
c) Чему равны S и N для полностью ассоциативного кэша емкостью C слов, длина строки которого равна b?
d) Чему равно S для кэша прямого отображения емкостью C слов, длина строки которого равна b?

Упражнение 3.9. Дан содержащий 16 слов кэш, параметры которого приведены в **упражнении 3.8**. Рассмотрим следующую повторяющуюся последовательность шестнадцатеричных адресов для команд загрузки (`LDR`):

40 44 48 4C 70 74 78 7C 80 84 88 8C 90 94 98 9C 0 4 8 C 10 14 18 1C 20

В предположении, что применяется стратегия вытеснения редко используемых данных (LRU) для ассоциативных кэшей, определите процент промахов при выполнении этой последовательности команд в случае использования одного из перечисленных ниже кэшей. Неизбежными промахами (compulsory misses) пренебречь.

a) кэш прямого отображения, $b = 1$;
b) полностью ассоциативный кэш, $b = 1$;
c) двухсекционный ассоциативный кэш, $b = 1$;
d) кэш прямого отображения, $b = 2$.

Упражнение 3.10. Повторите **упражнение 3.9** для следующей повторяющейся последовательности шестнадцатеричных адресов для команд загрузки (`LDR`) и приведенных ниже конфигураций кэша. Емкость кэша – 16 слов.

74 A0 78 38C AC 84 88 8C 7C 34 38 13C 388 18C

a) кэш прямого отображения, $b = 1$;
b) полностью ассоциативный кэш, $b = 2$;
c) двухсекционный ассоциативный кэш, $b = 2$;
d) кэш прямого отображения, $b = 4$.

Упражнение 3.11. Предположим, что выполняется программа, которая один раз выполняет последовательность обращений к памяти по следующим адресам:

0x0 0x8 0x10 0x18 0x20 0x28

a) Если используется кэш прямого отображения емкостью 1 КБ и длиной строки 8 байт (2 слова), то сколько в кэше наборов?
b) Каков процент промахов кэша прямого отображения для данной последовательности обращений? Емкость и длина строки такие же, как в пункте (a).
c) Что сильнее всего сократит процент промахов при данной последовательности обращений к памяти? Емкость кэша постоянна. Выберите один из вариантов.

   i) Увеличение степени ассоциативности до 2.
   ii) Увеличение длины строки до 16 байт.
   iii) Как (i), так и (ii).
   iv) Ни (i), ни (ii)

Упражнение 3.12. Вы разрабатываете кэш команд для процессора ARM. Его емкость равна $4C = 2^{c+2}$ байт, степень ассоциативности $N = 2^n$ ($N \geq 8$), длина строки $b = 2^{b'}$ байт ($b \geq 8$). Используя эти сведения, ответьте на следующие вопросы.

a) Какие биты адреса используются для выбора слова в строке?
b) Какие биты адреса используются для выбора набора в кэше?
c) Сколько битов в каждом теге?
d) Сколько битов тега во всем кэше целиком?

Упражнение 3.13. Дан кэш со следующими параметрами: N (ассоциативность) = 2, b (длина строки) = 2 слова, W (размер слова) = 32 бит, C (емкость кэша) = 2^{15} слов, A (размер адреса) = 32 бита. Нас интересуют только адреса слов.

a) Какие биты адреса занимают тег, индекс (номер набора), смещение в строке и байтовое смещение? Укажите, сколько бит требуется для каждого из этих полей.
b) Каков общий размер *всех* тегов кэш-памяти в битах?
c) Предположим, что каждая строка кэша содержит бит достоверности (V) и бит изменения (D). Чему равен размер набора кэша, включая данные, тег и служебные биты?
d) Разработайте кэш, используя строительные блоки, показанные на **Рис. 3.28**, и небольшое число двухвходовых логических элементов. Кэш должен включать память тегов, память данных, сравнение адресов, выбор данных, поступающих на выход, а также любые другие части, которые вы сочтете необходимыми. Учтите, что мультиплексоры и компараторы могут быть любой ширины (n и p бит соответственно), но блоки памяти SRAM должны иметь размер 16K × 4 бит. Не забудьте приложить блок-схему с пояснениями. Вам достаточно реализовать только операции чтения.

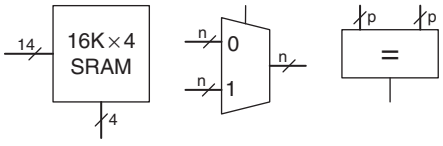


Рис. 3.28. Строительные блоки

Упражнение 3.14. Вы устроились в новый крутой интернет-стартап, разрабатывающий наручные часы со встроенным пейджером и веб-браузером. В разработке используется встроенный процессор с многоуровневой схемой кэширования, изображенной на **Рис. 3.29**. Процессор включает небольшой кэш на кристалле и большой внешний кэш второго уровня (да, часы весят полтора килограмма, но зато по Интернету просто летают!).

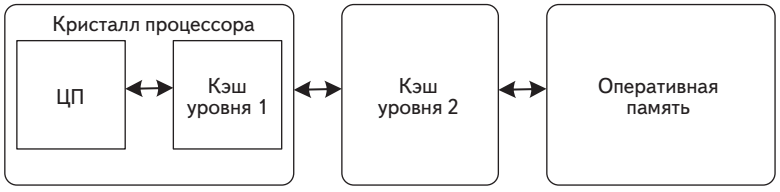


Рис. 3.29. Компьютерная система

Предположим, что процессор использует 32-битовые физические адреса, но обращается к данным только по адресам, выровненным на гра-

ницу слова. Характеристики кэшей приведены в **Табл. 3.5**. Время доступа к оперативной DRAM-памяти равно t_m, а ее размер – 512 МБ.

Таблица 3.5. Характеристики памяти

Характеристика	Кэш на кристалле	Внешний кэш
Способ организации	4-секционный наборно-ассоциативный	Прямого отображения
Процент попаданий	*A*	*B*
Время доступа	t_a	t_b
Длина строка	16 байт	16 байт
Число строк	512	256K

a) Во сколько разных мест в кэше на кристалле и в кэше второго уровня может быть загружено слово, находящееся в памяти?
b) Чему равен размер тега в битах для кэша на кристалле и кэша второго уровня?
c) Напишите выражение для вычисления среднего времени доступа к данным при чтении. Обращение к кэшам происходит по порядку: сначала к кэшу на кристалле, затем к кэшу второго уровня.
d) Измерения показывают, что для некоторой задачи процент попаданий в кэш на кристалле – 85%, а в кэш второго уровня – 90%. Но если кэш на кристалле отключен, то процент попаданий в кэш второго уровня подскакивает до 98.5%. Дайте краткое объяснение такому поведению.

Упражнение 3.15. В этой главе была описана стратегия замещения редко используемых строк (LRU) для многосекционных наборно-ассоциативных кэшей. Существуют и другие, хотя и менее распространенные, стратегии, например: «первым пришел, первым ушел» (first-in-first-out, FIFO) и замещение в случайном порядке. При использовании стратегии FIFO вытесняется та строка, которая находится в кэше дольше всего, независимо от того, как давно к ней обращались. При случайном замещении вытесняемая строка выбирается случайным образом.

a) Каковы достоинства и недостатки каждой стратегии?
b) Опишите последовательность обращений к памяти, при которой FIFO будет работать лучше, чем LRU.

Упражнение 3.16. Вы конструируете компьютер с иерархической системой памяти, состоящей из раздельного кэша команд и данных и оперативной памяти, и используете многотактный процессор ARM, показанный на **Рис. 2.30**, который работает на частоте 1 ГГц.

a) Предположим, что кэш команд работает идеально (т. е. промахов кэша нет), а доля промахов кэша данных – 5%. При промахе кэша процессор приостанавливает выполнение текущей команды на 60 нс, в течение которых происходит доступ к оперативной памяти, после чего возобновляет работу. Чему равно среднее время доступа к памяти с учетом промахов кэша?
b) Сколько тактов на команду (clocks per instruction, CPI) в среднем требуется командам чтения и записи слов, учитывая неидеальность системы памяти?
c) Рассмотрим тестовое приложение из **примера 2.5**, содержащее 25% команд загрузки, 10% команд сохранения, 13% команд перехода и 52% команд обработки данных. Каково среднее значение CPI для этого теста, учитывая неидеальность системы памяти?
d) Теперь предположим, что кэш команд тоже неидеален и промахи имеют место в 7% случаев. Каково тогда среднее значение CPI для теста из пункта (c)? Учитывайте промахи кэша команд и кэша данных.

Упражнение 3.17. Повторите **упражнение 3.16** со следующими параметрами.

a) Кэш команд идеален (т. е. промахов нет), а доля промахов кэша данных – 15%. При промахе процессор приостанавливает выполнение текущей команды на 200 нс, в течение которых обращается к оперативной памяти, а затем возобновляет работу. Чему равно среднее время доступа к памяти с учетом промахов кэша?
b) Сколько тактов на команду (CPI) в среднем требуется командам чтения и записи слов, учитывая неидеальность системы памяти?
c) Рассмотрим тестовое приложение из **примера 2.5**, содержащее 25% команд загрузки, 10% команд сохранения, 12% команд перехода и 52% команд обработки данных. Каково среднее значение CPI для этого теста, учитывая неидеальность системы памяти?
d) Теперь предположим, что кэш команд тоже неидеален – частота промахов равна 10%. Каково тогда среднее значение CPI для теста из пункта (c)? Учитывайте промахи кэша команд и кэша данных.

Упражнение 3.18. Если в компьютере используются 64-битовые виртуальные адреса, то сколько виртуальной памяти он может адресовать? Напомним, что 2^{40} байт = 1 *терабайт*, 2^{50} байт = 1 *петабайт*, а 2^{60} байт = 1 *экзабайт*.

Упражнение 3.19. Разработчик суперкомпьютера решил потратить 1 миллион долларов на оперативную память, изготовленную по технологии DRAM, и столько же на жесткие диски для виртуальной памяти. Сколько физической и виртуальной памяти будет у этого компьютера при ценах, приведенных на **Рис. 3.4**? Какого размера должны быть физические и виртуальные адреса для обращения к такой памяти?

Упражнение 3.20. Рассмотрим подсистему виртуальной памяти, способную адресовать в общей сложности 2^{32} байт. У вас есть неограниченное дисковое пространство, но всего 8 МБ полупроводниковой (физической) памяти. Предположим, что размер физических и виртуальных страниц равен 4 КБ.

a) Чему будет равна длина физического адреса в битах?
b) Каково максимальное число виртуальных страниц в подсистеме?
c) Сколько физических страниц в подсистеме?
d) Каков размер номеров виртуальных и физических страниц в битах?
e) Допустим, вы решили остановиться на схеме прямого отображения виртуальных страниц на физические. При таком отображении для определения номера физической страницы используются младшие биты номера виртуальной страницы. Сколько виртуальных страниц отображается на каждую физическую страницу в этом случае? Почему такое отображение – плохая идея?
f) Очевидно, что необходима более гибкая и динамичная схема трансляции виртуальных адресов в физические, нежели та, что была описана в пункте (e). Предположим, что вы используете таблицу страниц для хранения отображений (то есть информации, позволяющей соотнести номер виртуальной страницы с номером физической). Сколько элементов будет в такой таблице?
g) Предположим, что каждая запись в таблице страниц содержит, помимо номера физической страницы, еще и некоторую служебную информацию, состоящую из битов достоверности (V) и изменения (D). Сколько байтов понадобится для хранения каждой записи? Округлите результат с избытком до ближайшего целого числа байт.
h) Нарисуйте схему размещения таблицы страниц в памяти. Каков общий размер таблицы в байтах?

Упражнение 3.21. Рассмотрим подсистему виртуальной памяти, способную адресовать 2^{50} байт. У вас есть неограниченное дисковое пространство, но всего 2 ГБ полупроводниковой (физической) памяти. Предположим, что размер физической и виртуальной страницы равен 4 КБ.

a) Чему будет равна длина физического адреса в битах?
b) Каково максимальное число виртуальных страниц в подсистеме?

c) Сколько физических страниц в подсистеме?
d) Каков размер номеров виртуальных и физических страниц в битах?
e) Сколько записей будет в таблице страниц?
f) Предположим, что каждая запись в таблице страниц содержит, помимо номера физической страницы, еще и некоторую служебную информацию, состоящую из битов достоверности (V) и изменения (D). Сколько байтов понадобится для хранения каждой записи? Округлите результат с избытком до ближайшего целого числа байт.
g) Нарисуйте схему размещения таблицы страниц в памяти. Каков общий размер таблицы в байтах?

Упражнение 3.22. Вы решили ускорить работу подсистемы виртуальной памяти, описанной в **упражнении 3.20**, при помощи буфера ассоциативной трансляции (TLB). Предположим, что подсистема памяти обладает характеристиками, приведенными в **Табл. 3.6**. Процент промахов TLB и кэша показывает, как часто требуемый элемент будет не найден в соответствующей памяти. Процент промахов оперативной памяти показывает, как часто случаются страничные отказы.

Таблица 3.6. Характеристики памяти

Тип памяти	Время доступа в тактах	Процент промахов
TLB	1	0.05%
Кэш	1	2%
Оперативная память	100	0.0003%
Жесткий диск	1 000 000	0%

a) Каково среднее время доступа в подсистеме виртуальной памяти до и после добавления TLB? Считайте, что таблица страниц всегда расположена в физической памяти и никогда не загружается в кэш данных.
b) Чему равен суммарный размер TLB, состоящего из 64 элементов, в битах? Укажите значения для данных (номеров физических страниц), тегов (номеров виртуальных страниц) и битов достоверности для каждого элемента. Покажите, как вы пришли к таким результатам.
c) Сделайте эскиз TLB. Обозначьте все поля и размеры.
d) SRAM какого размера потребуется для организации TLB, описанного в пункте (c)? Дайте ответ в форме «глубина × ширина».

Упражнение 3.23. Вы решили ускорить систему виртуальной памяти из **упражнения 3.21** с помощью буфера трансляции адресов (TLB) на 128 элементов.

a) Каково суммарное число бит в TLB? Укажите значения для данных (номеров физических страниц), тегов (номеров виртуальных страниц) и битов достоверности для каждого элемента. Покажите, как вы пришли к таким результатам.
b) Нарисуйте эскиз TLB. Обозначьте все поля и их размеры.
c) Блок памяти SRAM какого размера потребуется для организации описанного в пункте (b) TLB? Ответ должен выглядеть как «глубина × ширина».

Упражнение 3.24. Предположим, что в многотактном процессоре ARM, описанном в **разделе 2.4**, используется подсистема виртуальной памяти.

a) Покажите расположение TLB в блок-схеме многотактного процессора.
b) Опишите, как добавление TLB повлияет на производительность процессора.

Упражнение 3.25. В подсистеме виртуальной памяти, которую вы разрабатываете, используется одноуровневая таблица страниц на базе специализированного оборудования (блоки памяти SRAM и сопутствующая логика). Она поддерживает 25-битовые виртуальные адреса, 22-битовые физические адреса и страницы размером 2^{16} байт (64 КБ). В каждой записи таблицы страниц имеется бит достоверности V и бит изменения D.

a) Чему равен размер всей таблицы в битах?
b) Группа разработчиков ОС предлагает сократить размер страницы с 64 до 16 КБ, но разработчики аппаратуры возражают, мотивируя это стоимостью дополнительной аппаратуры. Объясните, почему они против.
c) Таблица страниц будет расположена рядом с кэш-памятью на том же кристалле, что и процессор. Эта кэш-память работает только с физическими (а не виртуальными) адресами. Возможно ли при доступе в память одновременно обращаться к нужному набору в кэше и к таблице страниц? Кратко поясните условия, необходимые для одновременного обращения к набору кэша и к записи таблицы страниц.
d) Возможно ли при доступе в память одновременно выполнять сравнение тегов и обращаться к таблице страниц? Кратко поясните.

Упражнение 3.26. Опишите ситуацию, при которой наличие виртуальной памяти могло бы повлиять на разработку приложения. Подумайте о том, как размеры страницы и физической памяти влияют на производительность приложения.

Упражнение 3.27. Предположим, что имеется персональный компьютер (ПК) с 32-битовыми виртуальными адресами.

a) Чему равен максимальный объем виртуальной памяти, доступный каждой программе?
b) Как влияет на производительность размер жесткого диска ПК?
c) Как влияет на производительность размер физической памяти ПК?

Вопросы для собеседования

Приведенные ниже вопросы задавали на собеседованиях при приеме на работу.

Вопрос 3.1. Объясните разницу между кэшем прямого отображения, наборно-ассоциативным и полностью ассоциативным кэшами. Для каждого типа кэша опишите приложение, для которого кэш данного типа будет эффективнее прочих.

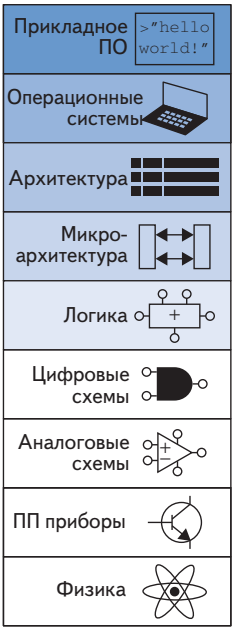
Вопрос 3.2. Объясните, как работают подсистемы виртуальной памяти.

Вопрос 3.3. Объясните достоинства и недостатки использования подсистем виртуальной памяти.

Вопрос 3.4. Объясните, как производительность кэша может зависеть от размера виртуальной страницы в подсистеме виртуальной памяти.

Глава 4

Системы ввода-вывода



4.1. Введение

Системы ввода-вывода (Input/Output, I/O) используются для подключения компьютера к внешним устройствам, которые называются *периферийными*. В персональном компьютере периферийные устройства включают в себя клавиатуры, мониторы, принтеры и беспроводные сети. Во встраиваемых системах – нагревательный элемент тостера, синтезатор речи куклы, топливный инжектор двигателя, двигатели позиционирования солнечных панелей спутника и т. д. Процессор получает доступ к устройствам ввода-вывода, используя шины адреса и данных так же, как при получении доступа к памяти.

В этой главе рассмотрены конкретные примеры устройств ввода-вывода. В **разделе 4.2** описаны основные принципы построения интерфейса между процессором и периферийным устройством и доступа к устройству из программы. В **разделе 4.3** ввод-вывод изучается в контексте

встраиваемых систем и показано, как использовать построенный на базе ARM одноплатный компьютер Raspberry Pi для доступа к размещенным на плате периферийным устройствам – общего назначения, последовательным и параллельным, а также к таймерам. В **разделе 4.4** приведены примеры организации интерфейса с другими распространенными устройствами, в т. ч. символьными светодиодами, VGA-мониторами, радиоприемниками, работающими по технологии Bluetooth, и двигателями. В **разделе 4.5** описаны интерфейсы шин, в качестве иллюстрации выбрана популярная шина AHB-Lite. **Раздел 4.6** посвящен основным системам ввода-вывода, используемым в ПК.

4.2. Ввод-вывод с отображением на память

Напомним (см. **раздел 1.5.1**), что часть адресного пространства отводится под устройства ввода-вывода вместо памяти. Предположим, к примеру, что под эти цели зарезервированы физические адреса от 0x20000000 до 0x20FFFFFF. Каждому устройству ввода-вывода назначается один или несколько адресов в этом диапазоне. При записи по такому адресу данные передаются устройству. При чтении происходит получение данных от устройства. Этот метод связи с устройствами ввода-вывода называется вводом-выводом с отображением на память (memory-mapped I/O, MMIO).

В системе с MMIO команды загрузки или сохранения могут осуществлять доступ либо к памяти, либо к устройству ввода-вывода. На **Рис. 4.1** изображены аппаратные средства, необходимые для поддержки двух устройств ввода-вывода, отображенных на память. Дешифратор адреса определяет, какое устройство связывается с процессором. Он использует сигналы *Address* и *MemWrite* для формирования управляющих сигналов для остального оборудования. Мультиплексор *ReadData* производит выбор между памятью и различными устройствами ввода-вывода. В регистрах с разрешением записи хранятся значения, записываемые в устройства ввода-вывода.

Пример 4.1. ОБМЕН ДАННЫМИ С УСТРОЙСТВАМИ ВВОДА-ВЫВОДА

Допустим, что устройству ввода-вывода 1 на **Рис. 4.1** назначен адрес памяти 0x20001000. Напишите на языке ассемблера ARM код для записи значения 7 в устройство ввода-вывода 1 и для чтения данных из устройства ввода-вывода 1.

Решение. Приведенный ниже ассемблерный код для ARM записывает значение 7 в устройство ввода-вывода 1.

```
        MOV R1, #7
        LDR R2, =ioadr
        STR R1, [R2]
ioadr DCD 0x20001000
```

Дешифратор адреса устанавливает сигнал *WE1*, потому что адрес равен 0x20001000 и сигнал *MemWrite* равен TRUE. Данные на шине *WriteData*, значение 7, записываются в регистр, который подключен к входным контактам устройства ввода-вывода 1.

Чтобы считать данные из устройства ввода-вывода 1, процессор должен выполнить команду:

```
LDR R1, [R2]
```

Дешифратор адреса устанавливает $RDsel_{1:0}$ в 01, поскольку видит, что адрес равен 0x20001000 и значение *MemWrite* равно FALSE. Выходные данные из устройства ввода-вывода 1 поступают через мультиплексор на шину *ReadData* и загружаются в регистр R1 процессора.

Адреса, связанные с устройствами ввода-вывода, часто называют *регистрами ввода-вывода*, потому что они могут соответствовать физическим регистрам устройства, как показано на **Рис. 4.1**.

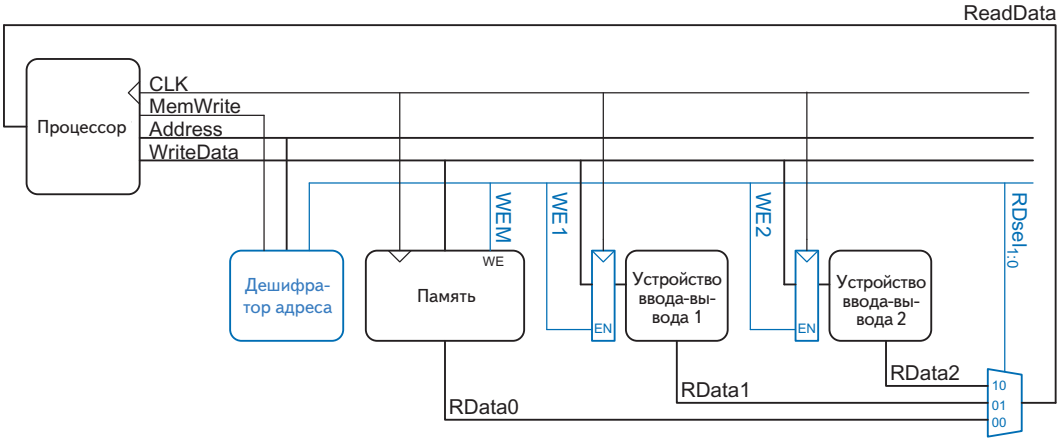


Рис. 4.1. Отображение устройств ввода-вывода на память

Программное обеспечение, которое взаимодействует с устройством ввода-вывода, называется *драйвером устройства*. Вы наверняка загружали или устанавливали драйверы для вашего принтера или другого устройства ввода-вывода. Написание драйвера требует детального знания аппаратной части устройства, в т. ч. адресов и поведения отображенных на память регистров ввода-вывода. Другие программы вызывают функции драйвера для получения доступа к устройству, при этом понимать низкоуровневые детали работы оборудования им не обязательно.

В некоторых архитектурах и прежде всего в x86 для взаимодействия с устройствами ввода-вывода используются специализированные команды, а не отображение на память. Эти команды имеют вид

```
LDRIO R1, device1
STRIO R2, device2
```

где `device1` и `device2` — уникальные идентификаторы периферийных устройств. Такой тип взаимодействия называется программируемым вводом-выводом.

В 2014 году было продано микроконтроллеров на сумму около 19 млрд долларов, и прогнозируется, что к 2020 году этот рынок достигнет 27 млрд долларов. В среднем микроконтроллер стоит менее 1 доллара, а 8-битовые микроконтроллеры можно интегрировать в систему на кристалле менее чем за один цент. Микроконтроллеры стали вездесущими и практически невидимыми: предположительно по 150 штук в каждом доме и по 50 — в каждом автомобиле в 2010 году. 8051 — классический 8-битовый микроконтроллер, который первоначально был разработан компанией Intel в 1980 году, а сейчас поставляется целым рядом производителей. Изделия компании Microchip серий PIC16 и PIC18 — лидеры на рынке 8-битовых микроконтроллеров. Серия Atmel AVR стала популярной среди энтузиастов-любителей в качестве «мозга» платформы Arduino. На рынке 32-битовых микроконтроллеров ведущие позиции занимает компания Renesas. Другие крупные производители микроконтроллеров — компании Freescale, Samsung, Texas Instruments и Infineon. Процессоры ARM стоят почти во всех смартфонах и планшетах и обычно являются частью системы на кристалле, содержащей многоядерный процессор приложений, графический процессор и развитую систему ввода-вывода.

4.3. Ввод-вывод во встраиваемых системах

Во встраиваемых системах процессор используется для управления взаимодействиями с физической средой. Обычно такие системы выстроены вокруг *микроконтроллеров* (МК), которые сочетают в себе микропроцессор с набором простых в использовании периферийных устройств: цифровых и аналоговых контактов ввода-вывода общего назначения, последовательных портов, таймеров и т. д. Как правило, МК недорогие и спроектированы так, чтобы минимизировать стоимость и размеры системы путем интеграции большинства необходимых компонентов на одном кристалле. Большинство из них меньше и легче, чем монета в десять центов, потребляют милливатты мощности, и цена их варьируется в пределах от нескольких десятков центов до нескольких долларов. Микроконтроллеры классифицируются по размеру данных, которыми оперируют. 8-битовые микроконтроллеры – самые маленькие и самые дешевые, в то время как 32-битовые предоставляют больше памяти и имеют большую производительность.

В качестве примера в этом разделе будет показан ввод-вывод в реальной встраиваемой системе. Конкретно речь пойдет о популярной и недорогой плате Raspberry Pi, которая содержит систему на кристалле (SoC) Broadcom BCM2835 с 32-битовым процессором ARM1176JZ-F с тактовой частотой 700 МГц и набором команд ARMv6. Принципы, изложенные в каждом подразделе, будут проиллюстрированы примерами, работающими на Pi. Все примеры протестированы на компьютере Pi, работающем под управлением ОС NOOBS Raspbian Linux, в 2014 году.

На **Рис. 4.2** представлена фотография платы Raspberry Pi Model B+, представляющей собой полноценный компьютер под управлением ОС Linux размером с половину кредитной карты, который продается за 35 долларов. Pi потребляет ток до 1 А от 5-вольтового USB-источника питания. На плате размещено ОЗУ емкостью 512 МБ и гнездо для SD-карты памяти, на которой записаны операционная система и файлы пользователя. Имеются разъемы для видео- и аудиовыходов, USB-порты для мыши и

клавиатуры, порт локальной сети Ethernet (Local Area Network), а также 40 контактов ввода-вывода общего назначения (GPIO), которые и будут представлять основной интерес в этой главе.

Система BCM2835 обладает многими возможностями, которых нет в типичном недорогом микроконтроллере, но ввод-вывод общего назначения организован почти так же, как в любом другом устройстве. Мы начнем эту главу с описания SoC BCM2835 в Raspberry Pi и драйвера устройства для ввода-вывода, отображенного на память. А затем проиллюстрируем, как во встраиваемых системах производится цифровой, аналоговый и последовательный вводы-выводы общего назначения. Для генерирования тактовых сигналов и точного измерения временных интервалов также используются таймеры.

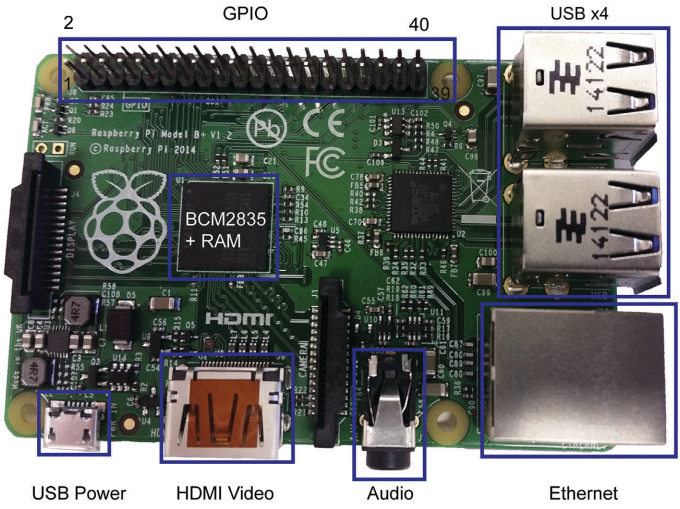
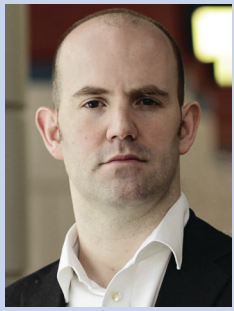


Рис. 4.2. Raspbery Pi Model B+

4.3.1. Система на кристалле BCM2835

Система на кристалле (SoC) BCM2835 – мощная и вместе с тем дешевая микросхема, разработанная компанией Broadcom для мобильных устройств и других мультимедийных приложений. Система включает микропроцессор ARM, который называется *процессором приложений*, процессор VideoCore для обработки графики, видео и фотографий, а также многочисленные периферийные устройства ввода-вывода. BCM2835 заключена в пласти-

Компьютер Raspberry Pi был разработан в 2011–2012 годах некоммерческим фондом Raspberry Pi Foundation в Великобритании в качестве платформы для преподавания информатики. Компьютер, построенный на основе недорогого смартфона, стал безумно популярен – к 2014 году было продано свыше 3 миллионов экземпляров. Название (raspberry – малина) – дань памяти первым домашним компьютерам, которые назывались Apple (яблоко), Apricot (абрикос) и Tangerine (мандарин). Аббревиатура Pi – производное от Python, языка программирования, широко используемого в образовательной сфере. Документацию и сведения о порядке приобретения можно найти на сайте raspberrypi.org.



Эбен Аптон (1978–)

Эбен Аптон – архитектор Raspberry Pi и основатель фонда Raspberry Pi Foundation. Получил степени бакалавра и доктора философии в Кэмбриджском университете, после чего поступил на работу в Broadcom Corporation на должность архитектора микросхем.

(Фотография воспроизводится с разрешения.)

Raspberry Pi продолжает развиваться, и вполне возможно, что к моменту, когда вы будете читать этот текст, появится новая модель с более мощным процессором и другим комплектом встраиваемых средств ввода-вывода. Но к ней, да и к другим типам микроконтроллеров все равно будут применимы те же принципы. Скорее всего, вы обнаружите такие же типы периферии. Чтобы выяснить, как соответствуют друг другу периферийное устройство, контакт на микросхеме и контакт на плате, а также узнать адреса отображенных на память регистров ввода-вывода, ассоциированных с каждым устройством, необходимо будет справиться с технической документацией. Для инициализации устройства нужно записать определенные значения в конфигурационные регистры, а для доступа к нему — читать и писать в регистры данных.

Когда эта книга уже была отправлена в печать, фонд Raspberry Pi Foundation выпустил модель Raspberry Pi 2 Model B с SoC-системой BCM2836, укомплектованную 4-ядерным процессором Cortex-A7 и 1 ГБ памяти. Pi 2 работает примерно в 6 раз быстрее B+, но оснащена теми же средствами ввода-вывода, что описаны в данной главе. Базовый адрес периферийных устройств поменялся с 0x20000000 на 0x3F000000. Обновленный код драйвера EasyPIO, поддерживающий обе модели, выложен на сайте книги[1].

[1] На момент составления перевода (2018 год) последней была плата Raspberry Pi 3+ Model B с SoC-системой BCM2837B0 на четырехядерном 64-битовом процессоре Cortex-A53 и с 1 ГБ памяти. — *Прим. перев.*

ковом корпусе BGA (ball grid array — массив шариковых выводов), на нижней плоскости которого расположены крохотные шариковые выводы из припоя; такой корпус лучше всего припаивается с помощью робота, который совмещает выводы с медными контактными площадками на печатной плате и создает нужную температуру. Broadcom не публикует полные технические данные, но на сайте Raspberry Pi можно найти краткую спецификацию, в которой описано, как получить доступ к периферии со стороны процессора ARM. В спецификации приведены сведения о многочисленных возможностях и регистрах ввода-вывода, которые в этой главе для простоты опущены.

```
www.raspberrypi.org/documentation/hardware/
```

На **Рис. 4.3** показана упрощенная схема макетной платы Raspberry Pi. Плата получает питание от 5-вольтового USB-источника, а регуляторы выдают напряжение 3.3, 2.5 и 1.8 В для ввода-вывода, аналоговых устройств и прочих функций. BCM2835 снабжена также внутренним импульсным регулятором, который выдает переменное напряжение для энергоэффективной SoC-системы. BCM2835 подключена к контроллеру USB/Ethernet и имеет прямой видеовыход. У нее есть также 54 конфигурируемых сигнала ввода-вывода, но из-за слишком малого размера только часть из них доступна пользователю посредством штыревых контактов. На штыревые контакты выведены также напряжение 3.3 и 5 В и земля, чтобы было удобно подавать питание на небольшие устройства, подключенные к Pi, но максимальный полный ток составляет 50 мА от источника 3.3 В и −300 мА от источника 5 В. Модели B и B+ похожи, но в B+ число контактов ввода-вывода увеличено с 26 до 40, а число USB-портов — с 2 до 4. Для подключения этих контактов к макетной плате имеются различные кабели, в т. ч. Adafruit Pi Cobbler.

В Raspberry Pi в качестве флеш-памяти используется SD-карта. Обычно на карту заранее записывается Raspbian Linux — сокращенная версия Linux, умещающаяся на карте емкостью 8 ГБ. Для работы с Pi можно либо подключить HDMI-монитор и мышь и клавиатуру USB — тогда получится полноценный компьютер, либо подключить ее к другому компьютеру по Ethernet-кабелю.

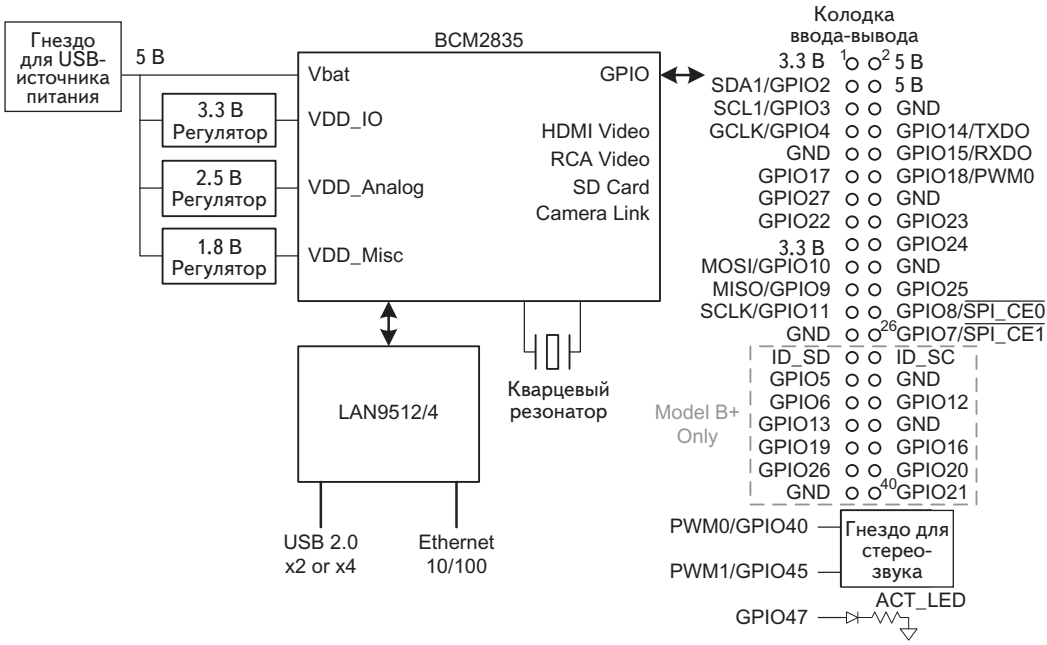


Рис. 4.3. Схема ввода-вывода Raspberry Pi

4.3.2. Драйверы устройств

Программист может манипулировать устройствами ввода-вывода напрямую посредством чтения или записи в отображенные на память регистры ввода-вывода. Но считается более правильным вызывать для доступа к ним специальные функции, которые называются *драйверами устройств*. Перечислим некоторые преимущества драйверов.

- Код проще читать, если в нем упоминается функция с понятным именем, а не производится запись в битовые поля, расположенные неизвестно по какому адресу.
- Драйвер мог быть написан человеком, хорошо знакомым с тонкостями работы устройства, а непосвященный пользователь может вызывать его, не вникая в детали.
- Код легко перенести на другой процессор с иным отображением на память или иными устройствами ввода-вывода, поскольку заменить нужно только драйвер.

Предостережение: разводка контактов ввода-вывода зависит от ревизии платы Raspberry Pi.

Предостережение: подача напряжения 5 В на один из контактов 3.3 В приведет к повреждению системы ввода-вывода и выходу из строя всей платы Raspberry Pi. При проверке контактов вольтметром следите за тем, чтобы случайно не замкнуть 5-вольтовый контакт с соседним!
Контакт 1 колодки ввода-вывода на Рис. 4.3 помечен. Прежде чем производить подключение, убедитесь, что вы правильно идентифицировали его и не повернули плату на 180 градусов. Это распространенная ошибка, в результате которой можно случайно испортить Pi.

- Если драйвер входит в состав операционной системы, то ОС может управлять совместным доступом к физическому устройству со стороны нескольких программ и контролировать безопасность (например, чтобы вредоносная программа не могла читать клавиатуру, когда вы вводите пароль в браузере).

В этом разделе мы разработаем простой драйвер устройства EasyPIO для доступа к устройствам BCM2835, чтобы вы понимали, как устроен драйвер изнутри. Обычные пользователи предпочитают библиотеку ввода-вывода с открытым исходным кодом WiringPi, в которой имеются функции, похожие, но не точно совпадающие с имеющимися в EasyPIO.

Код EasyPIO и примеров, приведенных в этой главе, можно скачать с сайта книги по адресу http://booksite.elsevier.com/9780128000564. Драйвер WiringPi и документация к нему имеются на сайте wiringpi.com.

В системе BCM2835 ввод-вывод отображен на диапазон физических адресов 0x20000000–0x20FFFFFF. Соответствие между периферийными устройствами и базовыми адресами приведено в **Табл. 4.1**. С каждым периферийным устройством ассоциировано несколько регистров, начиная с его базового адреса. Например, чтение по адресу 0x20200034 возвращает значения контактов GPIO 31:0. Устройства, выделенные полужирным шрифтом, будут рассмотрены в последующих разделах.

Таблица 4.1. Адреса отображенных на память устройств ввода-вывода

Физический базовый адрес	Периферийное устройство
0x20003000	Системный таймер
0x2000B200	Прерывания
0x2000B400	Таймер ARM
0x20200000	GPIO
0x20201000	UART0
0x20203000	PCM-аудио
0x20204000	SPI0
0x20205000	Первый ведущий микроконтроллер I^2C
0x2020C000	PWM
0x20214000	Ведомые устройства I^2C
0x20215000	miniUART1, SPI1, SPI2
0x20300000	Контроллер SD-карты
0x20804000	Второй ведущий микроконтроллер I^2C
0x20805000	Третий ведущий микроконтроллер I^2C

Raspberry Pi обычно работает под управлением операционной системы Linux с виртуальной памятью, что дополнительно усложняет отображенный на память ввод-вывод. В командах загрузки и сохранения указываются виртуальные, а не физические адреса, поэтому программа не может обратиться к отображенным на память регистрам непосредственно, а должна сначала запросить у операционной системы физические адреса, соответствующие интересующим ее виртуальным. В драйвере EasyPIO эту задачу решает функция `pioInit`, показанная в **примере 4.2**. В ее коде присутствуют низкоуровневые манипуляции с указателями на C. Общая идея состоит в том, чтобы открыть устройство `/dev/mem` – это принятый в Linux метод доступа к физической памяти. С помощью функции `mmap` переменной `gpio` присваивается указатель на физический адрес 0x20200000 – начало набора регистров GPIO. Указатель объявлен как `volatile`, поэтому компилятор знает, что значение по этому адресу может измениться асинхронно, и, следовательно, программа обязана всякий раз читать значение регистра, а не полагаться на старое значение. Константа GPLEV0 соответствует регистру ввода-вывода, отстоящему от GPIO на 13 слов, по адресу 0x20200034; он содержит значения контактов GPIO 31:0. Для краткости в этом примере опущена проверка ошибок, но в реальной библиотеке EasyPIO она присутствует. В следующих подразделах описаны дополнительные регистры и функции для доступа к устройствам ввода-вывода.

Из соображений безопасности Linux дает доступ к оборудованию, отображенному на память, только *суперпользователю*. Чтобы выполнить программу от имени суперпользователя, добавьте перед именем команды слово `sudo`. Пример будет приведен в следующем разделе.

Пример 4.2. ИНИЦИАЛИЗАЦИЯ ВВОДА-ВЫВОДА, ОТОБРАЖЕННОГО НА ПАМЯТЬ

```
#include <sys/mman.h>
#define BCM2835_PERI_BASE    0x20000000
#define GPIO_BASE            (BCM2835_PERI_BASE + 0x200000)
volatile unsigned int        *gpio; // Указатель на начало gpio
#define GPLEV0               (* (volatile unsigned int *) (gpio + 13))
#define BLOCK_SIZE           (4*1024)

void pioInit(){
   int mem_fd;
   void *reg_map;

   // /dev/mem - драйвер псевдоустройства для доступа к памяти в Linux
   mem_fd = open("/dev/mem", O_RDWR|O_SYNC);
   reg_map = mmap(
     NULL,    // начальный адрес локального отображения (null = не важно)
     BLOCK_SIZE,          // 4КБ - размер отображаемого блока памяти
     PROT_READ|PROT_WRITE, // разрешены чтение и запись в отображенную память
     MAP_SHARED,          // немонопольный доступ к этой памяти
     mem_fd,              // отобразить на /dev/mem
     GPIO_BASE);          // базовый адрес периферийного устройства GPIO
```

```
    gpio = (volatile unsigned *)reg_map;
    close(mem_fd);
}
```

4.3.3. Цифровой ввод-вывод общего назначения

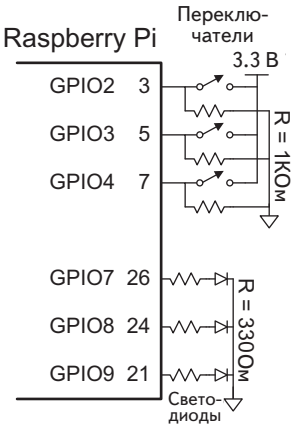


Рис. 4.4. Светодиоды и переключатели, подключенные к контактам GPIO

Контакты ввода-вывода общего назначения (GPIO) используются для чтения или записи цифровых сигналов. Например, на **Рис. 4.4** показаны три светодиода (LED) и три переключателя, подключенных к шести контактам GPIO. Светодиоды подключены таким образом, чтобы зажигаться, когда приходит логическая «1», и гаснуть, когда приходит «0». Последовательно со светодиодами включены токоограничивающие резисторы, которые задают яркость и предотвращают превышение допустимой нагрузки по току на GPIO. Переключатели подключены так, что выдают «1», когда замкнуты, и «0», когда разомкнуты. На схеме показаны названия контактов и номер соответствующего контакта в колодке.

Как минимум, любому регистру GPIO необходимы регистры для чтения входных значений, для записи выходных значений и задания направления контакта. Во многих встраиваемых системах контакты GPIO могут использоваться также одним или несколькими специализированными устройствами, поэтому необходимы дополнительные конфигурационные регистры, которые определяют, является контакт общим или специальным. Кроме того, процессор может генерировать прерывания при обнаружении на входном контакте положительного или отрицательного фронта сигнала, и с помощью конфигурационных регистров можно указать, при каких условиях должно возникать прерывание.

В контексте битовых операций «установка» означает запись в бит значения 1, а «очистка» — запись значения 0.

Напомним, что в BCM2835 имеется 54 контакта GPIO. Для управления ими служат регистры GPFSEL, GPLEV, GPSET, GPCLR. На **Рис. 4.5** показано, как эти регистры отображены на память. Регистры GPFSEL5...0 определяют, является контакт общим или специальным. В каждом из этих регистров выбора функции отведено по 3 бита на контакт, поэтому один 32-битовый регистр управляет 10 контактами GPIO, как показано в **Табл. 4.2**, и, следовательно, для управления всеми 54 контактами необходимо шесть регистров GPFSEL. Например, GPIO13 конфигурируется битами GPFSEL1[11:9]. Конфигурации описаны в **Табл. 4.3**; у многих контактов есть специальные функции, которые

будут рассмотрены в последующих разделах; чаще всего используется ALT0. Чтение регистров GPLEV1...0 возвращает значения контактов. Например, значение GPIO14 читается из регистра GPLEV0[14], а значение GPIO34 – из регистра GPLEV1[2]. Прямая запись в контакты запрещена; вместо этого следует установить соответствующий бит GPSET1...0 или GPCLR1...0. Например, чтобы перевести контакт GPIO14 в состояние «1», нужно записать GPSET0[14] = 1, а для перевода в состояние «0» – записать GPCLR0[14] = 1.

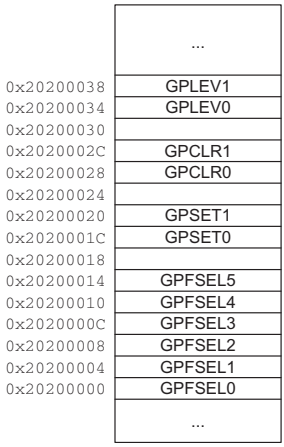


Рис. 4.5. Отображение регистров GPIO на память

Таблица 4.2. Отображение битовых полей регистров GPFSEL на GPIO

	GPFSEL0	GPFSEL1	GPFSEL2	GPFSEL3	GPFSEL4	GPFSEL5
[2:0]	GPIO0	GPIO10	GPIO20	GPIO30	GPIO40	GPIO50
[5:3]	GPIO1	GPIO11	GPIO21	GPIO31	GPIO41	GPIO51
[8:6]	GPIO2	GPIO12	GPIO22	GPIO32	GPIO42	GPIO52
[11:9]	GPIO3	GPIO13	GPIO23	GPIO33	GPIO43	GPIO53
[14:12]	GPIO4	GPIO14	GPIO24	GPIO34	GPIO44	
[17:15]	GPIO5	GPIO15	GPIO25	GPIO35	GPIO45	
[20:18]	GPIO6	GPIO16	GPIO26	GPIO36	GPIO46	
[23:21]	GPIO7	GPIO17	GPIO27	GPIO37	GPIO47	
[26:24]	GPIO8	GPIO18	GPIO28	GPIO38	GPIO48	
[29:27]	GPIO9	GPIO19	GPIO29	GPIO39	GPIO49	

Таблица 4.3. Конфигурация GPFSEL

GPFSEL	Функция контакта
000	Ввод
001	Вывод
010	ALT5
011	ALT4
100	ALT0
101	ALT1
110	ALT2
111	ALT3

В BCM2835 доступ в GPIO организован необычно сложно. В некоторых микроконтроллерах используется всего один регистр, чтобы задать направление каждого контакта – ввод или вывод, и еще один для чтения и записи контактов.

Пример 4.3. GPIO ДЛЯ ПЕРЕКЛЮЧАТЕЛЕЙ И СВЕТОДИОДОВ

Включите в EasyPIO функции `pinMode`, `digitalRead` и `digitalWrite` для задания направления контакта, а также чтения и записи его значения. Напишите на C программу, в которой эти функции будет использованы для чтения всех трех переключателей и включения соответствующих светодиодов с применением оборудования, изображенного на **Рис. 4.4**.

Решение. Ниже показан добавленный в EasyPIO код. Поскольку управление вводом-выводом осуществляется с использованием нескольких регистров, функции должны вычислять, к какому регистру обратиться и какие биты в нем изменять. После этого `pinMode` очищает 0-биты и устанавливает 1-биты в соответствии с требуемой 3-битовой функцией, `digitalWrite` отвечает за запись 1 или 0 с помощью регистров GPSET или GPCLR, а `digitalRead` читает значение указанного контакта и маскирует значения остальных.

```
#define GPFSEL  ((volatile unsigned int *) (gpio + 0))
#define GPSET   ((volatile unsigned int *) (gpio + 7))
#define GPCLR   ((volatile unsigned int *) (gpio + 10))
#define GPLEV   ((volatile unsigned int *) (gpio + 13))
#define INPUT   0
#define OUTPUT  1
...
void pinMode(int pin, int function) {
  int reg = pin/10;
  int offset = (pin%10)*3;
  GPFSEL[reg] &= ~((0b111 & ~function) << offset);
  GPFSEL[reg] |= ((0b111 & function) << offset);
}

void digitalWrite(int pin, int val) {
  int reg = pin / 32;
  int offset = pin % 32;

  if (val) GPSET[reg] = 1 << offset;
  else GPCLR[reg] = 1 << offset;
}

int digitalRead(int pin) {
  int reg = pin / 32;
  int offset = pin % 32;

  return (GPLEV[reg] >> offset) & 0x00000001;
}
```

Ниже приведена программа, которая читает переключатели и записывает светодиоды. Она инициализирует доступ к GPIO, затем назначает контакты 2–4 вводами переключателей, а контакты 7–9 – выводами светодиодов. После этого она в бесконечном цикле читает переключатели и записывает значения в соответствующие контакты светодиодов.

```
#include "EasyPIO.h"

void main(void) {
```

```
    pioInit();

    // Назначить GPIO 4:2 вводами
    pinMode(2, INPUT);
    pinMode(3, INPUT);
    pinMode(4, INPUT);

    // Назначить GPIO 9:7 выводами
    pinMode(7, OUTPUT);
    pinMode(8, OUTPUT);
    pinMode(9, OUTPUT);

    while (1) { // Читать переключатели и записывать соответствующие светодиоды
        digitalWrite(7, digitalRead(2));
        digitalWrite(8, digitalRead(3));
        digitalWrite(9, digitalRead(4));
    }
}
```

В предположении, что файл программы называется dip2led.c и что файл EasyPIO.h находится в том же каталоге, мы можем откомпилировать и запустить программу на Raspberry Pi, введя показанные ниже команды. Здесь `gcc` – компилятор C, а `sudo` необходима, для того чтобы программа могла обращаться к защищенной памяти, на которую отображены регистры ввода-вывода. Для остановки программы нажмите **Ctrl+C**.

```
gcc dip2led.c -o dip2led
sudo ./dip2led
```

4.3.4. Последовательный ввод-вывод

Если микроконтроллеру необходимо послать больше битов, чем количество свободных контактов GPIO, то придется разбить сообщение на несколько меньших пакетов. На каждом шаге можно посылать либо один бит, либо несколько битов. Первый метод называется *последовательным вводом-выводом*, второй – *параллельным вводом-выводом*. Последовательный ввод-вывод популярен, потому что используется небольшое число проводов и скорость достаточно велика для многих задач. На самом деле он настолько популярен, что для него разработано немало стандартов, и микроконтроллеры предлагают специализированное оборудование, чтобы было проще передавать данные в соответствии с этими стандартами. В этом разделе описываются протоколы последовательного периферийного интерфейса (serial peripheral interface, SPI) и универсального асинхронного приемопередатчика (universal asynchronous receiver/transmitter, UART).

Широко распространены и другие стандарты последовательного ввода-вывода: последовательная, а симметричная шина для связи между интегральными схемами внутри электронных приборов (inter-integrated circuit, I^2C), универсальная последовательная шина (universal serial bus, USB) и Ethernet. I^2C (произносится «ай квадрат си») – двухпроводной интерфейс с сигналом синхронизации и двунаправленным портом данных; он используется примерно так же, как SPI. USB и Ethernet – более сложные высокопроизводительные стандарты; они описаны в **разделах 4.6.1** и **4.6.4** соответственно. Все пять стандартов поддержаны на платформе Raspberry Pi.

Последовательный периферийный интерфейс SPI

Последовательный периферийный интерфейс SPI – простой синхронный последовательный протокол, легкий в использовании и относительно быстрый. Физический интерфейс состоит из трех контактов: Serial Clock (SCK), Master Out Slave In (MOSI, иногда его еще называют SDO) и Master In Slave Out (MISO, или SDI). SPI подключает *ведущее* (master) устройство к *ведомому* (slave), как показано на **Рис. 4.6 (a)**. Ведущее устройство генерирует сигнал синхронизации. Оно инициирует взаимодействие, посылая серию тактовых импульсов на контакт SCK. Если оно хочет передать данные ведомому устройству, то подает их на контакт MOSI, начиная со старшего бита. Ведомое устройство может одновременно отвечать, посылая данные на контакт MISO. На **Рис. 4.6 (b)** показаны формы сигналов SPI при передаче 8-битовых данных. Биты изменяются по отрицательному фронту сигнала SCK и сохраняют стабильность на положительном фронте, когда их можно читать. Интерфейс SPI позволяет также посылать разрешающий сигнал с активным низким уровнем (SPI_CE0), чтобы уведомить приемник о скором поступлении данных.

В SPI данные всегда передаются в обоих направлениях в каждом акте обмена. Если системе требуется только одностороннее взаимодействие, то она может игнорировать ненужные данные. Например, если ведущее устройство должно только отправлять данные ведомому, то байт, полученный от ведомого устройства, игнорируется. Если ведущее устройство хочет только получать данные от ведомого, то все равно должно инициировать обмен по протоколу SPI, отправив произвольный байт, который ведомое устройство проигнорирует. Сигнал синхронизации SPI меняет состояние, только когда ведущее устройство передает данные.

В BCM2835 есть три SPI ведущих порта и один ведомый. В этом разделе описывается ведущий порт 0, который на плате Raspberry Pi доступен через контакты GPIO 11:9. Чтобы использовать эти контакты для SPI, а не для GPIO, необходимо установить контакт GPFSEL в ALT0. Затем Pi должна сконфигурировать порт. Pi пишет в SPI, данные последовательно передаются ведомому устройству. Одновременно от ведомого устройства принимаются данные, и Pi сможет прочитать их по завершении передачи.

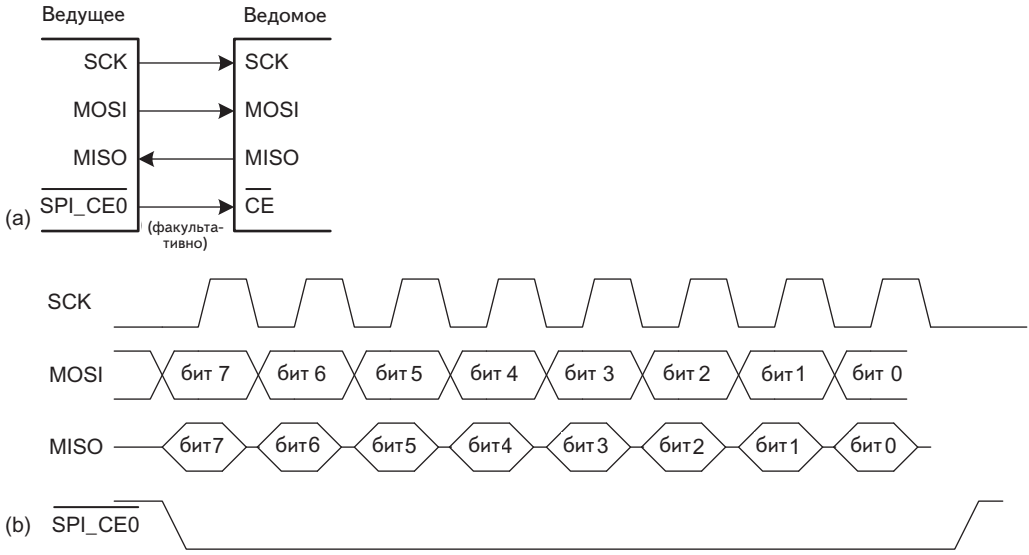


Рис. 4.6. Соединения и формы сигналов в SPI

Таблица 4.4. Поля регистра SPI0CS

Бит	Название	Функция	Если равен 0	Если равен 1
16	DONE	Передача завершена	Передача продолжается	Передача завершена
7	TA	Передача активна	SPI запрещен	SPI разрешен
3	CPOL	Полярность сигнала синхронизации	Сигнал синхронизации начинается с низкого уровня	Сигнал синхронизации начинается с высокого уровня
2	CPHA	Фаза сигнала синхронизации	Выборка данных производится по переднему фронту сигнала синхронизации	Выборка данных производится по заднему фронту сигнала синхронизации

Ведущий порт 0 в SPI ассоциирован с тремя регистрами при отображении на память, показанном на **Рис. 4.7**. SPI0CS – регистр управления. Он используется для включения SPI и установки таких атрибутов, как полярность сигнала синхронизации. В **Табл. 4.4** перечислены названия и функции некоторых битов SPI0CS, имеющих отношение к теме обсуждения. После сброса все они по умолчанию равны 0. Многие функции, например выбор микросхемы и прерывания, в этом разделе не используются, но описаны в технической документации. Запись в регистр

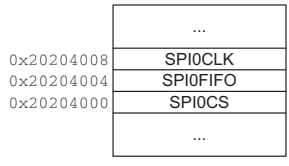


Рис. 4.7. Регистры ведущего порта 0 в SPI

SPI0FIFO производится для передачи байта и чтения байта, полученного в ответ. Регистр SPI0CLK служит для конфигурирования тактовой частоты SPI, для этого частота периферийного устройства 250 МГц делится на степень двойки, заданную в регистре. В **Табл. 4.5** перечислены значения тактовой частоты SPI.

Если частота слишком высока (>~1 МГц на макетной плате или десятки МГц на открытой печатной плате), то интерфейс SPI может работать ненадежно из-за отражений, взаимных помех и других проблем, угрожающих целостности сигнала.

Таблица 4.5. Частоты, задаваемые в регистре SPI0CLK

SPI0CLK	Частота SPI (КГц)
2	125 000
4	62 500
8	31 250
16	15 625
64	7812
128	1953

Пример 4.4. ПЕРЕДАЧА И ПРИЕМ БАЙТОВ ЧЕРЕЗ ИНТЕРФЕЙС SPI

Разработайте систему для обмена данными между ведущим устройством Raspberry Pi и ведомой ПЛИС (FPGA) через интерфейс SPI. Набросайте в общем виде схему интерфейса. Напишите на C программу для Pi, которая посылает символ 'A' и принимает его обратно. Напишите на HDL код для ведомой части интерфейса SPI, которая находится в ПЛИС. Как можно упростить эту часть интерфейса, если требуется только получать данные?

Решение. На **Рис. 4.8** показано, как соединить устройства, используя ведущий порт 0 SPI. Номера контактов взяты из спецификаций компонентов (например, см. **Рис. 4.3**). Обратите внимание, что на схеме приведены номера контактов и названия сигналов, чтобы показать как физические, так и логические соединения. Когда интерфейс SPI включен, эти контакты нельзя использовать для GPIO.

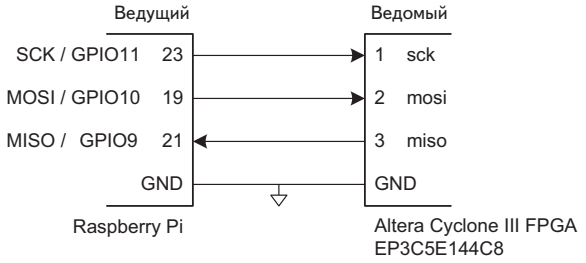


Рис. 4.8. Соединение интерфейсов SPI в Pi и ПЛИС

Следующий код в файле EasyPIO.h предназначен для инициализации SPI, а также отправки и получения символа. Определение отображения на память и адресов регистров делается почти так же, как для GPIO, поэтому повторять этот код мы не стали.

```
void spiInit(int freq, int settings) {
  pinMode(8, ALT0);          // CEOb
  pinMode(9, ALT0);          // MISO
  pinMode(10, ALT0);         // MOSI
  pinMode(11, ALT0);         // SCLK

  SPI0CLK = 250000000/freq;  // Задать делитель для получения
                             // нужной тактовой частоты
  SPI0CS = settings;
  SPI0CSbits.TA = 1;         // Включить SPI
}

char spiSendReceive(char send){
  SPI0FIFO = send;           // Отправить данные ведомому устройству
  while (!SPI0CSbits.DONE); // Ждать завершения SPI
  return SPI0FIFO;           // Вернуть полученные данные
}
```

Приведенный ниже код на C инициализирует SPI, а затем отправляет и принимает один символ. Тактовая частота SPI устанавливается равной 244 КГц.

```
#include "EasyPIO.h"

void main(void) {
  char received;

  pioInit();
  spiInit(244000, 0);  // Инициализировать SPI: тактовая частота 244 КГц,
                       // параметры по умолчанию
  received = spiSendReceive('A'); // Отправить букву A и получить байт в ответ
}
```

HDL-код для ПЛИС приведен ниже, а блок-схема и временная диаграмма показаны на **Рис. 4.9**. ПЛИС использует сдвиговый регистр для хранения битов, полученных от ведущего контроллера, и тех битов, которые ожидают отправки. После сигнала *Reset*, начиная с первого переднего фронта сигнала SCK и в течение 8 последующих тактов, в сдвиговый регистр загружается новый байт из `d`. На каждом последующем такте сигнала бит вдвигается в `mosi` и выдвигается из `miso`. `miso` приостанавливается до завершения заднего фронта `sck` и может быть прочитан ведущим устройством на следующем переднем фронте сигнала. По прошествии 8 тактов принятый байт будет находиться в `q`.

```
module spi_slave(input  logic sck,       // От ведущего
                 input  logic mosi,      // От ведущего
                 output logic miso,      // К ведущему
                 input  logic reset,     // Сброс системы
                 input  logic [7:0] d,   // Данные, подлежащие отправке
                 output logic [7:0] q);  // Полученные данные

  logic [2:0] cnt;
  logic       qdelayed;

  // 3-битовый счетчик отслеживает момент передачи полного байта
  always_ff @(negedge sck, posedge reset)
```

```
    if (reset) cnt = 0;
    else cnt = cnt + 3'b1;

  // Загружаемый сдвиговый регистр
  // Вначале загружается d, бит из mosi вдвигается в конец на каждом шаге
  always_ff @(posedge sck)
    q <= (cnt == 0) ? {d[6:0], mosi} : {q[6:0], mosi};

  // Совместить miso с задним фронтом sck
  // Загрузить d в начальный момент
  always_ff @(negedge sck)
    qdelayed = q[7];
  assign miso = (cnt == 0) ? d[7] : qdelayed;
endmodule
```

Если ведомое устройство должно только принимать данные от ведущего, то HDL-код можно упростить, обойдясь лишь сдвиговым регистром.

```
module spi_slave_receive_only(input  logic sck,      // От ведущего
                              input  logic mosi,     // От ведущего
                              output logic [7:0] q); // Полученные данные
  always_ff @(posedge sck)
    q <= {q[6:0], mosi};                             // Сдвиговый регистр
endmodule
```

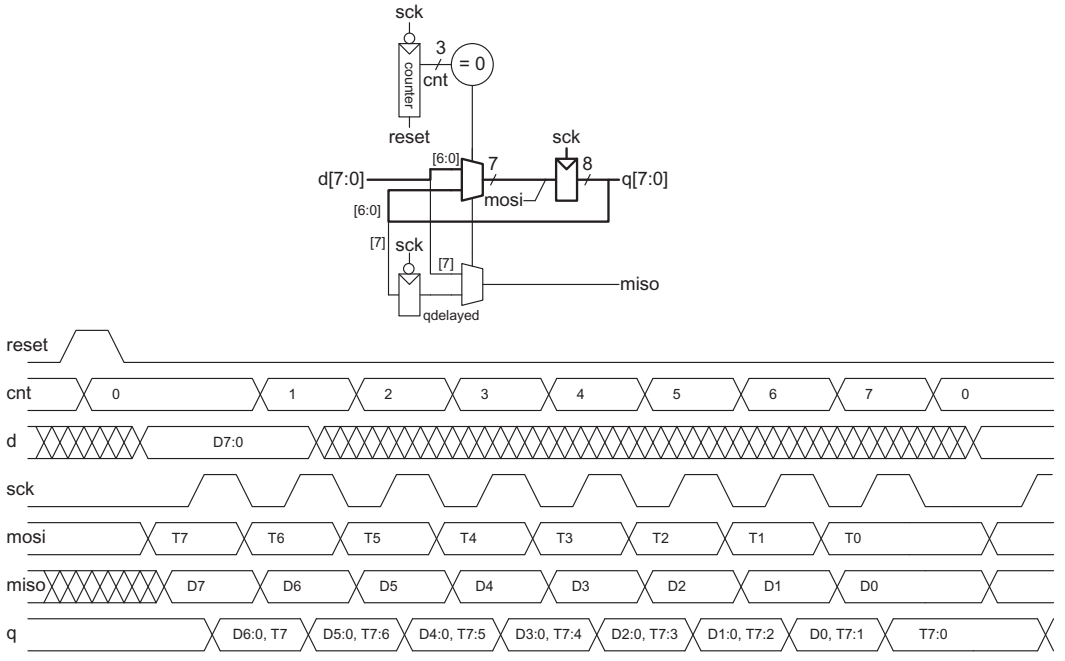


Рис. 4.9. Блок-схема ведомого устройства SPI и его временная диаграмма

Порты SPI конфигурируются в широких пределах, поэтому интерфейс годится для связывания между собой разнообразных последовательных устройств. К сожалению, при этом открывается возможность сконфигурировать порт неправильно, что ведет к искажению передаваемых данных. Иногда необходимо изменить конфигурационные биты для взаимодействия с устройством, имеющим другую временную диаграмму. Если CPOL = 1, то SCK инвертируется. Если CPHA= 1, то сигнал синхронизации меняет знак на полтакта раньше по отношению к данным. Эти режимы показаны на **Рис. 4.10**. Необходимо учитывать, что в различных изделиях, поддерживающих SPI, могут применяться разные названия и полярности сигналов; внимательно изучите формы сигналов для своего устройства. Также полезно проверить сигналы SCK, MOSI и MISO осциллографом, если при взаимодействии возникают проблемы.

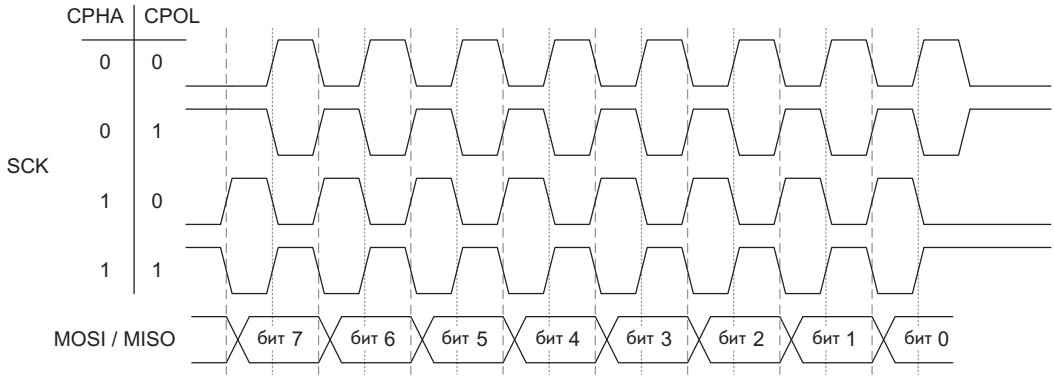


Рис. 4.10. Временные диаграммы сигнала синхронизации и данных в SPI

Универсальный асинхронный приемопередатчик (UART)

UART (произносится как «юарт») является периферийным устройством последовательного ввода-вывода для обмена данными между двумя системами без сигнала синхронизации. Вместо этого системы должны заранее договориться о скорости передачи данных, и каждая из них должна локально генерировать свой собственный тактовый сигнал. Поскольку тактовые сигналы не синхронизированы, передача является асинхронной. Хотя системные тактовые сигналы могут немного отличаться по частоте, а соотношение фаз неизвестно, UART обеспечивает надежную асинхронную связь. UART используется в таких протоколах, как RS-232 и RS-485. Например, в последовательных портах старых компьютеров использовался стандарт RS-232C, введенный в 1969 году Ассоциацией электронной промышленности (Electronic Industries Association). Стандарт первоначально предполагалось использовать для под-

ключения *оконечного оборудования данных* (data terminal equipment, DTE), например мейнфреймов, к *оборудованию передачи данных* (data communication equipment, DCE), например модему. Хотя UART относительно медленный, по сравнению с SPI, и чувствителен к ошибкам конфигурирования, стандарт существует так долго, что остается важным и по сей день.

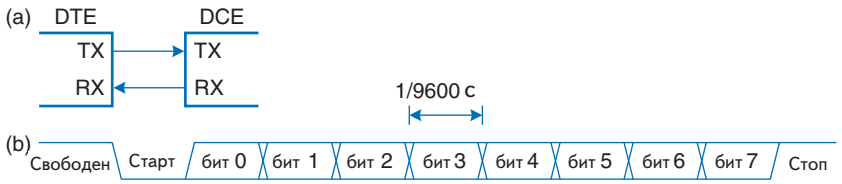


Рис. 4.11. Асинхронный последовательный канал связи

Скорость передачи в бодах (baud rate) — это скорость передачи сигналов, измеряемая в символах в секунду, а скорость передачи в битах (bit rate) — скорость передачи данных, измеряемая в битах в секунду. В этой книге мы обсуждали двухуровневую передачу сигналов, когда каждый символ представляется одним битом. Но бывает и многоуровневая сигнализация, когда символ представлен несколькими битами. Например, при 4-уровневой сигнализации на символ отводится два бита. В таком случае скорость передачи в битах будет в два раза больше скорости передачи в бодах. В простой системе типа SPI, где каждый символ — это бит и каждый символ представляет данные, скорость передачи в бодах равна скорости передачи в битах. В UART и при некоторых других соглашениях о сигнализации нужны дополнительные биты, помимо информационных. Например, в двухуровневой системе сигнализации с добавлением стартового и стопового битов для каждых 8 бит данных, которая работает со скоростью передачи в бодах 9600, скорость передачи в битах будет равна (9600 символов/с)×(8 бит/10 символов) = 7680 бит/с = 960 знаков/с.

На **Рис. 4.11 (a)** показан асинхронный последовательный канал связи. DTE посылает данные DCE по линии TX и получает данные обратно по линии RX. На **Рис. 4.11 (b)** показано, как по одной из этих линий передается знак со скоростью передачи данных в бодах 9600. Линия свободна, когда сигнал принимает значение «1». Каждый отправленный знак состоит из стартового бита (0), 7 или 8 битов данных, необязательного бита четности и одного или более стоповых битов (1). UART обнаруживает спадание уровня сигнала с 1 до 0 и в этот момент переходит в состояние передачи. Хотя семи бит данных достаточно для передачи знака в кодировке ASCII, обычно используется восемь бит, потому что в таком случае можно передать произвольный байт данных.

Необязательный *бит четности* позволяет системе проверить наличие ошибки в одном бите во время передачи. Его можно настроить на контроль с дополнением *до четного* или *до нечетного*; в первом случае бит четности выбирается так, чтобы в объединении байта данных с битом четности было четное число единиц. Иными словами, бит четности вычисляется как ИСКЛЮЧАЮЩЕЕ ИЛИ (XOR) битов данных. Приемник может затем проверить, было ли получено четное число единиц или нет, и сигнализировать об ошибке. Контроль с дополнением до нечетного аналогичен, только общее число единиц должно быть нечетным.

Обычно выбирают 8 бит данных без контроля четности и 1 стоповый бит, что в общей сложности составляет 10 символов для передачи 8-битового знака. Поэтому скорость передачи измеряется в бодах, а не в бит/с. На-

пример, 9600 бод означает 9600 символов/с, или 960 знаков/с. Обе системы должны быть настроены одинаково, иначе данные будут искажены. Это затруднительно, особенно для пользователей без технической подготовки, что явилось одной из причин, по которой универсальная последовательная шина Universal Serial Bus (USB) заменила UART в персональных компьютерах.

Типичные скорости передачи данных: 300, 1200, 2400, 9600, 14 400, 19 200, 38 400, 57 600 и 115 200 бод. Малые значения скорости использовались в 1970-х и 1980-х годах для модемов, которые передавали данные по телефонным линиям. В современных системах наиболее распространены скорости 9600 и 115 200; 9600 встречается там, где скорость не имеет значения, а 115 200 является самой быстрой стандартной скоростью, хотя и медленной, по сравнению с другими современными стандартами последовательного ввода-вывода.

В 1950–1970-х годах первые хакеры, называвшие себя телефонными фрикерами, научились управлять коммутаторами телефонных компаний свистом соответствующей частоты. Звук с частотой 2600 Гц, производимый игрушечным свистком из коробки с кукурузными хлопьями «Капитан Кранч», позволял бесплатно совершать междугородные и международные звонки.

Рис. 4.12. Боцманский свисток капитана Кранча. (Печатается с разрешения Evrim Sen)

В стандарте RS-232 определено несколько дополнительных сигналов. Сигналы запроса на передачу (request to send, RTS) и готовности к передаче (clear to send, CTS) могут использоваться для *аппаратного подтверждения связи* (handshaking). Они могут работать в одном из двух режимов. В *режиме управления потоком* DTE сбрасывает RTS в 0, когда готов к приему данных от DCE. Точно так же DCE устанавливает CTS в 0, когда готов к приему данных от DTE. В некоторых технических описаниях используется надчеркивание, чтобы указать, что разрешающий сигнал – низкого уровня. В старом режиме односторонней передачи (*симплексном*) DTE обнуляет RTS, когда готов к передаче. DCE отвечает обнулением CTS, когда готов к приему.

Подтверждением связи (англ. *handshaking* – «рукопожатие») называют процедуру согласования сеанса двумя системами; обычно одна система сообщает о том, что готова отправлять или принимать данные, а другая отвечает на запрос подтверждением.

В некоторых системах, особенно связанных по телефонной линии, используются также сигналы Data Terminal Ready (DTR), Data Carrier Detect (DCD), Data Set Ready (DSR) и Ring Indicator (RI), показывающие, когда оборудование подключено к линии.

Оригинальный стандарт рекомендовал массивный 25-контактный разъем DB-25, но с появлением ПК он подвергся рационализации – остался 9-контактный разъем-вилка DE-9 типа с разводкой, показанной на **Рис. 4.13 (a)**. Провода обычно подключаются напрямую, как показано на **Рис. 4.13 (b)**. Однако когда непосредственно соединяются два DTE, может понадобиться *нуль-модемный* кабель, показанный на **Рис. 4.13 (c)**, в котором RX и TX поменяны местами. Чтобы еще осложнить дело, встречаются как разъемы-вилки (разъем типа «папа»), так и разъемы-розетки (разъем типа «мама»). В итоге для соединения двух систем по интерфейсу RS-232 иногда приходится гадать и иметь под рукой

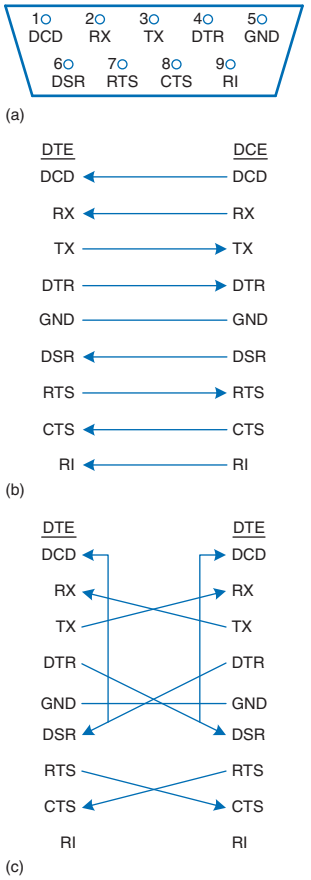


Рис. 4.13. Кабель DE-9 с разъемом-вилкой: a) разводка, b) стандартное соединение, c) нуль-модемное соединение

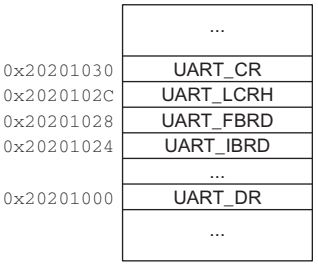


Рис. 4.14. Регистры UART0

большую коробку с кабелями. И это еще одна причина перехода к USB. К счастью, во встраиваемых системах обычно используется упрощенная 3- или 5-проводная разводка, включающая контакты GND, TX, RX и, возможно, RTS и CTS.

В RS-232 логический «0» электрически представляется напряжением от 3 В до 15 В, а «1» – от −3 В до −15 В; это называется *биполярной сигнализацией*. Приемопередатчик преобразует цифровые логические уровни UART в положительные и отрицательные уровни напряжения, ожидаемые RS-232, а также обеспечивает защиту от электростатического разряда, чтобы предотвратить повреждение последовательного порта в момент подключения кабеля пользователем. MAX3232E – популярный приемопередатчик, совместимый с уровнями напряжения 3.3 В и 5 В. Он содержит схему накачки заряда (charge pump), которая в сочетании с внешними конденсаторами генерирует выходное напряжение ±5 В от одного источника питания низкого напряжения. В некоторых последовательных периферийных устройствах для встраиваемых систем вообще нет приемопередатчика – для представления логического «0» используется напряжение 0 В, а для «1» – 3.3 В или 5 В; читайте документацию!

В BCM2835 имеется два UART'а: UART0 и UART1. Какой из них использовать для приема и передачи данных, задается с помощью контактов 14 и 15, но UART0 обладает более широкой функциональностью, его мы и опишем. Чтобы использовать эти контакты для UART0, а не для GPIO, их сигнал GPFSEL должен иметь значение ALT0. Как и в случае SPI, плата Pi должна первым делом сконфигурировать порт. Но, в отличие от SPI, чтение и запись могут производиться независимо, потому что любая система может передавать данные, ничего не принимая, или наоборот. Регистры UART0 показаны на **Рис. 4.14**.

Для конфигурирования UART сначала задается скорость передачи в бодах. В UART имеется внутренний тактовый генератор с частотой 3 МГц, которую нужно поделить для получения сигнала синхронизации с частотой, в 16 раз превышающей требуемую скорость передачи в бодах. Следовательно, делитель BRD вычисляется по формуле

$$BRD = 3\,000\,000 / (16 \times \text{скорость передачи в бодах}).$$

BRD представляется в виде 16-битовой целой части

в регистре UART_IBRD и 6-битовой дробной части в регистре UART_FBRD: BRD= IBRD + FBRD / 64. В **Табл. 4.6** их значения приведены для популярных скоростей передачи[1].

Таблица 4.6. Значения делителя BRD

Требуемая скорость передачи в бодах	UART_IBRD	UART_FBRD	Фактическая скорость передачи в бодах	Погрешность (%)
300	625	0	300	0
1200	156	16	1200	0
2400	78	8	2400	0
9600	19	34	9600	0
19 200	9	49	19 200	0
38 400	4	56	38 461	0.16
57 600	3	16	57 692	0.16
115 200	1	40	115 384	0.16

Затем с помощью регистра управления линией UART_LCRH задается количество бит данных, количество стоповых бит и режим контроля четности. По умолчанию подразумевается 1 стоповый бит, без контроля четности, но – как ни странно – в режиме по умолчанию передаются и принимаются 5-битовые слова. Чтобы задать 8-битовые слова, в поле WLEN (биты 6:5) регистра UART_LCRH следует записать 3. Наконец, включите UART, подняв бит 0 (UARTEN) регистра управления UART_CR.

Данные передаются и принимаются с помощью регистра данных UART_DR и регистра кадровой синхронизации UART_FR. Для передачи данных дождитесь, когда бит 7 (TXFE) в регистре UART_FR станет равен 1 (передатчик не занят), после чего запишите байт в UART_DR. Для приема данных дождитесь, когда бит 4 (RXFE) в регистре UART_FR станет равен 0 (в приемнике есть данные), после чего прочитайте байт из UART_DR.

Пример 4.5. ПОСЛЕДОВАТЕЛЬНЫЙ ОБМЕН ДАННЫМИ С ПЕРСОНАЛЬНЫМ КОМПЬЮТЕРОМ

Разработайте схему и напишите программу на C для Raspberry Pi, которая будет осуществлять обмен данными с персональным компьютером через последовательный порт со скоростью 115 200 бод в формате: 8 бит данных, 1 стоповый бит, без контроля четности. На компьютере должна быть запущена консоль, на-

[1] Не все скорости передачи нацело делят частоту 3 МГц, поэтому при некоторых значениях делителя частота вычисляется с погрешностью. Но в силу своей асинхронной природы UART справляется с этой погрешностью, если она не слишком велика.

пример PuTTY[2], для передачи данных через последовательный порт. Программа должна попросить пользователя ввести строку, а затем сообщить пользователю, что именно он ввел.

Решение. На **Рис. 4.15 (a)** показана схема последовательного соединения. Поскольку почти не осталось компьютеров с последовательными портами, мы используем внешний адаптер Plugable USB to RS-232 DB9 Serial Adapter компании `plugable.com`, показанный на **Рис. 4.16**. Этот адаптер подключается кабелем с разъемом-розеткой DE-9 к приемопередатчику, который преобразует биполярные уровни напряжения RS-232 к уровню 3.3 В, используемому в Pi. Плата Pi и персональный компьютер являются оконечными устройствами данных, поэтому контакты TX и RX должны быть соединены накрест. Аппаратный контроль передачи данных RTS/CTS в Pi не используется, так что контакты RTS и CTS на разъеме DE9 соединены друг с другом, чтобы персональный компьютер всегда договаривался сам с собой – процедура заведомо успешная. На **Рис. 4.15 (b)** показан более простой подход с применением последовательного кабеля Adafruit 954 для соединения USB с TTL. Этот кабель совместим с уровнями 3.3 В, и его розетка точно подходит к вилке Raspberry Pi.

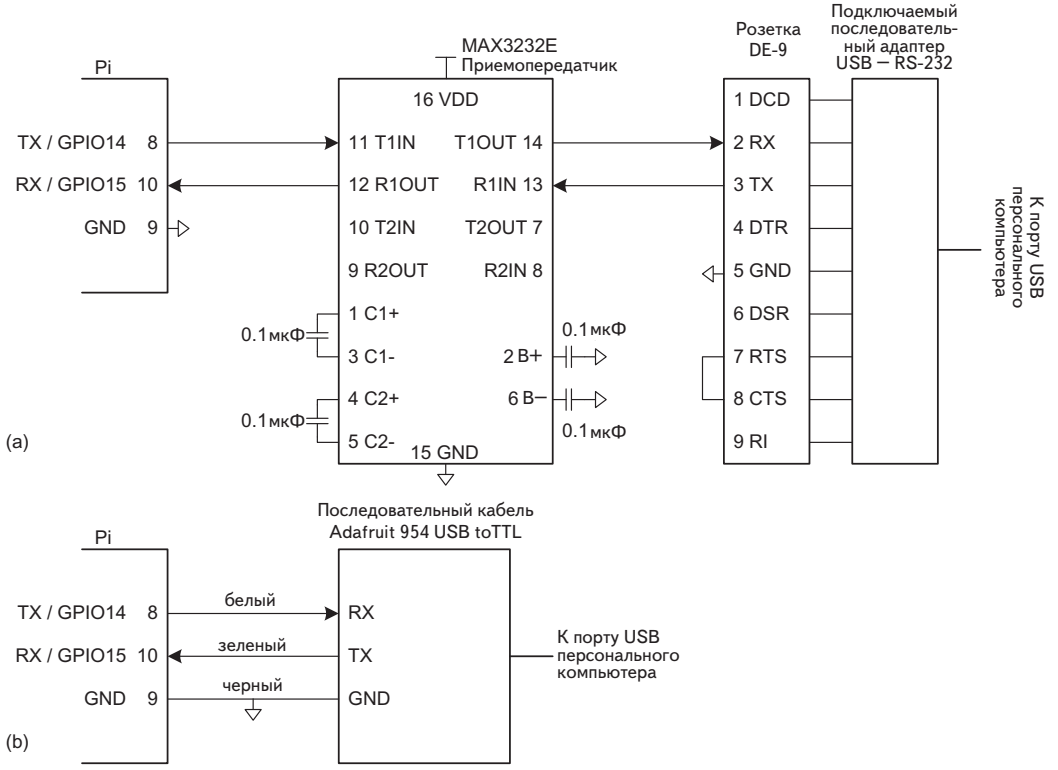


Рис. 4.15. Последовательное подключение Pi к ПК: (a) подключаемый кабель; (b) кабель Adafruit

[2] Программа PuTTY доступна для скачивания и свободного использования на сайте *www.putty.org*.

При настройке PuTTY для работы с последовательным соединением установите переключатель *Connection type* равным *Serial*, а в поле *Speed* запишите 115 200. В поле *Serial line* укажите номер COM-порта, назначенного операционной системой USB-адаптеру. В операционной системе Windows номер порта можно найти в диспетчере устройств. Например, это может быть COM3. На странице *Connection* → *Serial* задайте режим управления потоком (*Flow control*) NONE или RTS/CTS. На странице *Terminal* установите переключатель *Local Echo* равным *Force On*, чтобы символы отображались в консоли по мере ввода.

Рис. 4.16. Подключаемый адаптер USB – RS-232 DB9

Отметим, что операционная система также выводит приглашение к входу в последовательный порт. Поэтому в случае, когда и ОС, и ваша программа пользуются этим портом, могут возникать любопытные взаимодействия.

Ниже приведен код драйвера последовательного порта в файле EasyPIO.h. Клавише Enter соответствует знак возврата каретки, который в программе на C представляется как `'\r'` и имеет ASCII-код 0x0D. Для перехода на следующую строку в напечатанном тексте нужно отправить оба знака `'\n'` и `'\r'` (новая строка и возврат каретки)[3]. Функция `uartInit` конфигурирует UART, как было описано выше. Функции `getCharSerial` и `putCharSerial` ждут готовности UART, после чего соответственно читают байт данных из регистра или записывают его в регистр.

```
void uartInit(int baud) {
  uint fb = 12000000/baud;    // тактовый генератор UART с частотой 3 МГц

  pinMode(14, ALT0);          // TX
  pinMode(15, ALT0);          // RX
  UART_IBRD = fb >> 6;  // 6 бит дробной части и 16 бит целой части делителя BRD
  UART_FBRD = fb & 63;
  UART_LCRHbits.WLEN = 3;  // 8 бит данных, 1 стоповый бит, без контроля
                           // четности, без FIFO, без управления потоком
  UART_CRbits.UARTEN = 1;  // Включить uart
}

char getCharSerial(void) {
  while (UART_FRbits.RXFE);    // Ждать готовности данных
  return UART_DRbits.DATA;     // Вернуть знак из последовательного порта
}

void putCharSerial(char c) {
  while (!UART_FRbits.TXFE);  // Ждать готовности к передаче
  UART_DRbits.DATA = c;       // Отправить знак в последовательный порт
}
```

Функция `main` демонстрирует печать на консоль и чтение с консоли с помощью функций `putStrSerial` и `getStrSerial`.

```
#include "EasyPIO.h"

#define MAX_STR_LEN 80

void getStrSerial(char *str) {
```

[3] PuTTY печатает правильно, даже если знак `\r` опущен.

```
  int i = 0;
  do {  // Читать строку до ввода
     str[i] = getCharSerial();       // знака возврата каретки
  } while ((str[i++] != '\r') && (i < MAX_STR_LEN));  // Искать возврат
                                                      // каретки
  str[i-1] = 0;  // Null-terminate the string
}

void putStrSerial(char *str) {
  int i = 0;
  while (str[i] ! = 0) {        // Перебрать все знаки в строке
     putCharSerial(str[i++ ]);  // Отправить каждый знак
  }
}

int main(void) {
  char str[MAX_STR_LEN];

  pioInit();
  uartInit(115200);           // Инициализировать UART, задать
                              // скорость передачи в бодах

  while (1) {
     putStrSerial("Please type something: \r\n");
     getStrSerial(str);
     putStrSerial("You typed: ");
     putStrSerial(str);
     putStrSerial("\r\n");
  }
}
```

Обмен через последовательный порт из программы на C, работающей на ПК, несколько затруднителен, потому что библиотеки, работающие с драйвером последовательного порта, в разных операционных системах различны. В других программных средах, например Python, Matlab или LabVIEW, этой проблемы нет.

4.3.5. Таймеры

Встраиваемые системы обычно нуждаются в измерении времени. Например, микроволновой печи необходим один таймер для отслеживания времени суток и еще один для задания времени готовки. Третий таймер может генерировать импульсы двигателя, вращающего блюдо, а четвертый – контролировать уровень мощности, активируя микроволны лишь на долю каждой секунды.

В BCM2835 имеется системный таймер с 64-разрядным автономным счетчиком, который увеличивается на 1 каждую микросекунду (т. е. с частотой 1 МГц), и четыре 32-битовых канала сравнения с таймером. На **Рис. 4.17** показано отображение системного таймера на память.

Регистры SYS_TIMER_CLO и CHI содержат старшую и младшую половину 64-битового значения. Регистры SYS_TIMER_C0...C3 – 32-битовые каналы сравнения. Если какой-то канал сравнения совпадает с SYS_TIMER_CLO, то устанавливается соответствующий бит совпадения (M0–M3) – один из четырех младших битов регистра SYS_TIMER_CS. Бит совпадения очищается путем записи в него 1. На первый взгляд, это противоречит интуиции, но так сделано, чтобы предотвратить случайную очистку других битов совпадения. Чтобы отсчитать определенное число микросекунд, следует прибавить это время к CLO, записать его в C1, очистить бит SYS_TIMER_CS.M1, а затем подождать, когда бит SYS_TIMER_CS.M1 окажется установленным.

Графический процессор и операционная система могут использовать каналы 0, 2 и 3 для своих надобностей, поэтому пользователю следует проверять бит SYSTEM_TIMER_C1.

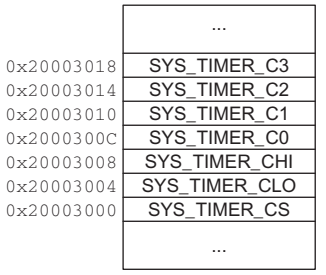


Рис. 4.17. Регистры системного таймера

К несчастью, Linux – многозадачная операционная система, которая может переключать процессы без предупреждения. Если ваша программа ждет совпадения с таймером, а в это время начинает выполняться другой процесс, то программа может не возобновить выполнение в течение долгого времени после совпадения, и время будет измерено неправильно. Чтобы избежать этого, программа может запретить прерывания на время критического хронометража, чтобы Linux не переключала процессы. Но не забудьте потом разрешить прерывания. В EasyPIO определены функции `noInterrupts` и `interrupts`, которые соответственно запрещают и разрешают прерывания. Если прерывания запрещены, то Pi не будет переключать процессы и даже не сможет отреагировать на нажатие **Ctrl+C** для снятия программы. Если программа зависнет, то придется выключить питание и перезагрузить Pi.

Пример 4.8. МИГАЮЩИЙ СВЕТОДИОД

Напишите программу, которая заставляет светодиод состояния на плате Raspberry Pi мигать с частотой 5 раз в секунду на протяжении 4 секунд.

Решение. Функция `delayMicros` в файле EasyPIO организует задержку на заданное количество микросекунд, пользуясь каналом 1 сравнения с таймером.

```
void delayMicros(int micros) {
     SYS_TIMER_C1 = SYS_TIMER_CLO + micros;     // Установить регистр
                                                // сравнения
     SYS_TIMER_CSbits.M1 = 1;             // Сбросить флаг совпадения в 0
     while (SYS_TIMER_CSbits.M1 == 0);    // Ждать, когда будет установлен
                                          // флаг совпадения
}

void delayMillis(int millis) {
     delayMicros(millis*1000);            // 1000 мкс в 1 мс
}
```

На плате Pi B + светодиодом управляет контакт GPIO47. Программа назначает этот контакт выводом и запрещает прерывания. Затем она гасит и зажигает светодиод, несколько раз производя запись с частотой повторения один раз в 200 мс (5 Гц). В конце программа разрешает прерывания.

```
#include "EasyPIO.h"

void main(void) {
     int i;

     pioInit();

     pinMode(47, OUTPUT);    // Контакт светодиода состояния - вывод
     noInterrupts();         // Запретить прерывания

     for (i = 0; i<20; i ++) {
          delayMillis(150);
          digitalWrite(47, 0);  // Погасить светодиод
          delayMillis(50);
          digitalWrite(47, 1);  // Зажечь светодиод
     }
     interrupts();              // Разрешить прерывания
}
```

4.3.6. Аналоговый ввод-вывод

Наш мир аналоговый. Многие встраиваемые системы нуждаются в аналоговых входах и выходах для взаимодействия с миром. В них используются аналого-цифровые преобразователи (АЦП) для квантования аналоговых сигналов по уровню с целью получения цифровых значений и цифроаналоговые преобразователи (ЦАП), выполняющие обратную операцию. На **Рис. 4.18** показаны условные графические обозначения этих компонентов. Такие преобразователи характеризуются разрешением, динамическим диапазоном, частотой дискретизации и точностью. Например, АЦП может иметь N = 12-битовое разрешение в диапазоне от V_{ref}^- до V_{ref}^+ – от 0 до 5 В – с частотой дискретизации f_s = 44 КГц и точностью ±3 младших бита. Частота дискретизации также измеряется в отсчетах в секунду (samples per second, sps), где 1 sps = 1 Гц. Отношение между напряжением аналогового входа $V_{in}(t)$ и цифровой выборкой $X[n = t / f_s]$ можно выразить следующим образом:

$$X[n] = 2^N \frac{V_{in}(t) - V_{ref-}}{V_{ref+} - V_{ref-}}.$$

Например, входное напряжение 2.5 В (половина полной шкалы) будет соответствовать выходу $100000000000_2 = 800_{16}$ с погрешностью не более 3 младших разрядов.

Аналогично ЦАП может иметь N = 16-битовое разрешение в полном диапазоне выходных напряжений относительно V_{ref} = 2.56 В. Выдаваемое им напряжение равно:

$$V_{out}(t) = \frac{X[n]}{2^N} V_{ref}$$

Во многих микроконтроллерах имеются встроенные АЦП средней производительности. Для получения более высокой производительности (например, 16-битовое разрешение или частота дискретизации свыше 1 МГц) часто необходимо использовать отдельный АЦП, соединенный с микроконтроллером. Реже встречаются микроконтроллеры со встроенным ЦАП, так что здесь также приходится использовать отдельные микросхемы. Тем не менее микроконтроллеры часто формируют аналоговые выходы, применяя метод *широтно-импульсной модуляции* (ШИМ).

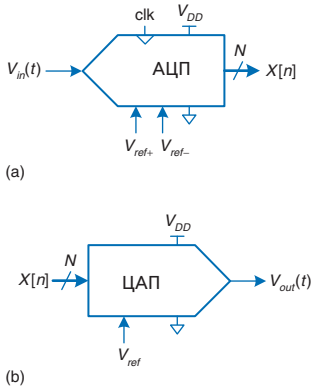


Рис. 4.18. Условные обозначения АЦП и ЦАП

Цифроаналоговое преобразование

В BCM2835 имеется специализированный ЦАП для формирования композитного видеосигнала, но преобразователя общего назначения нет. В этом разделе мы рассмотрим цифроаналоговое преобразование с применением внешнего ЦАП и проиллюстрируем интерфейс работы Raspberry Pi с другими микросхемами по параллельному и последовательному портам. А в следующем разделе тот же результат будет получен с помощью широтно-импульсной модуляции.

Некоторые ЦАПы принимают N-битовый цифровой входной сигнал по параллельному интерфейсу с N проводами, другие – по последовательному интерфейсу, например SPI. Одни ЦАПы нуждаются в источниках как положительного, так и отрицательного напряжения, другие работают от одного источника. Одни поддерживают гибкий диапазон питающего напряжения, другие требуют строго определенного напряжения. Входные логические уровни должны быть совместимы с цифровым источником. Некоторые ЦАПы выдают на выходе напряжение, пропорциональное цифровому входу, в других выходом является уровень тока; для преобразования силы тока в напряжение в заданном диапазоне необходим операционный усилитель.

В этом разделе мы будем использовать 8-битовый параллельный ЦАП AD558 компании Analog Devices и 12-битовый последовательный ЦАП LTC1257 компании Linear Technology. Оба выдают на выходе напряжение, питаются от одного источника 5–15 В, в обоих V_{IH} = 2.4 В, т. е. они совместимы с устройствами ввода-вывода с напряжением 3.3 В. Оба выполнены в двухрядном корпусе (DIP), так что их удобно использовать с макетными платами. AD558 порождает выходное напряжение в диапазоне 0–2.56 В,

потребляет 75 мВт, выполнен в 16-контактном корпусе, время установления выходного напряжения для него равно 1 мкс, что позволяет достигать выходной частоты 1 миллион отсчетов в секунду. Технические характеристики приведены на сайте analog.com. LTC1257 порождает выходное напряжение в диапазоне 0–2.048 В, потребляет меньше 2 мВт, выполнен в 8-контактном корпусе, время установления выходного напряжения для него равно 6 мкс. Его SPI работает с максимальной частотой 1.4 МГц. Технические характеристики приведены на сайте linear.com.

Пример 4.7. АНАЛОГОВЫЙ ВЫВОД С ПРИМЕНЕНИЕМ ВНЕШНЕГО ЦАП

Нарисуйте эскиз схемы и напишите программное обеспечение простого генератора сигналов синусоидальной и треугольной формы с использованием Raspberry Pi, AD558 и LTC1257

Решение. Схема показана на **Рис. 4.19**. AD558 подключается к Pi с помощью контактов GPIO14, 15, 17, 18, 22, 23, 24 и 25. Он соединяет выводы *Vout Sense* и *Vout Select* с *Vout*, чтобы покрыть весь выходной диапазон 2.56 В. LTC1257 подключается к Pi через SPI0. Оба ЦАПа запитываются от источника 5 В и шунтируются развязывающим конденсатором емкостью 0.1 мкФ для снижения наводки от блока питания. Низкие активные уровни сигнала готовности ЦАП и сигнала загрузки на ЦАП запускают преобразование входного цифрового значения. Во время загрузки нового значения в ЦАП оба сигнала должны иметь высокий уровень.

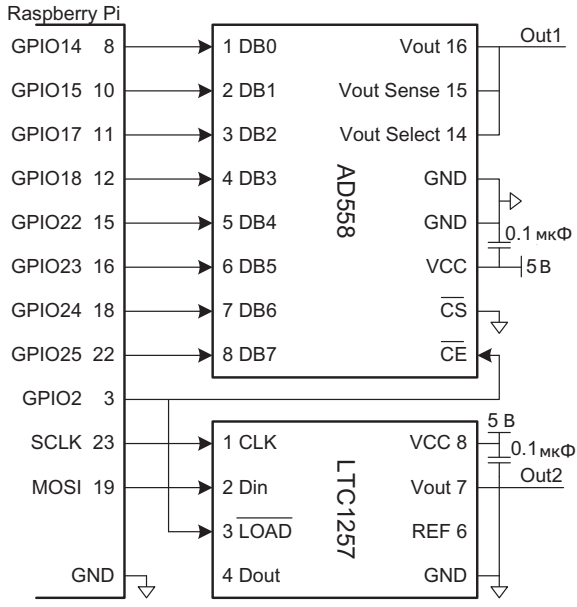


Рис. 4.19. Подключение ЦАП к Raspberry Pi при помощи параллельного и последовательного интерфейсов

Программа приведена ниже. Функции `pinsMode` и `digitalWrites` похожи на `pinMode` и `digitalWrite`, но работают с массивом контактов. Программа назначает все 8 контактов параллельного порта выводами и конфигурирует GPIO2 как вывод для формирования сигналов разрешения микросхемы и нагрузки. SPI инициализируется для работы на частоте 1.4 МГц. Функция `initWaveTables` заранее вычисляет массив отсчетов для синусоидальной и треугольной формы сигнала. Синусоидальная форма вычисляется для 12-разрядной шкалы, треугольная – для 8-разрядной. Для каждой формы обсчитываются 64 точки в периоде; варьирование этой величины означает компромисс между точностью и частотой. Функция `genWaves` организует цикл по отсчетам. Она запрещает прерывания, чтобы подавить переключение процессов и возможное из-за этого искажение формы сигнала. Для каждого отсчета запрещаются сигналы CE и LOAD, адресованные ЦАП, передается новый отсчет в последовательный и параллельный порты, разрешаются сигналы ЦАП, а затем функция ждет, когда сработает таймер и можно будет загружать следующий отсчет. Функция `spiSendReceive16` передает два байта, но LTC1257 обрабатывает только последние 12 бит. При вызове функции `genWaves`, которая передает точки, задается максимальная частота, немного превышающая 1000 Гц (64 тысячи отсчетов/с); основным фактором, определяющим частоту, является передача по интерфейсу SPI.

```
#include "EasyPIO.h"
#include <math.h> // необходимо для использования функции sin

#define NUMPTS 64
int sine[NUMPTS], triangle[NUMPTS];
int parallelPins[8] = {14,15,17,18,22,23,24,25};

void initWaveTables(void) {
     int i;
     for (i = 0; i<NUMPTS; i ++) {
          sine[i] = 2047*(sin(2*3.14159*i/NUMPTS) + 1); // 12-разрядная шкала
          if (i<NUMPTS/2) triangle[i] = i*511/NUMPTS; // 8-разрядная шкала
          else triangle[i] = 510-i*511/NUMPTS;
        }
}

void genWaves(int freq) {
     int i, j;
     int microPeriod = 1000000/(NUMPTS*freq);

     noInterrupts(); // запретить прерывания для точного хронометража
     for (i = 0; i<2000; i ++ ){
          for (j = 0; j<NUMPTS; j ++) {
               SYS_TIMER_C1 = SYS_TIMER_CLO + microPeriod; // Задать время
                                                           // между отсчетами
               SYS_TIMER_CSbits.M1 = 1; // Очистить бит совпадения с таймером
               digitalWrite(2,1);       // Запретить загрузку во время
                                        // изменения входа
               spiSendReceive16(sine[j]);
               digitalWrites(parallelPins, 8, triangle[j]);
               digitalWrite(2,0);            // Загрузить новые точки в ЦАПы
               while (!SYS_TIMER_CSbits.M1); // Ждать совпадения с таймером
          }
```

```
    }
    interrupts();
}

void main(void) {
    pioInit();

    pinsMode(parallelPins, 8, OUTPUT); // Назначить контакты, подключенные
                                       // в AD558, выводами
    pinMode(2, OUTPUT); // Назначить контакт 2 выводом для управления LOAD и CE
    spiInit(1400000, 0); // Частота SPI 1.4 МГц, параметры по умолчанию
    initWaveTables();
    genWaves(1000);
}
```

Широтно-импульсная модуляция

Другим способом генерировать аналоговый выходной сигнал в цифровой системе является *широтно-импульсная модуляция* (ШИМ), при которой периодический выходной сигнал принимает высокое значение в течение части периода передачи и низкое для оставшейся части. Доля периода, для которой сигнал имеет высокое значение, называется *коэффициентом заполнения*, или *скважностью* (см. **Рис. 4.20**).

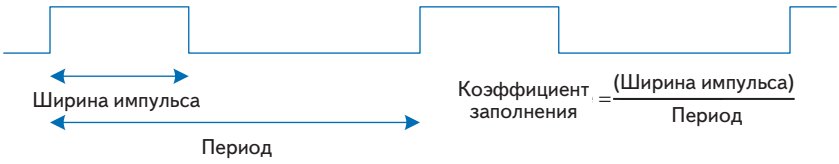


Рис. 4.20. Сигнал, полученный методом широтно-импульсной модуляции

Среднее значение на выходе пропорционально коэффициенту заполнения. Например, если выходное напряжение изменяется от 0 до 3.3 В и коэффициент заполнения составляет 25%, то среднее значение будет 0.25 × 3.3 = 0.825 В. Низкочастотная фильтрация сигнала ШИМ исключает колебания, и выходной сигнал принимает требуемое среднее значение. Таким образом, ШИМ является эффективным способом формирования аналогового выходного сигнала, если частота следования импульсов гораздо выше, чем представляющая интерес частота аналогового сигнала.

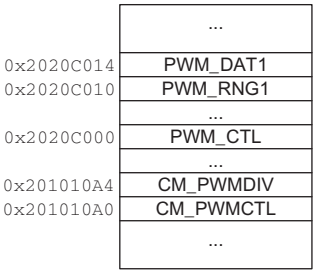


Рис. 4.21. Регистры ШИМ и блока управления синхронизацией

В BCM2835 имеется контроллер ШИМ (PWM0), способный одновременно генерировать два выходных сигнала. PWM0 доступен через контакт GPIO18 как функция ALT5, и оба выходных сигнала ШИМ доступны через стереофо-

нический аудиоразъем. На **Рис. 4.21** показано, как отображаются на память блок ШИМ и блок управления синхронизацией, от которого он зависит.

Регистр PWM_CTL служит для включения широтно-импульсной модуляции. Чтобы разрешить формирование выходного сигнала, бит 0 (PWEN1) должен быть установлен. Для разрешения широтно-импульсной модуляции вида, показанного на **Рис. 4.20**, где выходной сигнал имеет высокий уровень в течение одной части периода и низкий в течение другой части, должен быть также установлен бит 7 (MSEN1: mark-space enable).

Сигналы ШИМ формируются с частотой ШИМ, генерируемой блоком управления синхронизацией BCM2835. Регистры PWM_RNG1 и PWM_DAT1 управляют периодом и коэффициентом заполнения соответственно, в них задается число тактов ШИМ на протяжении всего сигнала и той его части, где уровень высокий. Например, если блок управления синхронизацией генерирует частоту 25 МГц и PWM_RNG1 = 1000, PWM_DAT1 = 300, то на выходе ШИМ будет сигнал частотой (25 МГц / 1000) = 25 КГц, а коэффициент заполнения равен 300 / 1000 = 30%.

Блок управления синхронизацией конфигурируется с помощью регистра CM_PWMCTL, а частота задается в регистре CM_PWMDIV. В **Табл. 4.7** перечислены битовые поля регистра CM_PWMCTL. Максимальная частота тактового генератора ШИМ равна 25 МГц. Ее можно получить от 500-мегагерцового генератора с фазовой автоподстройкой частоты (PLLD), имеющегося на плате Pi, следующим образом:

Регистры CM_PWM не описаны в технической документации по BCM2835. Информацию о них можно получить, поискав в Интернете статью G.J. van Loo «BCM2835 Audio & PWM».

- CM_PWMCTL: записать 0x5A в PASSWD и 1 в KILL, чтобы остановить тактовый генератор;
- CM_PWMCLT: ждать, пока очистится бит BUSY, это будет означать, что генератор остановлен;
- CM_PWMCTL: записать 0x5A в PASSWD, 1 в MASH и 6 в SRC, чтобы выбрать PLLD без модуляции акустического шума;
- CM_PWMDIV: записать 0x5A в PASSWD и 20 в биты 23:12, чтобы поделить частоту PLLD на 20, т. е. понизить с 500 до 25 МГц;
- CM_PWMCTL: записать 0x5A в PASSWD и 1 в ENAB, чтобы снова запустить тактовый генератор;
- CM_PWMCTL: ждать, пока установится бит BUSY, это будет означать, что генератор работает.

Таблица 4.7. Поля регистра CM_PWMCTL

Биты	Имя	Описание
31:24	PASSWD	Должны содержать 5A при записи
10:9	MASH	Модуляция акустического шума
7	BUSY	Тактовый генератор работает
5	KILL	Записать 1, чтобы остановить тактовый генератор
4	ENAB	Записать 1, чтобы запустить тактовый генератор
3:0	SRC	Источник тактовых сигналов

Пример 4.8. АНАЛОГОВЫЙ ВЫВОД С ИСПОЛЬЗОВАНИЕМ ШИМ

Напишите функцию `analogWrite(val)` для генерации аналогового вывода в виде создания некоторого уровня напряжения с использованием широтно-импульсной модуляции (ШИМ) и внешнего RC-фильтра. Аргументом функции является целое число в диапазоне между 0 (напряжение на выходе 0 В) и 255 (максимальное напряжение на выходе 3.3 В).

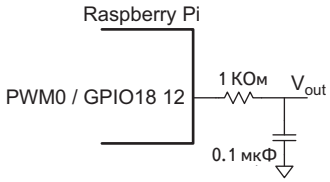


Рис. 4.22. Формирование аналогового выходного сигнала с применением ШИМ и фильтра нижних частот

Решение. Воспользуемся модулем PWM0 для генерации сигнала с частотой 78.125 КГц на выходном контакте GPIO18. Фильтр нижних частот на **Рис. 4.22** имеет угловую частоту

$$f_c = \frac{1}{2\pi RC} = 1.6 \text{ КГц}.$$

Фильтр подавляет высокочастотные колебания и сглаживает напряжение на выходе.

Ниже приведены функции, относящиеся к ШИМ, в файле EasyPIO. Функция `pwmInit` конфигурирует контакт GPIO18 для работы с модулем ШИМ, как описано выше. Функция `setPWM` задает частоту и коэффициент заполнения выходного сигнала ШИМ. Коэффициент заполнения должен принимать значение от 0 (всегда OFF) до 1 (всегда ON). Функция `analogWrite` задает коэффициент заполнения относительно максимального значения 255.

```
// Значение PLLD по умолчанию равно 500 [МГц]
#define PLL_FREQUENCY 500000000
// Максимальная тактовая частота ШИМ равна 25 [МГц]
#define CM_FREQUENCY 25000000
#define PLL_CLOCK_DIVISOR (PLL_FREQUENCY / CM_FREQUENCY)

void pwmInit() {
     pinMode(18, ALT5);

     // Сконфигурировать блок управления синхронизацией, так чтобы он
     // генерировал частоту ШИМ 25 Мгц. Документация по блоку управления
     // синхронизацией отсутствует в официальной спецификации, но ее можно
```

```
    // найти в статье G.J. van Loo "BCM2835 Audio and PWM Clocks"
    // от 6 февраля 2013 г.
    // Максимальная рабочая частота тактового генератора ШИМ равна 25 МГц.
    // При записи в регистры блока управления синхронизацией нужно
    // одновременно записывать "пароль", равный 5A, в старшие биты,
    // чтобы уменьшить риск случайной записи.
    CM_PWMCTL = 0; // Выключить ШИМ перед изменением
    CM_PWMCTL = PWM_CLK_PASSWORD|0x20; // Выключить тактовый генератор
    while (CM_PWMCTLbits.BUSY); // Ждать, пока генератор остановится
    CM_PWMCTL = PWM_CLK_PASSWORD|0x206; // Источник = CLKD 500 МГц
                                        // без фильтрации
    CM_PWMDIV = PWM_CLK_PASSWORD|(PLL_CLOCK_DIVISOR << 12); // 25 МГц
    CM_PWMCTL = CM_PWMCTL|PWM_CLK_PASSWORD|0x10; // Разрешить синхронизацию PWM
    while (!CM_PWMCTLbits.BUSY); // Ждать, пока генератор запустится
    PWM_CTLbits.MSEN1 = 1; // Канал 1 в режиме посылки-паузы
    PWM_CTLbits.PWEN1 = 1; // Включить ШИМ
}

void setPWM(float freq, float duty) {
    PWM_RNG1 = (int)(CM_FREQUENCY / freq);
    PWM_DAT1 = (int)(duty * (CM_FREQUENCY / freq));
}

void analogWrite(int val) {
    setPWM(78125, val/255.0);
}
```

Функция `main` тестирует ШИМ, выставляя половинное напряжение (1.65 В).

```
#include "EasyPIO.h"


void main(void) {
    pioInit();
    pwmInit();
    analogWrite(128);
}
```

Аналого-цифровое преобразование

В BCM2835 нет встроенного АЦП, поэтому в этом разделе описывается аналого-цифровое преобразование с помощью внешнего преобразователя, похожего на внешний ЦАП.

Пример 4.9. АНАЛОГОВЫЙ ВВОД С ВНЕШНИМ АЦП

Реализуйте интерфейс 10-битового АЦП MCP3002 с Raspberry Pi с применением SPI и напечатайте полученное на входе значение. Максимальное напряжение должно быть равно 3.3 В. Детальное описание работы устройства найдите в Интернете.

Решение. На **Рис. 4.23** показана схема подключения. Устройство MCP3002 использует сигнал VDD в качестве полного опорного уровня. Оно запитывается

от источника 3.3–5.5 В, мы выбрали напряжение 3.3 В. АЦП имеет два канала, мы подключили канал 0 к потенциометру, чтобы, вращая его ручку, можно было регулировать входное напряжение от 0 до 3.3 В.

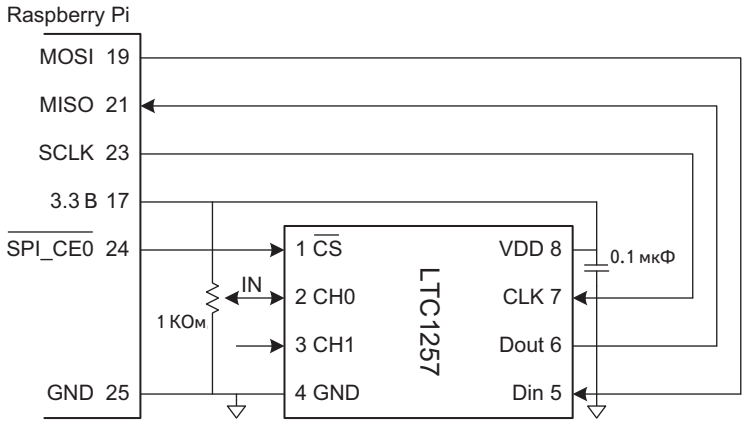


Рис. 4.23. Преобразование аналогового входа с помощью внешнего АЦП

Программа для Pi инициализирует SPI и в цикле читает и печатает отсчеты. Согласно технической документации, Raspberry Pi должна послать 16-битовое значение 0x6000 через SPI, чтобы прочитать CH0, а в ответ получит результат в 10 младших битах 16-битового числа. Преобразователю нужен также сигнал выбора микросхемы, который легко получить с вывода CE интерфейса SPI.

```
#include "EasyPIO.h"

void main(void) {
     int sample;

     pioInit();
     spiInit(200000, 0); // тактовая частота SPI 200 КГц, параметры по умолчанию

     while (1){
         sample = spiSendReceive16(0x6000);
         printf("Read %d\n", sample);
     }
}
```

4.3.7. Прерывания

До сих пор мы полагались на технику *опроса*, когда программа постоянно проверяет, произошло ли некоторое событие, например поступление данных в UART или достижение таймером заданного значения. Это напрасная трата ресурсов процессора, к тому же при таком подходе трудно писать программы, которые выполняют содержательную работу, пока событие не наступило.

Большинство микроконтроллеров поддерживает *прерывания*. Когда происходит событие, микроконтроллер может приостановить выполнение программы и перейти к обработчику прерывания, который отреагирует на прерывание, а затем вернется в то место, откуда был вызван, как будто ничего не случилось.

Raspberry Pi обычно работает под управлением ОС Linux, которая перехватывает прерывания до того, как программа их увидит. Поэтому в настоящее время писать управляемые прерываниями программы не так-то просто, и в этой книге мы не станем приводить соответствующие примеры.

4.4. Другие периферийные устройства микроконтроллеров

Микроконтроллеры часто взаимодействуют с другими внешними периферийными устройствами. В этом разделе рассматривается несколько примеров таких устройств, в т. ч. символьные жидкокристаллические дисплеи (LCD), VGA-мониторы, беспроводные каналы связи Bluetooth и управление двигателем. Стандартные интерфейсы связи, включая USB и Ethernet, описаны в **разделах 4.6.1** и **4.6.4**.

4.4.1. Символьный ЖК-дисплей

Символьный ЖК-дисплей – небольшой жидкокристаллический дисплей, способный показывать одну или несколько строк текста. Они широко используются в передних панелях приборов, таких как кассовые аппараты, лазерные принтеры и факсы, которые должны отображать лишь ограниченное количество информации. Они легко подключаются к микроконтроллеру через параллельный интерфейс, RS-232 или SPI. Компания Crystalfontz America продает широкий спектр символьных ЖК-дисплеев, начиная от 8 столбцов × 1 строку до 40 столбцов × 4 строки с выбором цвета, подсветки, питанием 3.3 или 5 В и видимостью при дневном свете. Эти ЖК-дисплеи могут стоить 20 и более долларов в розницу и менее 5 долларов оптом.

В этом разделе приведен пример взаимодействия Raspberry Pi с символьным ЖК-дисплеем по 8-битовому параллельному интерфейсу. Интерфейс совместим с LCD-контроллером HD44780, который изначально был разработан компанией Hitachi, а теперь является промышленным стандартом. На **Рис. 4.24** показан параллельный ЖК-дисплей 20×2 CFAH2002A-TMI-JT компании Crystalfontz.

На **Рис. 4.25** показан ЖК-дисплей, подключенный к Pi через параллельный 8-битовый интерфейс. Его логические схемы работают от напряжения 5 В, но совместимы с входными сигналами уровня 3.3 В от

Pi. Контрастность ЖК-дисплея устанавливается вторым напряжением, выдаваемым потенциометром; обычно экран лучше всего читается, если напряжение принадлежит диапазону 4.2–4.8 В. ЖК-дисплей получает три управляющих сигнала: RS (1 – для символов, 0 – для команд), R/$\overline{W}$ (1 – читать с дисплея, 0 – записывать) и E (поднимаемый до высокого уровня как минимум на 250 нс, для того чтобы включить ЖК-дисплей, когда следующий байт будет готов). При чтении команды бит 7 содержит флаг занятости, равный 1, если ЖК-дисплей занят, и 0, если он готов принять следующую команду.

Рис. 4.24. Символьный ЖК-дисплей CFAH2002A-TMI 20×2 компании Crystalfontz (© 2012 Crystalfontz America, печатается с разрешения)

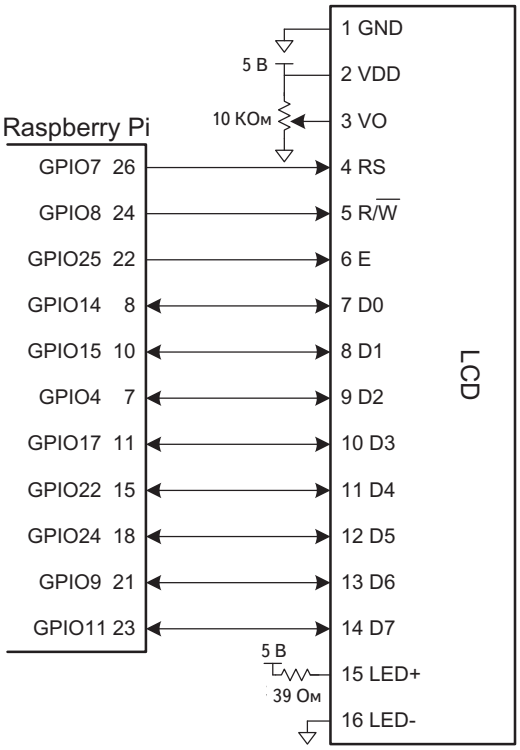


Рис. 4.25. Подключение ЖК-дисплея при помощи параллельного интерфейса

Для инициализации ЖК-дисплея Pi должна записать последовательность команд, показанную в **Табл. 4.8**. Эти команды записываются при RS = 0 и R/$\overline{W}$ = 0: значение помещается на восемь линий данных, после чего подается импульс на линию E. После каждой команды нужно подождать как минимум указанное время (а иногда до очистки флага занятости).

Таблица 4.8. Последовательность инициализации ЖК-дисплея

Записать	Назначение	Ждать (мкс)
(подать напряжение на V_{DD})	Включить устройство	15 000
0x30	Установить 8-битовый режим	4100
0x30	Еще раз установить 8-битовый режим	100
0x30	И еще раз установить 8-битовый режим	Пока не очистится флаг занятости
0x3C	Установить 2 строки и шрифт 5×8	Пока не очистится флаг занятости
0x08	Выключить дисплей	Пока не очистится флаг занятости
0x01	Очистить дисплей	1530
0x06	Установить режим ввода со сдвигом курсора после каждого символа	Пока не очистится флаг занятости
0x0C	Включить дисплей без курсора	

Затем, чтобы вывести текст на ЖК-дисплей, Pi может отправить последовательность знаков в кодировке ASCII. После отправки каждого знака необходимо дождаться очистки бита занятости. Pi может также отправить команду 0x01, чтобы очистить дисплей, или 0x02, чтобы вернуться в начальную позицию в левом верхнем углу.

Пример 4.10. УПРАВЛЕНИЕ ЖК-ДИСПЛЕЕМ

Напишите программу для вывода строки «I love LCDs» на символьный ЖК-дисплей.

Решение. Приведенная ниже программа выводит строку «I love LCDs» на ЖК-дисплей. Сначала она инициализирует дисплей, а затем отправляет знаки.

```
#include "EasyPIO.h"

int LCD_IO_Pins[] = {14, 15, 4, 17, 22, 24, 9, 11};

typedef enum {INSTR, DATA} mode;
```

```
#define RS 7
#define RW 8
#define E 25

char lcdRead(mode md) {
     char c;
     pinsMode(LCD_IO_Pins, 8, INPUT);
     digitalWrite(RS,(md == DATA)); // Установить режим команд/данных
     digitalWrite(RW, 1);           // Режим чтения
     digitalWrite(E, 1);            // Включить
     delayMicros(10);               // Ждать ответа от ЖК-дисплея
     c = digitalReads(LCD_IO_Pins, 8); // Читать байт из параллельного порта
     digitalWrite(E, 0);               // Выключить
     delayMicros(10);
     return c;
}

void lcdBusyWait(void) {
     char state;
     do {
          state = lcdRead(INSTR);
     } while (state & 0x80);
}

void lcdWrite(char val, mode md) {
     pinsMode(LCD_IO_Pins, 8, OUTPUT);
     digitalWrite(RS, (md == DATA)); // Установить режим команд/данных.
                                     // OUTPUT = 1, INPUT = 0
     digitalWrite(RW, 0); // Установить контакт RW в режим записи (т. е. 0)
     digitalWrites(LCD_IO_Pins, 8, val); // Писать символ в параллельный порт
     digitalWrite(E, 1); delayMicros(10); // Импульс на линию E
     digitalWrite(E, 0); delayMicros(10);
}

void lcdClear(void) {
     lcdWrite(0x01, INSTR); delayMicros(1530);
}

void lcdPrintString(char* str) {
     while (*str != 0) {
          lcdWrite(*str, DATA); lcdBusyWait();
          str++;
     }
}

void lcdInit(void) {
     pinMode(RS, OUTPUT); pinMode(RW, OUTPUT); pinMode(E,OUTPUT);
     // процедура инициализации перед отправкой данных
     delayMicros(15000);
     lcdWrite(0x30, INSTR); delayMicros(4100);
     lcdWrite(0x30, INSTR); delayMicros(100);
     lcdWrite(0x30, INSTR); lcdBusyWait();
     lcdWrite(0x3C, INSTR); lcdBusyWait();
     lcdWrite(0x08, INSTR); lcdBusyWait();
     lcdClear();
```

```
    lcdWrite(0x06, INSTR); lcdBusyWait();
    lcdWrite(0x0C, INSTR); lcdBusyWait();
}

void main(void) {
    pioInit();
    lcdInit();
    lcdPrintString("I love LCDs!");
}
```

4.4.2. VGA-монитор

Более гибкий вариант дисплея – монитор компьютера. В Raspberry Pi встроена поддержка HDMI и композитного видеовыхода. В этом разделе описываются низкоуровневые детали управления VGA-монитором непосредственно из ПЛИС .

Стандарт *Video Graphics Array* (VGA) был принят в 1987 году для компьютеров IBM PS/2, имевших монитор на электронно-лучевой трубке (ЭЛТ) с разрешением 640×480 пикселей и 15-контактным разъемом, который передавал цветовую информацию с помощью аналоговых напряжений. Современные ЖК-мониторы имеют более высокое разрешение, но сохраняют обратную совместимость со стандартом VGA.

В электронно-лучевой трубке электронная пушка сканирует экран слева направо, активируя флуоресцентный материал для формирования изображения. В цветных ЭЛТ-экранах используются три типа люминофоров для красного, зеленого и синего цветов и три электронных луча. Мощность луча определяет интенсивность цвета в пикселе. В конце каждой строки развертки пушка должна выключиться на время, равное *горизонтальному интервалу гашения*, чтобы вернуться к началу следующей строки. После того как все строки развертки пройдены, пушку необходимо выключить снова на время *вертикального интервала гашения* (обратного хода луча), необходимое для возврата в левый верхний угол. Процесс повторяется 60–75 раз в секунду, чтобы создать визуальную иллюзию устойчивого изображения. В ЖК-дисплее электронной пушки нет, но ради совместимости с VGA используется такой же интерфейс.

В VGA-мониторе с разрешением 640×480 пикселей с частотой обновления кадра 59.94 Гц попиксельное обновление выполняется с частотой 25.175 МГц, так что на обновление каждого пикселя приходится 39.72 нс. Полный экран можно рассматривать как 525 горизонтальных строк развертки по 800 пикселей каждая, но только 480 из этих строк и 640 пикселей в каждой строке фактически передают изображение, в то время как остальные остаются черными. Сканирование строки начинается со строчного гасящего импульса (СГИ), так называемого «*заднего крыльца*» (back porch) – пустой секции в левой части экрана. Затем стро-

ка содержит 640 пикселей, после чего идет *кадровый гасящий импульс* (КГИ), так называемое «*переднее крыльцо*» (front porch) в правой части экрана, и импульс горизонтальной синхронизации (*Hsync*), который быстро перемещает луч к левому краю.

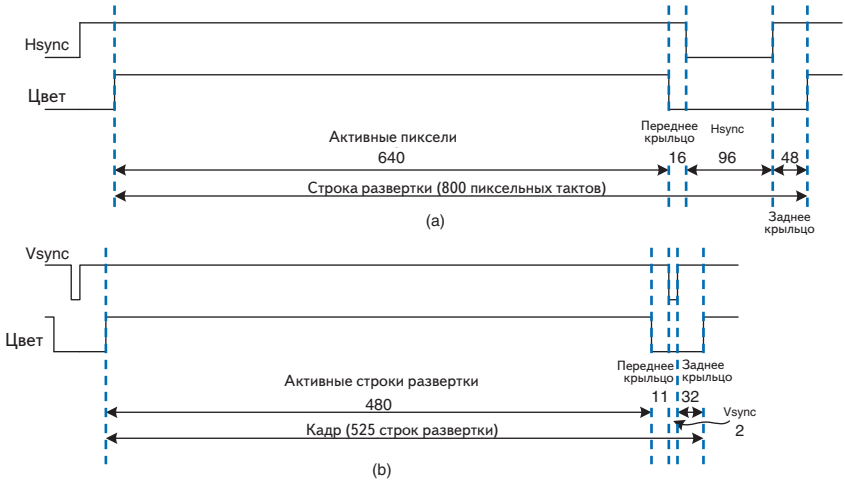
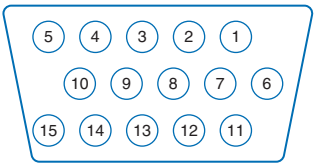


Рис. 4.26. Временные диаграммы VGA: (a) горизонтальная, (b) вертикальная



1: Красный
2: Зеленый
3: Синий
4: Зарезервировано
5: GND
6: GND
7: GND
8: GND
9: 5 В (факультативно)
10: GND
11: Зарезервировано
12: Данные I²C
13: Hsync
14: Vsync
16: Синхронизация I²C

Рис. 4.27. Разводка разъема VGA

На **Рис. 4.26 (a)** показаны временные диаграммы всех участков строки развертки, начиная с активных пикселей. Развертка одной строки занимает 31.778 мкс. В вертикальном направлении экран начинается пустой областью в верхней части, за которой следует 480 активных строк развертки, пустая область и импульс вертикальной синхронизации (*Vsync*) для возврата в начало и развертки следующего кадра. В секунду отображается 60 кадров.

На **Рис. 4.26 (b)** показана временная диаграмма вертикальной развертки; заметим, что единицей времени теперь является строка развертки, а не время формирования пикселя. Чем выше разрешение, тем быстрее должен формироваться пиксель; при частоте кадров 85 Гц и разрешении 2048×1536 частота следования пикселей должна быть равна 388 МГц при 2048×1536 при 85 Гц. При разрешении 1024×768 и частоте кадров 60 Гц достаточно частоты следования пикселей 65 МГц.

Горизонтальная временная диаграмма включает КГИ длительностью 16 тактов, сигнал *Hsync* длительностью 96 и СГИ длительностью 48 тактов. Вертикальная временная диаграмма включает КГИ, занимающий 11 строк развертки, сигнал *Vsync*, занимающий 2 строки, и СГИ, занимающий 32 строки.

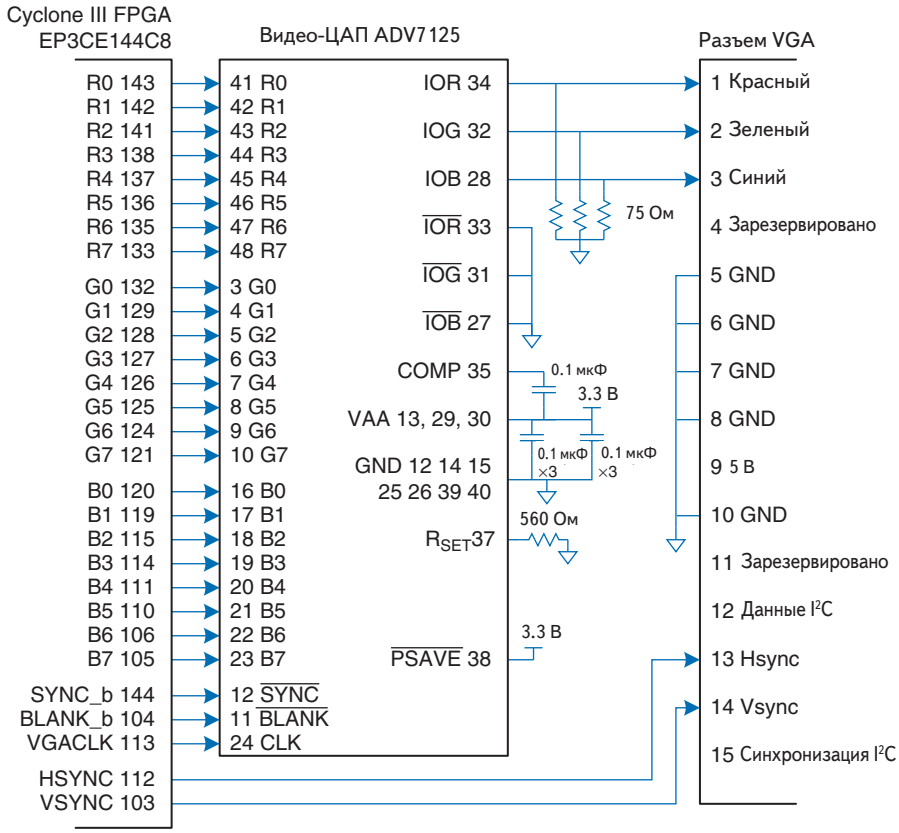


Рис. 4.28. Кабель VGA, управляемый ПЛИС через ЦАП

На **Рис. 4.27** представлена разводка разъема-розетки, являющегося частью источника видеосигнала. Информация о пикселе передается тремя аналоговыми напряжениями, соответствующими красному, зеленому и синему цветам. Каждое напряжение изменяется от 0 до 0.7 В: чем выше напряжение, тем ярче пиксель. Напряжения должны быть равны 0 во время КГИ и СГИ. Видеосигнал должен генерироваться в режиме реального времени с высокой скоростью. Этого трудно достичь с помощью микроконтроллера, но легко – с помощью ПЛИС (FPGA). Простой черно-белый дисплей может быть реализован подачей на все три цветовых контакта напряжения 0 или 0.7 В при помощи делителя напряжения, подключенного к цифровому выходному контакту. С другой стороны, в цветном мониторе используется видео-ЦАП с тремя отдельными ЦАП для независимого управления цветовыми контактами. На **Рис. 4.28** показано, как ПЛИС управляет VGA-монитором через тройной 8-битовый видео-ЦАП ADV7125. ЦАП получает 8 бит R, G и B от ПЛИС, а также

сигнал SYNC_b, который устанавливается в низкий активный уровень, как только активированы сигналы HSYNC или VSYNC. Видео-ЦАП генерирует три выходных уровня тока для управления красной, зеленой и синей аналоговыми линиями, которые обычно являются 75-омными параллельными линиями передачи, подключенными к видео-ЦАП и монитору. Резистор R_{SET} устанавливает величину выходного тока для достижения полного спектра цвета. Тактовая частота зависит от разрешения и частоты регенерации. Она может достигать значения 330 МГц в быстродействующем ЦАП модели ADV7125JSTZ330.

Пример 4.11. ВЫВОД НА VGA-МОНИТОР

Напишите HDL-код для вывода текста и зеленого прямоугольника на VGA-монитор, используя схему, показанную на **Рис. 4.28**.

Решение. Мы предполагаем, что системная частота синхронизации равна 40 МГц и что в ПЛИС используется *фазовая автоподстройка частоты* (ФАПЧ) для генерации синхросигнала VGA частотой 25.175 МГц. Настройка ФАПЧ может быть разной в различных ПЛИС; для Cyclone III частоты, генерируемые ФАПЧ, настраиваются с помощью специального мастера установки от Altera. Альтернативно синхросигнал VGA может поступать от внешнего генератора сигналов.

Контроллер VGA отсчитывает строки и столбцы на экране, генерируя сигналы HSYNC и VSYNC в нужные моменты времени. Он также генерирует сигнал BLANK_b, низкий уровень которого служит для закрашивания пространства экрана за пределами рабочей области 640×480 черным цветом.

Видеогенератор порождает значения красного, зеленого и синего цветов для каждого пикселя на экране, адресуемого координатами (x, y). Начало координат $(0, 0)$ расположено в левом верхнем углу экрана. Генератор выводит на экран набор символов и зеленый прямоугольник. Изображения символов имеют размер 8×8 пикселей. Таким образом, рабочая область экрана вмещает 80×60 символов. Генератор символов хранит изображения символов в своей постоянной памяти (ROM). Каждый символ представляется в виде таблицы размером 8 строк на 6 столбцов, содержащий 0 и 1. Оставшиеся две колонки пустые, т. е. заполнены нулями. Порядок бит инвертируется кодом на языке SystemVerilog, т. к. самая левая колонка в ROM-файле соответствует старшему биту, который должен отображаться в позиции с наименьшей координатой x.

На **Рис. 4.29** приведена фотография экрана VGA-монитора во время выполнения этой программы. Цвет строк, содержащих буквы, чередуется: одна строка красная, другая – синяя. Зеленый прямоугольник накладывается на изображение на экране.

vga.sv

```
module vga(input  logic       clk,
           output logic       vgaclk,  // тактовая частота VGA 25.175 МГц
           output logic       hsync, vsync,
           output logic       sync_b, blank_b, // К монитору и ЦАП
           output logic [7:0] r, g, b);        // К видео-ЦАП

  logic [9:0] x, y;
```

```
  // Использовать ФАПЧ[4] для генерации частоты пикселей VGA 25.175 МГц.
  // Частоте 25.175 МГц соответствует период 39.772 нс
  // Ширина экрана - 800 тактов, высота - 525 тактов, но используется только 640×480
  // HSync = 1/(39.772 нс *800) = 31.470 КГц
  // Vsync = 31.474 КГц / 525 = 59.94 Гц (частота регенерации ~60 Гц)
  pll vgapll(.inclk0(clk), .c0(vgaclk));

  // Генерировать сигналы синхронизации монитора
  vgaController vgaCont(vgaclk, hsync, vsync, sync_b, blank_b, x, y);

  // Пользовательский модуль для определения цветов пикселей
  videoGen videoGen(x, y, r, g, b);
endmodule

module vgaController #(parameter HACTIVE = 10'd640,
                                 HFP     = 10'd16,
                                 HSYN    = 10'd96,
                                 HBP     = 10'd48,
                                 HMAX    = HACTIVE + HFP + HSYN + HBP,
                                 VBP     = 10'd32,
                                 VACTIVE = 10'd480,
                                 VFP     = 10'd11,
                                 VSYN    = 10'd2,
                                 VMAX    = VACTIVE + VFP + VSYN + VBP)
                       (input  logic       vgaclk,
                        output logic       hsync, vsync, sync_b, blank_b,
                        output logic [9:0] x, y);
  // счетчики горизонтальной и вертикальной позиций
  always @(posedge vgaclk) begin
    x++;
    if (x == HMAX) begin
      x = 0;
      y++;
      if (y == VMAX) y = 0;
    end
  end

  // Вычислить сигналы синхронизации (активный уровень - низкий)
  assign hsync = ~(x >= HACTIVE + HFP &  x < HACTIVE + HFP + HSYN);
  assign vsync = ~(y >= VACTIVE + VFP & y < VACTIVE + VFP + VSYN);
  assign sync_b = hsync & vsync;

  // Вне рабочей области экрана должен выводиться черный цвет
  assign blank_b = (x < HACTIVE) & (y < VACTIVE);
endmodule

module videoGen(inputlogic [9:0] x, y, output logic [7:0] r, g, b);
  logic pixel, inrect;

  // Зная координату y, решить, какой символ отображать,
  // затем найти значение пикселя в ПЗУ генератора символов
```

[4] ФАПЧ по-английски – PLL (Phase-Locked Loop), и поэтому в данном примере соответствующий модуль называется pll. Сам же модуль pll, как было сказано выше, можно автоматически сгенерировать мастером установки от Altera либо каким-то другим способом, предусмотренным изготовителем вашей ПЛИС. – *Прим. перев.*

```
  // и нарисовать его красным или синим цветом. Также нарисовать
  // зеленый прямоугольник.
  chargenrom chargenromb(y[8:3] + 8'd65, x[2:0], y[2:0], pixel);
  rectgen rectgen(x, y, 10'd120, 10'd150, 10'd200, 10'd230, inrect);
  assign {r, b} = (y[3]==0) ? {{8{pixel}},8'h00} : {8'h00,{8{pixel}}};
  assign g = inrect ? 8'hFF : 8'h00;
endmodule

module chargenrom(input  logic [7:0] ch,
                  input  logic [2:0] xoff, yoff,
                  output logic       pixel);

  logic [5:0] charrom[2047:0]; // ПЗУ генератора символов
  logic [7:0] line;            // строка, прочитанная из ПЗУ

  // Инициализировать ПЗУ символами из текстового файла
  initial
    $readmemb("charrom.txt", charrom);

  // Найти в ПЗУ строку, соответствующую символу
  assign line = charrom[yoff + {ch-65, 3'b000}]; // Вычесть 65, потому что
                                                 // в позиции 0 находится символ A

  // Инвертировать порядок бит
  assign pixel = line[3'd7-xoff];
endmodule

module rectgen(input  logic [9:0] x, y, left, top, right, bot,
               output logic       inrect);
  assign inrect = (x >= left & x < right & y >= top & y < bot);
endmodule
```

charrom.txt

```
// A ASCII 65
011100
100010
100010
111110
100010
100010
100010
000000
//B ASCII 66
111100
100010
100010
111100
100010
100010
111100
000000
//C ASCII 67
011100
100010
100000
```

```
100000
100000
100010
011100
000000
...
```

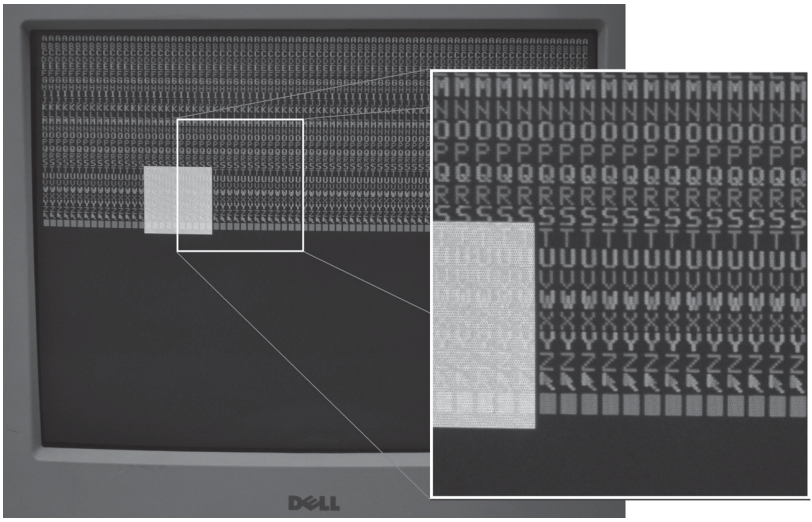


Рис. 4.29. **Вывод на экран VGA-монитора**

4.4.3. Беспроводная связь Bluetooth

В настоящее время есть много стандартов беспроводной связи, в том числе Wi-Fi, ZigBee и Bluetooth. Эти изощренные стандарты требуют сложных интегральных схем, но растущий ассортимент модулей позволяет абстрагироваться от сложности и предоставить пользователю простой интерфейс для беспроводной связи. Одним из таких модулей является BlueSMiRF, простой беспроводной интерфейс Bluetooth, который можно использовать вместо последовательного кабеля.

Bluetooth назван в честь короля Дании Харальда Синий Зуб, правившего в X веке, который объединил враждующие датские племена. Этому стандарту удалось добиться лишь частичного успеха в деле унификации множества конкурирующих протоколов беспроводной связи!

Беспроводной стандарт Bluetooth разработан компанией Ericsson в 1994 году для маломощной связи на умеренной скорости на расстоянии 5–100 метров в зависимости от мощности передатчика. Он широко используется для подключения гарнитуры к телефону или клавиатуры к компьютеру. В отличие от инфракрасных каналов связи, Bluetooth не требует прямой видимости между устройствами.

Bluetooth работает в нелицензионном диапазоне частот 2.4 ГГц для промышленного, научного и медицинской оборудования. В Bluetooth определено 79 радиоканалов с интервалом в 1 МГц, начиная с 2402 МГц. Несущая частота сигнала скачкообразно меняется между этими каналами псевдослучайным образом, чтобы избежать помех со стороны других устройств, например беспроводных маршрутизаторов, работающих в том же диапазоне. Как показано в **Табл. 4.9**, передатчики Bluetooth классифицируются по одному из трех уровней мощности, определяющих радиус действия и энергопотребление. В режиме базовой скорости передатчик работает на частоте 1 Мбит/с, используя гауссову частотную манипуляцию (ГЧМ). При обычной ЧМ каждый бит передается на частоте $f_c \pm f_d$, где f_c – центральная частота канала, а f_d – сдвиг частоты минимум на 115 кГц. Скачкообразная смена частоты при передаче бит требует дополнительного расширения полосы частот. При гауссовой ЧМ смена частоты сглаживается, чтобы спектр использовался более эффективно. На **Рис. 4.30** показано, как меняется частота при передаче последовательности нулей и единиц в канале 2402 МГц при использовании ЧМ и ГЧМ.

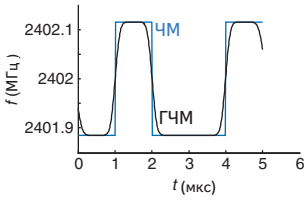


Рис. 4.30. Формы сигналов для ЧМ и ГЧМ

Таблица 4.9. Классы Bluetooth

Класс	Мощность передатчика (мВт)	Радиус действия (м)
1	100	100
2	2.5	10
3	1	5

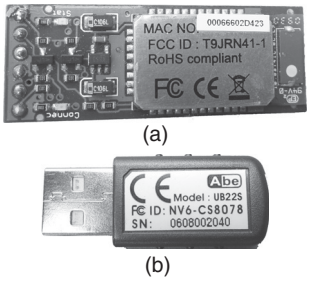


Рис. 4.31. Модуль BlueSMiRF и USB-адаптер

Модуль BlueSMiRF Silver, показанный на **Рис. 4.31 (а)**, содержит логические схемы приемопередатчика, модема и интерфейса Bluetooth класса 2. Он выполнен в виде небольшой карты с последовательным интерфейсом. Модуль связывается с другим устройством Bluetooth, например USB-адаптером Bluetooth, подключенным к ПК. Таким образом, он может обеспечить последовательный канал беспроводной связи между Pi и ПК, аналогичный каналу связи на **Рис. 4.15**, но без кабеля. Беспроводной канал совместим с тем же программным обеспечением, что и проводной.

На **Рис. 4.32** показана схема такого соединения. Контакт TX модуля BlueSMiRF подключается к контакту RX на плате Pi, и наоборот. Контакты BlueSMiRF RTS и CTS соединены друг с другом.

BlueSMiRF по умолчанию работает со скоростью 115.2K бод с 8 бит данных, 1 стоповым битом, без контроля четности и управления потоком. Рабочее напряжение составляет 3.3 В, поэтому для соединения с другим устройством с таким же уровнем напряжения приемопередатчик RS-232 не нужен.

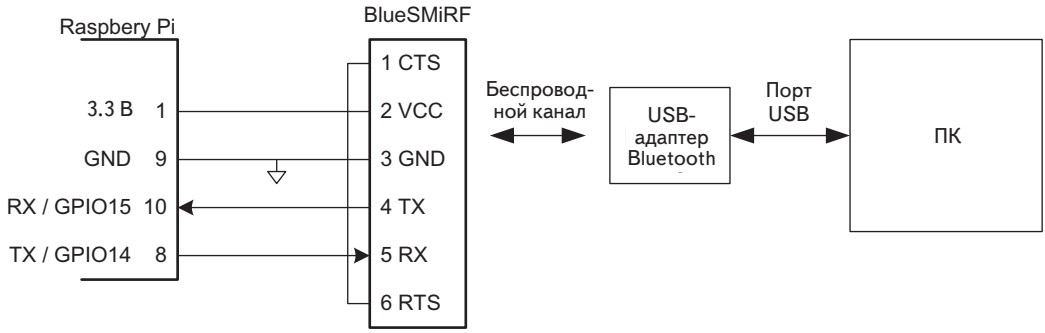


Рис. 4.32. Схема соединения между Raspberry Pi и ПК с применением BlueSMiRF

Чтобы использовать этот интерфейс, подключите USB-адаптер Bluetooth к ПК. Включите Pi и BlueSMiRF в сеть. На BlueSMiRF начнет мигать красная лампочка STAT, показывающая, что устройство ожидает подключения. Щелкните по значку Bluetooth в области уведомлений ПК и откройте мастер **Добавление устройства Bluetooth**, чтобы выполнить сопряжение адаптера с BlueSMiRF. По умолчанию ключ доступа для BlueSMiRF равен 1234. Обратите внимание, какой COM-порт назначен адаптеру. После сопряжения обмен данными будет работать так, будто устройства соединены последовательным кабелем. Обычно адаптер работает со скоростью 9600 бод, и PuTTY должна быть настроена соответствующим образом.

4.4.4. Управление двигателями

Еще одно важное применение микроконтроллеры находят для управления приводами, в т. ч. двигателями. В этом разделе описываются три типа двигателей: двигатели постоянного тока, серводвигатели и шаговые двигатели. *Двигатели постоянного тока* нуждаются в высоком токе возбуждения, поэтому между микроконтроллером и двигателем следует включить мощный драйвер, например *H-мост*. Необходим также *отдельный датчик угла поворота*, если пользователь хочет знать текущее положение двигателя. *Серводвигатель* принимает ШИМ-сигнал, задающий положение в ограниченном диапазоне углов. С такими двигателями очень просто взаимодействовать, но они менее мощные и не подходят для продолжительного непрерывного вращения. *Шаговые двигатели* принимают последовательность импульсов, каждый из которых вращает двигатель на фиксированный угол, называемый шагом. Они стоят дороже, и им также нужен H-мост для управления большим током, но положение подвижной части двигателя можно точно контролировать.

Двигатель может потреблять значительный ток, что способно привести к сбоям источника питания и, как следствие, к нарушению логики

работы цифровых схем. Один из способов смягчения этой проблемы заключается в использовании одного источника питания или батареи для питания двигателя и другого – для цифровых схем.

Электродвигатели постоянного тока

На **Рис. 4.33** показано устройство щеточного двигателя постоянного тока. Это двухполюсник, содержащий постоянный неподвижный магнит, *статор*, и вращающийся электромагнит, *ротор*, или *якорь*, соединенный с валом. Передний конец ротора соединяется с металлическим разрезным кольцом, называемым *коллектором*. Металлические щетки, присоединенные к входам питания (входные клеммы), трутся о коллектор, обеспечивая поступление тока к электромагниту ротора. Это индуцирует магнитное поле в роторе, которое заставляет ротор вращаться, так чтобы скорость его вращения совпала с частотой магнитного поля статора. После того как ротор совершит часть полного оборота и скорости окажутся близки, щетки касаются противоположных сторон коллектора, направление тока и магнитного поля изменяется на противоположное, в результате чего вращение может продолжаться неограниченно долго.

Двигатели постоянного тока, как правило, вращаются, совершая тысячи оборотов в минуту (RPM) при очень низком крутящем моменте. В большинство систем добавляют зубчатую передачу, чтобы уменьшить скорость до более приемлемого уровня и увеличить крутящий момент. Поищите зубчатую передачу, подходящую к вашему двигателю. Компания Pittman производит широкую номенклатуру высококачественных двигателей постоянного тока и аксессуаров к ним, а недорогие двигатели для игрушек популярны среди любителей.

Двигатель постоянного тока потребляет значительный ток и напряжение, чтобы обеспечить необходимую мощность на нагрузке. Ток должен быть обратимым, чтобы двигатель мог вращаться в обоих направлениях. Большинство микроконтроллеров не способны вырабатывать ток, достаточный для управления двигателем постоянного тока непосредственно. Поэтому вместе с ними используется H-мост, принципиальная схема которого содержит четыре электрически управляемых переключателя, как показано на **Рис. 4.34 (a)**. Если замкнуты переключатели A и D, ток течет слева направо, и двигатель вращается в одном направлении. Если замкнуты B и C, ток течет справа налево, и двигатель

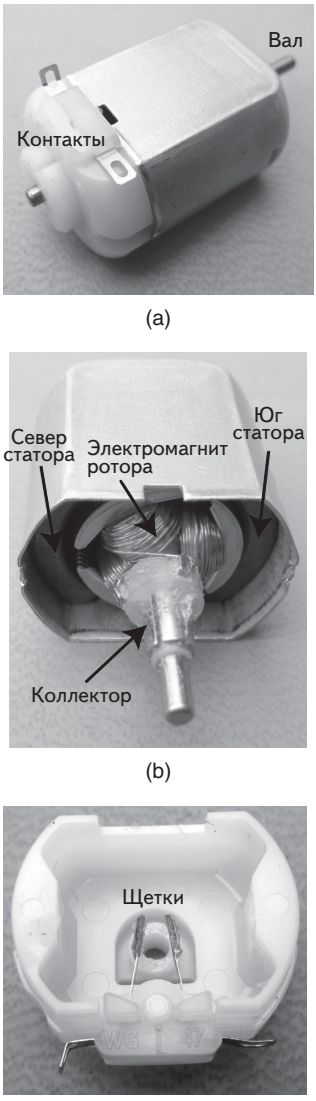


Рис. 4.33. Двигатель постоянного тока

вращается в другом направлении. Если A и C или B и D замкнуты, напряжение на двигателе оказывается равно 0, в результате чего двигатель активно тормозится. Если ни один из переключателей не замкнут, то двигатель будет работать по инерции до полной остановки. В роли переключателей в H-мосте используются транзисторы большой мощности. H-мост также содержит цифровые схемы для удобства управления переключателями.

Когда ток двигателя скачкообразно меняется, индуктивность электромагнитного двигателя индуцирует высокий пик напряжения, что может повредить мощные транзисторы. Поэтому во многих H-мостах имеются защитные диоды, включенные параллельно с переключателями, как показано на **Рис. 4.34 (b)**. Если индуктивный выброс повысит напряжение на любом полюсе двигателя до уровня выше V_{motor} или ниже уровня GND, диоды включатся и удержат напряжение на безопасном уровне. H-мосты могут рассеивать большое количество энергии, так что иногда необходим теплоотвод для охлаждения.

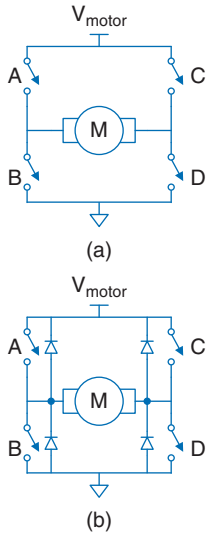


Рис. 4.34. H-мост

Пример 4.12. АВТОНОМНОЕ ТРАНСПОРТНОЕ СРЕДСТВО

Разработайте систему, в которой Raspberry Pi управляла бы двумя электромоторами беспилотного автомобиля. Напишите библиотеку, состоящую из функций для инициализации драйвера двигателя, перемещения автомобиля вперед и назад, поворота вправо и влево и остановки. Используйте ШИМ для управления скоростью двигателей.

Решение. На **Рис. 4.35** приведена схема, в которой два двигателя постоянного тока управляются микроконтроллером Pi с использованием двойного H-моста Texas Instruments SN754410. Этой микросхеме требуется питание напряжением 5 В, поступающее на вход V_{CC1}, а также питание напряжением 4.5–36 В для двигателей, поступающее на V_{CC2}. Напряжение на V_{IH} составляет 2 В, то есть этот порт совместим с портами ввода-вывода Pi, которые работают от напряжения 3.3 В. Микросхема SN754410 обеспечивает постоянный ток силой до 1 А для каждого двигателя. Напряжение V_{motor} должно поступать от отдельного аккумулятора; 5-вольтовый выход Pi не способен вырабатывать ток, достаточный для работы большинства двигателей, это могло бы привести к выходу Pi из строя.

В **Табл. 4.10** показано, как с помощью подачи сигналов на входы H-моста производится управление двигателем. Микроконтроллер управляет скоростью двигателей, используя ШИМ-сигнал. Для управления направлением вращения роторов двигателей используются четыре других контакта.

ШИМ настроена на работу на частоте примерно 5 КГц. Коэффициент заполнения изменяется от 0 до 100%. Если частота ШИМ значительно выше полосы частот двигателя, то будет наблюдаться эффект плавного движения. Отметим, что зависимость между коэффициентом заполнения и скоростью вращения двигателя нелинейна и что при коэффициенте заполнения ниже определенного порога двигатель вообще не будет вращаться.

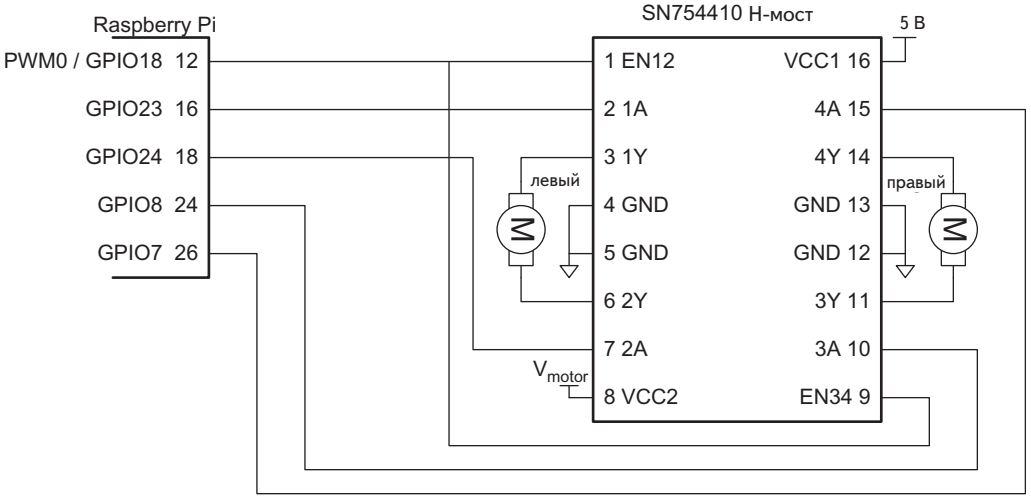


Рис. 4.35. Управление двигателем с помощью двойного Н-моста

Таблица 4.10. Управление Н-мостом

EN12	1A	2A	Двигатель
0	X	X	Вращение по инерции
1	0	0	Торможение
1	0	1	Реверс
1	1	0	Вперед
1	1	1	Торможение

```
#include "EasyPIO.h"

// Константы
#define MOTOR_1A 23
#define MOTOR_2A 24
#define MOTOR_3A 8
#define MOTOR_4A 7

void setSpeed(float dutycycle) {  // pwmInit() должна вызываться первой
    setPWM(5000, dutycycle);
}

void setMotorLeft(int dir) {           // 1 = вперед, 0 = назад
    digitalWrite(MOTOR_1A, dir);
    digitalWrite(MOTOR_2A, !dir);
}

void setMotorRight(int dir) {          // 1 = вперед, 0 = назад
```

```
    digitalWrite(MOTOR_3A, dir);
    digitalWrite(MOTOR_4A, !dir);
}

void forward(void) {
    setMotorLeft(1); setMotorRight(1);  // Оба двигателя вращаются вперед
}

void backward(void) {
    setMotorLeft(0); setMotorRight(0);  // Оба двигателя вращаются назад
}

void left(void) {
    setMotorLeft(0); setMotorRight(1);  // Левый назад, правый вперед
}

void right(void) {
    setMotorLeft(1); setMotorRight(0);  // Правый назад, левый вперед
}

void halt(void) {                       // Выключить оба двигателя
    digitalWrite(MOTOR_1A, 0);
    digitalWrite(MOTOR_2A, 0);
    digitalWrite(MOTOR_3A, 0);
    digitalWrite(MOTOR_4A, 0);
}

void initMotors(void) {
    pinMode(MOTOR_1A, OUTPUT);
    pinMode(MOTOR_2A, OUTPUT);
    pinMode(MOTOR_3A, OUTPUT);
    pinMode(MOTOR_4A, OUTPUT);
    halt();                     // Убедиться, что двигатели не вращаются
    pwmInit();                  // Включить ШИМ
    setSpeed(0.75);             // По умолчанию неполная мощность
}

main(void) {
    pioInit();
    initMotors();
    forward(); delayMillis(5000);
    backward(); delayMillis(5000);
    left(); delayMillis(5000);
    right(); delayMillis(5000);
    halt();
}
```

В предыдущем примере нет возможности измерить положение каждого двигателя. Маловероятно, что оба двигателя будут точно синхронизированы, скорее, один будет вращаться чуть быстрее другого, в результате чего автомобиль будет сбиваться с курса. Для решения этой проблемы в некоторых системах добавляют датчики угла поворота (ДУП). На

Рис. 4.36 (a) показан простой ДУП, состоящий из диска с прорезями, прикрепленного к валу двигателя. На одной стороне размещен светодиод, а на другой – оптический датчик. ДУП генерирует импульс, всякий раз как прорезь проходит мимо светодиода. Микроконтроллер может считать эти импульсы для измерения суммарного угла поворота вала. Если использовать две пары светодиод–датчик, отстоящие друг от друга на половину ширины внутреннего пространства корпуса двигателя, то усовершенствованный таким образом датчик сможет формировать квадратурные выходы, показанные на **Рис. 4.36 (b)**, показывающие не только угол, но и направление поворота вала. Иногда к ДУП добавляют еще один зазор, который показывает, когда вал находится в заданной позиции.

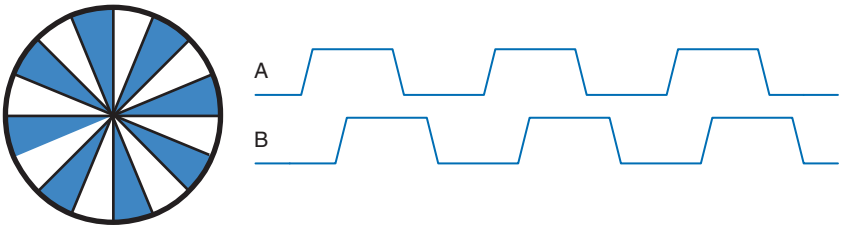


Рис. 4.36. Датчик угла поворота (ДУП): (a) диск; (b) квадратурные выходы

Серводвигатель

Серводвигатель – двигатель постоянного тока, объединенный с зубчатой передачей, ДУП и логикой управления, поэтому он проще в использовании. Такие двигатели имеют ограниченный угол поворота: обычно 180°. На **Рис. 4.37** изображен серводвигатель со снятой крышкой. Серводвигатель имеет 3-контактный интерфейс с контактами питания (как правило, 5 В), земли и управляющим входом. На управляющий вход обычно подается ШИМ-сигнал с рабочей частотой 50 Гц. Логика управления серводвигателя приводит вал в положение, определяемое коэффициентом заполнения входного сигнала управления. В качестве ДУП серводвигателя, как правило, выступает поворотный потенциометр, который выдает напряжение, зависящее от положения вала.

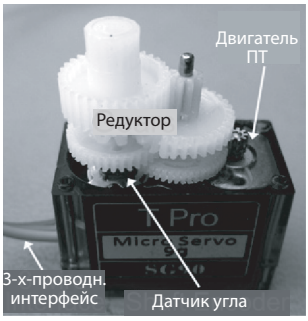


Рис. 4.37. Серводвигатель SG90

В типичном серводвигателе с поворотом на угол до 180° импульс шириной 0.5 мс приводит вал в положение 0°, 1.5 мс – в положение 90° и 2.5 мс – в положение 180°. Например, на **Рис. 4.38** показан управляющий сигнал с шириной импульса 1.5 мс. Выход серводвигателя за пределы рабочего диапазона может привести к упору в механические ограничители и повреждению. Питание серводвигателю поступает

от контакта питания, а не от контакта управления, поэтому управление можно подключить непосредственно к микроконтроллеру без H-моста. Серводвигатели обычно используются в моделях самолетов с дистанционным управлением и небольших роботах из-за их размеров, легкости и удобства. Найти подходящий по характеристикам двигатель не всегда легко. Центральный контакт с красным проводом, как правило, служит для подачи питания, а черный или коричневый провод – обычно земля.

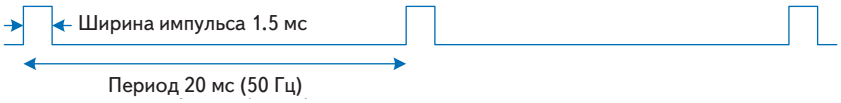


Рис. 4.38. Временная диаграмма управляющего сигнала серводвигателя

Пример 4.13. СЕРВОДВИГАТЕЛЬ

Разработайте систему, в которой Raspberry Pi управляет серводвигателем и поворачивает его вал на заданный угол.

Решение. На **Рис. 4.39** показана схема подключения серводвигателя SG90 с указанием цветов проводов в кабеле. Двигатель работает от источника с напряжением 4.0–7.2 В. Если необходимо большое усилие, то двигатель может вырабатывать ток силой до 0.5 А, но если нагрузка невелика, то может работать прямо от источника питания Raspberry Pi. Для передачи ШИМ-сигнала достаточно одного провода, по которому передается сигнал с напряжением 5 В или 3.3 В. Программа конфигурирует генератор ШИМ и вычисляет коэффициент заполнения, необходимый для поворота вала двигателя на заданный угол. Перебираются углы 0, 90 и 180°.

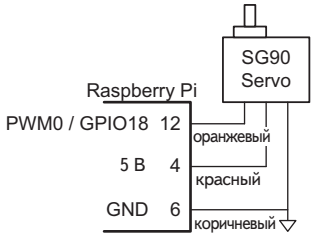


Рис. 4.39. Управление серводвигателем

```
#include "EasyPIO.h"

void setServo(float angle) {
  setPWM(50.0, 0.025 + (0.1 * (angle / 180)));
}

void main(void) {
  pioInit();
  pwmInit();
  while (1) {
    setServo(0.0);       // Влево
    delayMillis(1000);
    setServo(90.0);      // Посередине
    delayMillis(1000);
    setServo(180.0);     // Вправо
    delayMillis(1000);
  }
}
```

Кроме того, можно преобразовать обычный серводвигатель в непрерывно вращающийся сервопривод. Для этого нужно разобрать его, убрать механический упор, а вместо потенциометра установить фиксированный делитель напряжения. На многих веб-сайтах приведены подробные указания для конкретных серводвигателей. В таком случае ШИМ будет контролировать скорость, а не положение; импульс шириной 1.5 мс означает остановку, шириной 2.5 мс – вращение вперед с максимальной скоростью, а шириной 0.5 мс – вращение в обратную сторону с максимальной скоростью. Непрерывно вращающийся серводвигатель может оказаться удобнее и дешевле обычного двигателя постоянного тока в сочетании с H-мостом и зубчатой передачей.

Шаговый двигатель

Шаговый двигатель поворачивается на дискретный шаг, когда на его входы поочередно подается напряжение. Шаг обычно составляет несколько градусов, что позволяет выполнить точное позиционирование и продолжительное вращение. Небольшие шаговые двигатели, как правило, имеют два набора катушек, называемых *фазами*, исполненных в *биполярном* или *униполярном* вариантах. Биполярные двигатели мощнее и дешевле при заданном размере, но требуют наличия H-моста в качестве драйвера, в то время как униполярные двигатели могут управляться транзисторами, работающими как переключатели. В этом разделе рассматривается более эффективный биполярный шаговый двигатель.

На **Рис. 4.40 (a)** показан упрощенный двухфазный биполярный двигатель с шагом 90°. Ротором служит постоянный магнит с одним северным и одним южным полюсом. Статор – электромагнит с двумя парами катушек, содержащий две фазы. Двухфазные биполярные двигатели, таким образом, имеют четыре вывода. На **Рис. 4.40 (b)** показан символ шагового двигателя, где две намотки обозначают катушки индуктивности. В реальных двигателях имеется еще зубчатая передача для уменьшения величины шага и увеличения крутящего момента.

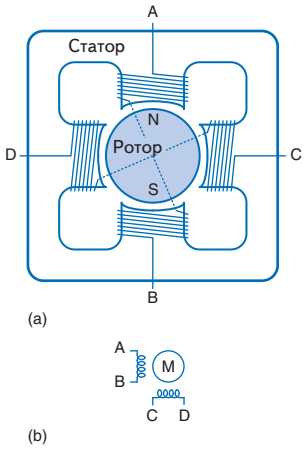
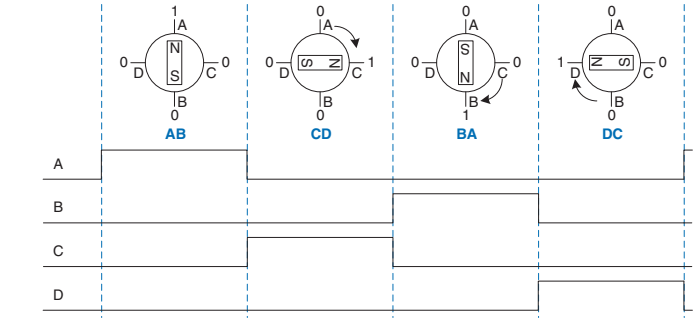


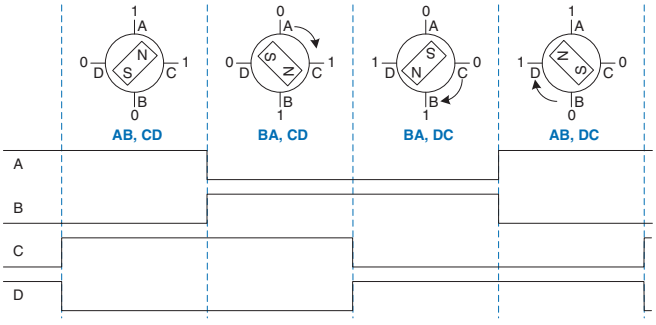
Рис. 4.40. Двухфазный биполярный двигатель: (a) упрощенная схема; (b) символ

На **Рис. 4.41** показаны три основных последовательности управления биполярным двухфазным двигателем. На **Рис. 4.41 (a)** изображен *волновой привод*, когда на катушки подается напряжение в последовательности AB – CD – BA – DC. Отметим, что BA означает, что обмотка AB находится под напряжением с током, текущим в обратную сторону; отсюда и возникло название *биполярный*. Ротор поворачивается на 90 градусов на каждом шаге. На **Рис. 4.41 (b)** показана работа *привода с двумя одновременно включаемыми фазами*, работающего по схеме (AB, CD) – (BA, CD) – (BA, DC) – (AB, DC). (AB, CD) означает, что обе катушки AB и CD находятся под напряжением

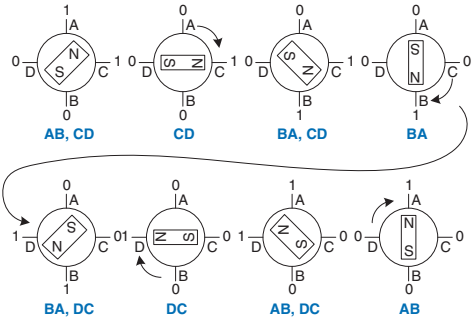
одновременно. В этом случае ротор тоже поворачивается на 90 градусов на каждом шаге, но производится автоматическое выравнивание на полпути между двумя полюсами. При этом получается самый высокий крутящий момент, т. к. обе катушки подают мощность одновременно. На **Рис. 3.66 (c)** иллюстрируется привод *с полушагом*, работающий по схеме (AB, CD) – CD – (BA, CD) – BA – (BA, DC) – DC – (AB, DC) – AB. Ротор поворачивается на 45 градусов на каждом полушаге. Скорость выполнения этой последовательности определяет скорость вращения двигателя. Чтобы изменить направление вращения двигателя, те же управляющие последовательности подаются в обратном порядке.



(a) Волновой привод



(b) Бифазный привод



(c) Полушаговый привод

Рис. 4.41. Биполярный двигатель

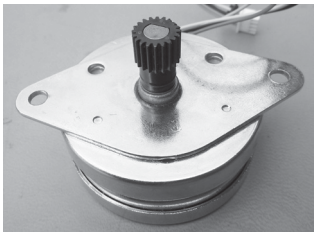


Рис. 4.42. Биполярный шаговый двигатель AIRPAX LB82773-M1

В реальном двигателе ротор имеет много полюсов, чтобы угол между шагами был существенно меньше. Например, на **Рис. 4.42** показан биполярный шаговый двигатель AIRPAXLB82773-M1 с величиной шага 7.5 градуса. Двигатель питается от источника 5 В и подает ток 0.8 А на каждую катушку.

Крутящий момент двигателя пропорционален току в катушке. Этот ток определяется приложенным напряжением, а также индуктивностью L и сопротивлением R катушки. Самый простой режим работы называют *приводом прямого напряжения*, или L/R-приводом, в этом случае напряжение V прикладывается непосредственно к катушке. Ток достигает значения $I = V/R$ за постоянное время, зависящее от L/R, как показано на **Рис. 4.43 (a)**. Этот режим хорошо подходит, когда требуется небольшая скорость. Однако при более высокой скорости ток не успевает достичь максимального уровня, как показано на **Рис. 4.43 (b)**, и крутящий момент падает.

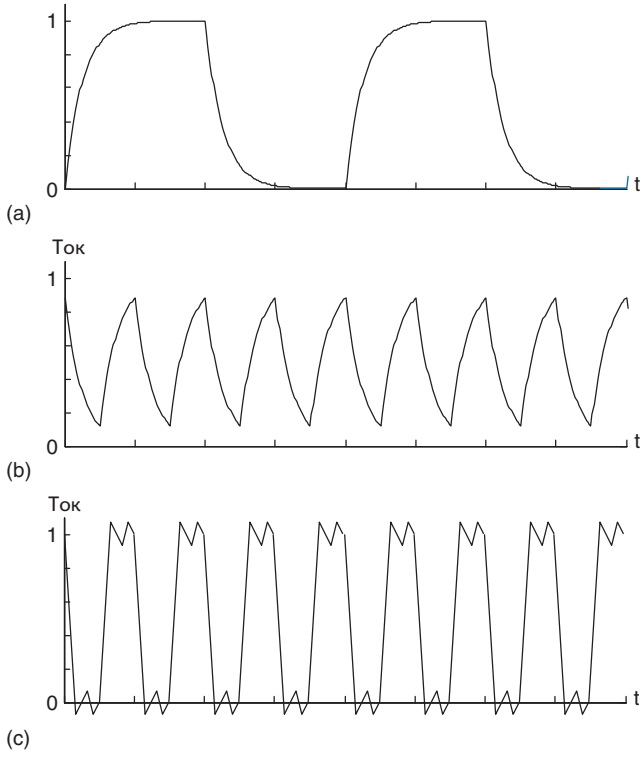


Рис. 4.43. Режим привода прямого напряжения в биполярном шаговом двигателе: a) медленное вращение, b) быстрое вращение, c) быстрое вращение с ограничением постоянного тока

Более эффективный способ управления шаговым двигателем – метод широтно-импульсной модуляции более высокого напряжения. Высокое напряжение заставляет ток нарастать до максимального значения быстрее, после чего напряжение снимается (ШИМ), чтобы избежать перегрузки двигателя. Затем напряжение модулируется или *ограничивается*, чтобы поддерживать ток вблизи требуемого уровня. Этот режим (**Рис. 4.43 (c)**) называется *приводом с ограничителем по постоянному току*. Контроллер содержит небольшой резистор, включенный последовательно с двигателем, и, измеряя падение напряжения на нем, определяет протекающий ток. Когда ток достигает нужного уровня, контроллер посылает разрешающий сигнал H-мосту, чтобы выключить привод. В принципе, микроконтроллер мог бы генерировать сигналы правильной формы, но это проще сделать с помощью контроллера шагового двигателя. Контроллер L297 компании ST Microelectronics – подходящий выбор, особенно в сочетании с двойным H-мостом L298 с контактами для измерения тока и максимально допустимым пиком 2 А. К сожалению, L298 не выпускается в DIP-корпусе, поэтому его сложнее монтировать на макетной плате. Документация компании ST на AN460 и AN470 содержит ценную информацию для разработчиков шаговых двигателей.

Пример 4.14. БИПОЛЯРНЫЙ ШАГОВЫЙ ДВИГАТЕЛЬ В РЕЖИМЕ ПРЯМОГО ВОЛНОВОГО ПРИВОДА

Разработайте систему для придания биполярному шаговому двигателю AIRPAX заданной скорости и направления вращения с помощью прямого волнового привода.

Решение. На **Рис. 4.44** показан биполярный шаговый двигатель, управляемый непосредственно H-мостом с таким же интерфейсом, как в случае двигателя постоянного тока. Отметим, что на ввод MCC2 должны подаваться достаточные напряжение и ток, отвечающие техническим требованиям производителя, иначе двигатель может пропускать шаги по мере увеличения скорости вращения.

```
#include "EasyPIO.h"

#define STEPSIZE 7.5
#define SECS_PER_MIN 60
#define MICROS_PER_SEC 1000000
#define DEG_PER_REV 360

int stepperPins[] = {18, 8, 7, 23, 24};
int curStepState; // Отслеживает текущее положение шагового двигателя

void stepperInit(void) {
   pinsMode(stepperPins, 5, OUTPUT);
   curStepState = 0;
}

void stepperSpin(int dir, int steps, float rpm) {
    int sequence[4] = {0b00011, 0b01001, 0b00101, 0b10001}; //{2A, 1A, 4A, 3A, EN}
```

```
    int step = 0;
    unsigned int microsPerStep = (SECS_PER_MIN * MICROS_PER_SEC * STEPSIZE) /
                                 (rpm * DEG_PER_REV);
    for (step = 0; step < steps; step++) {
        digitalWrites(stepperPins, 5, sequence[curStepState]);
        if (dir == 0) curStepState = (curStepState + 1) % 4;
        else          curStepState = (curStepState + 3) % 4;
        delayMicros(microsPerStep);
    }
}

void main(void) {
    pioInit();
    stepperInit();
    stepperSpin(1, 12000, 120); // Выполнить 60 оборотов со скоростью 120 об/мин
}
```

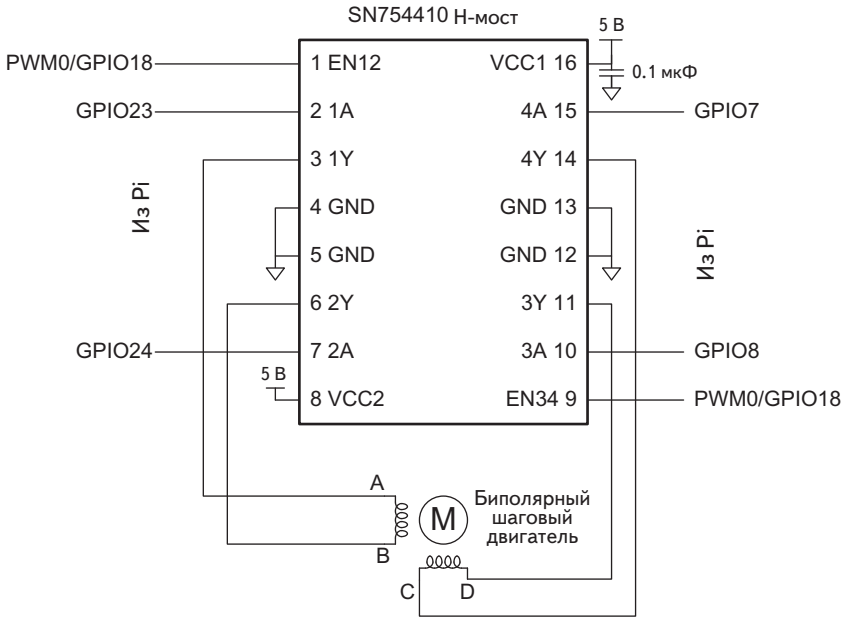


Рис 4.44. Биполярный шаговый двигатель в режиме прямого волнового привода с Н-мостом

4.5. Интерфейсы шин

Интерфейс шины предназначен для подключения к процессору памяти и периферийных устройств. В общем случае интерфейс шины поддерживает один или более задатчиков шины (bus master), которые могут инициировать запросы чтения или записи, и один или более исполнителей

шины (bus slave), отвечающих на запросы. Обычно процессоры выступают в роли задатчиков, а память и периферийные устройства – в роли исполнителей.

Advanced Microcontroller Bus Architecture (AMBA) – открытый стандарт интерфейса шины для соединения компонентов, размещенных на кристалле. Первоначальная версия стандарта была разработана компанией ARM в 1996, затем он претерпел ряд ревизий для повышения производительности и расширения функциональности и стал стандартом де-факто для встраиваемых микроконтроллеров. Advanced High-performance Bus (AHB) – один из стандартных протоколов, определенных в AMBA. AHB-Lite – упрощенный вариант AHB с поддержкой единственного задатчика шины. В этом разделе описывается протокол AHB-Lite с целью проиллюстрировать характеристики типичного интерфейса шины и показать, как проектируются память и периферийные устройства, подключаемые к стандартной шине.

AHB – пример двухточечной шины считывания, противопоставляемый устаревшим архитектурам, в которых использовалась одна общая шина данных, к которой каждый исполнитель получал доступ с помощью тристабильного драйвера. Наличие двухточечной линии связи между каждым исполнителем и мультиплексором считывания ускоряет работу шины и позволяет избежать потерь мощности, когда один исполнитель включает свой драйвер, до того как другой выключился.

4.5.1. AHB-Lite

На **Рис. 4.45** показана простая шина AHB-Lite, соединяющая процессор (задатчик шины) с ОЗУ, ПЗУ и двумя периферийными устройствами (исполнителями). Обратите внимание, что эта шина очень похожа на изображенную на **Рис. 4.1**, отличаются только названия. Задатчик генерирует сигнал синхронизации (HCLK) для всех исполнителей и может сбрасывать их, установив низкий уровень сигнала HRESETn. Задатчик посылает адрес. Дешифратор адреса использует старшие биты для генерации сигнала HSEL, который выбирает исполнителя, а исполнители используют младшие биты, чтобы определить ячейку памяти или регистр. Задатчик помещает подлежащие записи данные на линию HWDATA. Каждый исполнитель помещает прочитанные данные на свою линию HRDATA, а мультиплексор селектирует данные от выбранного исполнителя.

Задатчик помещает на шину 32-битовый адрес в одном такте и записывает или считывает данные в следующем такте. Операция чтения или записи называется *передачей*. В случае записи задатчик устанавливает высокий уровень сигнала HWRITE и помещает на шину подлежащие записи 32-битовые данные HWDATA. В случае чтения задатчик устанавливает низкий уровень сигнала HWRITE, и исполнитель в ответ посылает 32-битовые данные HRDATA. Передачи могут перекрываться во времени, т. е. задатчик может послать адрес следующей передачи, пока происходит чтение или запись для текущей передачи. На **Рис. 4.46** приведена временная диаграмма работы шины для случая, когда за записью сразу следует чтение. Обратите внимание, что данные на один такт отстают от адреса и что передачи частично перекрываются.

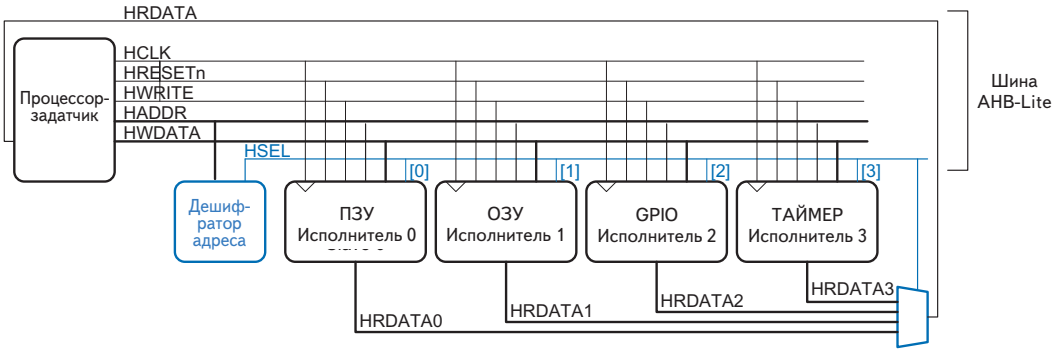


Рис. 4.45. Шина AHB-Lite

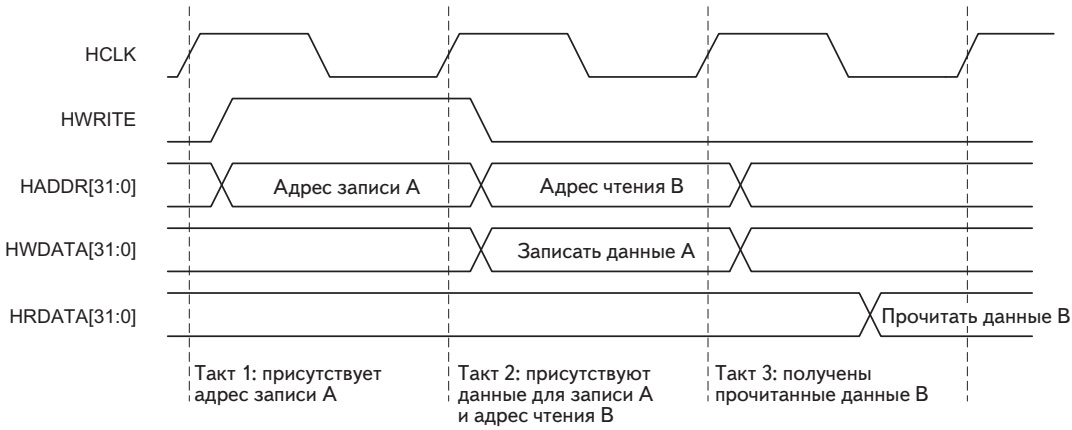


Рис. 4.46. Временная диаграмма работы шины AHB-Lite

В этом примере предполагается, что по шине передается одно 32-битовое слово за операцию и что исполнитель отвечает в одном такте. В протоколе AHB-Lite определены дополнительные сигналы для задания размера передачи (8–1024 бит) и для передачи пакетов от 4 до 16 элементов. Задатчик может также определить тип передачи, защиту и блокировку шины. Исполнитель может убрать сигнал HREADY, означающий, что ему нужно несколько тактов для ответа, или выставить сигнал HRESP, индицирующий ошибку. Интересующиеся читатели могут ознакомиться с документом «AMBA 3 AHB-Lite Protocol Specification», доступным в сети.

4.5.2. Пример интерфейса с памятью и периферийными устройствами

В этом разделе демонстрируется подключение ОЗУ, ПЗУ, GPIO и таймера к процессору по шине AHB-Lite. На **Рис. 4.47** показано отображение

регистров на память для системы на **Рис. 4.45**, имеющей ОЗУ объемом 128 КБ и ПЗУ объемом 64 КБ. GPIO управляет 32 контактами ввода-вывода. 32-битовый регистр GPIO_DIR определяет, предназначен ли контакт для вывода (1) или для ввода (0). В 32-битовый регистр GPIO_PORT можно записать подлежащее выводу значение или прочитать из него возвращенное значение. Модуль Timer напоминает счетчик BCM2835, описанный в **разделе 4.3.5**; он содержит 64-битовый счетчик, увеличивающийся на единицу с частотой HCLK (TIMER_CHI:TIMER_CLO), четыре 32-битовых канала сравнения (TIMER_C3:0) и регистр совпадения (TIMER_CS).

В **примере 4.1** приведен HDL-код системы на языке SystemVerilog. Дешифратор основан на карте памяти. Блоки памяти и периферийные устройства подключены к шине. Неиспользуемые сигналы опущены; например, ПЗУ игнорирует сигналы записи. Модуль GPIO подключен также к 32 контактам, которым может быть назначена роль ввода или вывода.

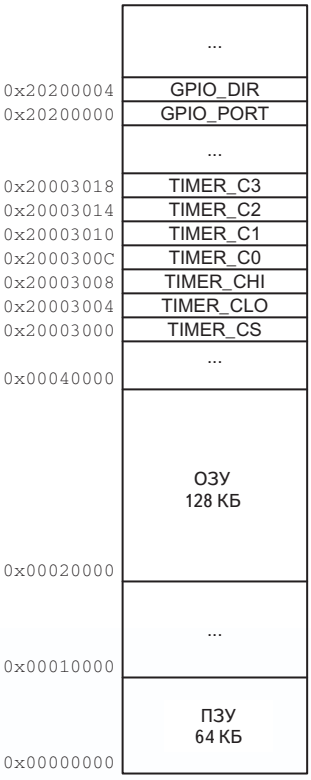


Рис. 4.47. Карта памяти системы

Пример HDL 4.1

```
module ahb_lite(input  logic        HCLK,
                input  logic        HRESETn,
                input  logic [31:0] HADDR,
                input  logic        HWRITE,
                input  logic [31:0] HWDATA,
                output logic [31:0] HRDATA,
                inout  tri [31:0]   pins);

 logic [3:0] HSEL;
 logic [31:0] HRDATA0, HRDATA1, HRDATA2,
              HRDATA3;
 logic [31:0] pins_dir, pins_out, pins_in;
 logic [31:0] HADDRDEL;
 logic HWRITEDEL;

 // Задержать сигналы адреса и записи для
 // выравнивания во времени с данными
 flop #(32) adrreg(HCLK, HADDR, HADDRDEL);
 flop #(1) writereg(HCLK, HWRITE, HWRITEDEL);

 // Декодирование карты памяти
 ahb_decoder dec(HADDRDEL, HSEL);
 ahb_mux mux(HSEL, HRDATA0, HRDATA1, HRDATA2,
             HRDATA3, HRDATA);
 // Блоки памяти и периферийные устройства
 ahb_rom rom (HCLK, HSEL[0], HADDRDEL[15:2], HRDATA0);
```

```
 ahb_ram ram (HCLK, HSEL[1], HADDRDEL[16:2], HWRITEDEL, HWDATA, HRDATA1);
 ahb_gpio gpio (HCLK, HRESETn, HSEL[2], HADDRDEL[2], HWRITEDEL, HWDATA, HRDATA2,
pins_dir, pins_out, pins_in);
 gpio_pins_gpio_drv (pins_dir, pins_out, pins_in, pins);
 ahb_timer timer(HCLK, HRESETn, HSEL[3], HADDRDEL[4:2], HWRITEDEL, HWDATA, HRDATA3);
endmodule

module ahb_decoder(input  logic [31:0] HADDR,
                   output logic [3:0] HSEL);

 // Декодировать, исходя из старших бит адреса
 assign HSEL[0]=(HADDR[31:16] ==16'h0000);    // 64KB ROM на 0x00000000 – 0x0000FFFF
 assign HSEL[1]=(HADDR[31:17] ==15'h0001);    // 128KB RAM на 0x00020000 - 0x003FFFFF
 assign HSEL[2]=(HADDR[31:4]  ==28'h2020000); // GPIO на 0x20200000 - 0x20200007
 assign HSEL[3]=(HADDR[31:8]  ==24'h200030);  // Таймер на 0x20003000 - 0x2000301B
endmodule

module ahb_mux(input  logic [3:0]  HSEL,
               input  logic [31:0] HRDATA0, HRDATA1, HRDATA2, HRDATA3,
               output logic [31:0] HRDATA);
 always_comb
   casez(HSEL)
     4'b???1: HRDATA <= HRDATA0;
     4'b??10: HRDATA <= HRDATA1;
     4'b?100: HRDATA <= HRDATA2;
     4'b1000: HRDATA <= HRDATA3;
   endcase
endmodule

module ahb_ram(input  logic        HCLK,
               input  logic        HSEL,
               input  logic [16:2] HADDR,
               input  logic        HWRITE,
               input  logic [31:0] HWDATA,
               output logic [31:0] HRDATA);

 logic [31:0] ram[32767:0]; // ОЗУ 128КБ, организованное в виде блоков 32К x 32 бит

 assign HRDATA = ram[HADDR]; // *** проверить корректность адресации

 always_ff @(posedge HCLK)
   if (HWRITE & HSEL) ram[HADDR] <= HWDATA;
endmodule

module ahb_rom(input  logic        HCLK,
               input  logic        HSEL,
               input  logic [16:2] HADDR,
               output logic [31:0] HRDATA);

 logic [31:0] rom[16383:0]; // ПЗУ 64КБ, организованное в виде блоков 16К × 32 бит

 // *** загрузить в ПЗУ содержимое файла на диске

 assign HRDATA = rom[HADDR]; // *** проверить корректность адресации
endmodule
```

```
module ahb_gpio(input  logic        HCLK,
                input  logic        HRESETn,
                input  logic        HSEL,
                input  logic [2]    HADDR,
                input  logic        HWRITE,
                input  logic [31:0] HWDATA,
                output logic [31:0] HRDATA,
                output logic [31:0] pin_dir,
                output logic [31:0] pin_out,
                input  logic [31:0] pin_in);

 logic [31:0] gpio[1:0]; // регистры GPIO

 // записать в выбранный регистр
 always_ff @(posedge HCLK or negedge HRESETn)
   if (~HRESETn) begin
     gpio[0] <= 32'b0; // GPIO_PORT
     gpio[1] <= 32'b0; // GPIO_DIR
   end else if (HWRITE & HSEL)
     gpio[HADDR] <= HWDATA;

 // прочитать выбранный регистр
 assign HRDATA = HADDR ? gpio[1] : pin_in;

 // отправить значения и направление драйверам ввода-вывода
 assign pin_out = gpio[0];
 assign pin_dir = gpio[1];
endmodule

module ahb_timer(input  logic        HCLK,
                 input  logic        HRESETn,
                 input  logic        HSEL,
                 input  logic [4:2]  HADDR,
                 input  logic        HWRITE,
                 input  logic [31:0] HWDATA,
                 output logic [31:0] HRDATA);

 logic [31:0] timers[6:0]; // регистры таймера
 logic [31:0] chi, clo; // следующее значение счетчика
 logic [3:0] match, clr; // совпадает ли счетчик с регистром сравнения

 // записать в выбранный регистр и обновить регистры таймера и совпадения
 always_ff @(posedge HCLK or negedge HRESETn)
 if (~HRESETn) begin
    timers[0] <= 32'b0; // TIMER_CS
    timers[1] <= 32'b0; // TIMER_CLO
    timers[2] <= 32'b0; // TIMER_CHI
    timers[3] <= 32'b0; // TIMER_C0
    timers[4] <= 32'b0; // TIMER_C1
    timers[5] <= 32'b0; // TIMER_C2
    timers[6] <= 32'b0; // TIMER_C3
 end else begin
    timers[0] <= {28'b0, match};
    timers[1] <= (HWRITE & HSEL & HADDR == 3'b001 ) ? HWDATA : clo;
```

```
    timers[2] <= (HWRITE & HSEL & HADDR == 3'b010) ? HWDATA : chi;
    if (HWRITE & HSEL & HADDR == 3'b011) timers[3] <= HWDATA;
    if (HWRITE & HSEL & HADDR == 3'b100) timers[4] <= HWDATA;
    if (HWRITE & HSEL & HADDR == 3'b101) timers[5] <= HWDATA;
    if (HWRITE & HSEL & HADDR == 3'b110) timers[6] <= HWDATA;
 end

 // прочитать выбранный регистр
 assign HRDATA = timers[HADDR];

 // увеличить 64-битовый счетчик, представленный в виде пары TIMER_CHI, TIMER_CLO
 assign {chi, clo} = {timers[2], timers[1]} + 1;

 // установить биты совпадения: бит устанавливается, если счетчик совпадает с
 // соответствующим регистром сравнения
 // очистить бит, если в эту позицию регистра совпадения записана 1
 assign clr = (HWRITE & HSEL & HADDR == 3'b000 & HWDATA[3:0]);
 assign match[0] = ~clr[0] & (timers[0][0] | (timers[1] == timers[3]));
 assign match[1] = ~clr[1] & (timers[0][1] | (timers[1] == timers[4]));
 assign match[2] = ~clr[2] & (timers[0][2] | (timers[1] == timers[5]));
 assign match[3] = ~clr[3] & (timers[0][3] | (timers[1] == timers[6]));
endmodule

module gpio_pins(input  logic [31:0] pin_dir, // 1 = вывод, 0 = ввод
                 input  logic [31:0] pin_out, // значение, подаваемое на выводы
                 output logic [31:0] pin_in,  // значение, считываемое с ввода
                 inout  tri [31:0] pin);      // тристабильные контакты

 // Тристабильные контакты управляются индивидуально
 // В SystemVerilog нет способа управлять тристабильным состоянием побитово
 genvar i;
 generate
 for (i = 0; i<32; i = i + 1) begin: pinloop
     assign pin[i] = pin_dir[i] ? pin_out[i] : 1'bz;
 end
 endgenerate

 assign pin_in = pin;
endmodule
```

4.6. Интерфейсы ввода-вывода персональных компьютеров

В персональных компьютерах (ПК) применяется множество протоколов ввода-вывода для различных целей: передачи и получения данных из памяти, работы с дисками и сетью, с внутренними картами расширений и с внешними устройствами. Эти протоколы развивались, чтобы обеспечить высочайшую производительность и дать пользователям возможность без

труда подключать внешние устройства. Но за это приходится расплачиваться сложностью протоколов ввода-вывода. В настоящем разделе рассматриваются основные стандарты ввода-вывода современных ПК, а также возможности подключения к ПК пользовательской цифровой логики и другого внешнего оборудования.

На **Рис. 4.48** изображена материнская плата с разъемом для процессора Core i5 или i7. Процессор помещен в корпус с 1156 золочеными контактными площадками – LGA-корпус (land grid array, LGA). Эти площадки служат для подвода к процессору питания, земли, подсоединения к нему памяти и устройств ввода-вывода. На материнской плате расположено два слота для оперативной памяти (DRAM), различные разъемы для устройств ввода-вывода и разъем для подключения питания, регуляторы напряжений и конденсаторы. Оперативная память подключается через интерфейс DDR3. Периферийные устройства, например клавиатуры или веб-камеры, подключаются через интерфейс USB. Высокопроизводительные платы расширения, например видеокарты, подключаются к слоту PCI Express с 16 линиями передачи (x16), в то время как для менее требовательных по производительности карт расширения можно использовать слот с одной линией передачи (x1) или более старый интерфейс PCI. ПК подключается к сети через Ethernet-кабель. Жесткий диск подключается к разъему SATA. Далее мы дадим общее представление о работе каждого из перечисленных стандартов ввода-вывода.

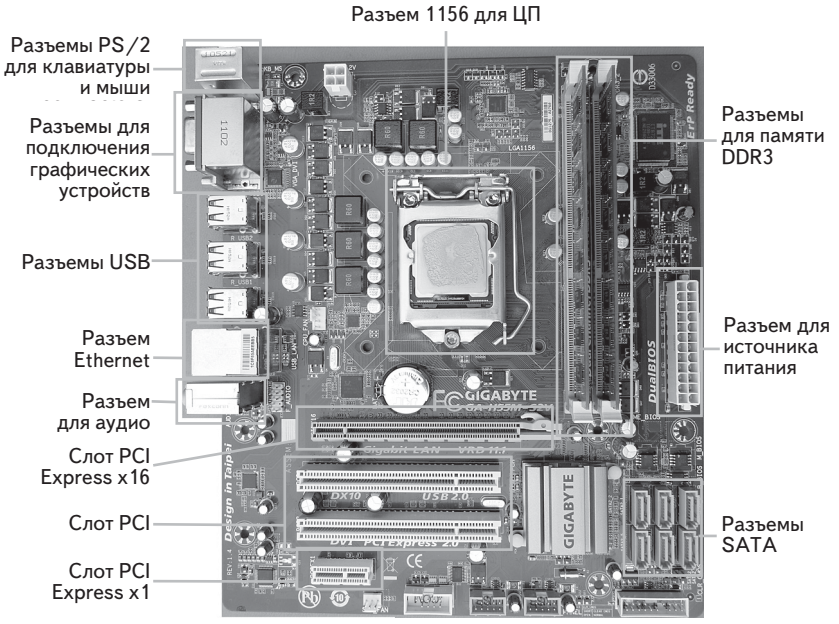


Рис. 4.48. Материнская плата Gigabyte GA-H55M-S2V

Одним из главных достижений в стандартах ввода-вывода ПК является разработка высокоскоростных последовательных интерфейсов. До недавнего времени большинство интерфейсов были параллельными. Они включали широкую шину данных и отдельный сигнал синхронизации. Максимально возможная скорость передачи данных ограничивалась разницей во времени прохождения сигналов по проводникам шины. К тому же у шин, подключаемых к нескольким устройствам, возникают проблемы, присущие длинным линиям передачи: отражения и разное время прохождения к различным нагрузкам. Шум также может исказить данные. Двухточечные последовательные интерфейсы решают многие из этих проблем. Данные обычно передаются по дифференциальной паре. Внешним шумом, который одинаково воздействует на оба проводника в паре, можно пренебречь. Линии передачи проще подводить, поэтому отражения малы (см. информацию о линиях передачи в **разделе A.8** (книга 1)). Сигнал синхронизации не подается в явном виде; вместо этого приемник восстанавливает синхронизацию по временам переключения данных (из 0 в 1 и из 1 в 0). Проектирование высокоскоростных последовательных интерфейсов – отдельный предмет. Хорошие интерфейсы могут работать на скоростях более 10 Гбит/с по медным проводникам, а по оптоволокну – даже быстрее.

4.6.1. USB

До середины 1990-х годов подключение периферийных устройств к ПК требовало технической подкованности. Для добавления плат расширения необходимо было открыть корпус, выставить перемычки в правильное положение и вручную устанавливать драйвер устройства. Для добавления устройства через интерфейс RS-232 требовалось найти подходящий кабель и правильно настроить скорость передачи данных, количество бит данных, режим контроля четности и количество стоповых битов. *Универсальная последовательная шина* (USB), разработанная компаниями Intel, IBM, Microsoft и другими, значительно упростила добавление периферийных устройств благодаря стандартизации кабелей и процесса конфигурирования программного обеспечения. В настоящее время каждый год продаются миллиарды устройств, подключаемых к компьютеру через интерфейс USB.

Стандарт USB 1.0 был выпущен в 1996 году. В нем используется простой кабель с четырьмя проводниками: питание (+5 В), земля и дифференциальная пара для передачи данных. Кабель невозможно подключить не той стороной или «вверх ногами». Данные по интерфейсу передаются со скоростью до 12 Мбит/с. Подключаемое устройство может получать ток силой до 500 мА от порта USB, поэтому для клавиатуры, мыши и других периферийных устройств не нужны батарейки или отдельные кабели питания.

Стандарт USB 2.0, выпущенный в 2000 году, поднял скорость передачи данных до 480 Мбит/с за счет увеличения скорости работы дифференциальной пары. После повышения производительности появилась возможность подключать веб-камеры и внешние жесткие диски. Флеш-накопители с интерфейсом USB также пришли на смену гибким магнитным дискам (дискетам) в качестве средства передачи файлов между компьютерами.

Стандарт USB 3.0, выпущенный в 2008 году, принес еще большее увеличение скорости передачи данных – до 5 Гбит/с. В нем используется разъем той же формы, но кабель состоит из большего числа проводов, сигнал по которым передается на очень высокой скорости. Он позволяет подключать высокопроизводительные жесткие диски. Примерно в то же время в стандарт USB была добавлена спецификация зарядки аккумуляторных батарей, что резко увеличило мощность, подаваемую на порт, и позволило ускорить зарядку мобильных устройств.

Простота использования достигается за счет гораздо более сложной аппаратной и программной реализации. Построение USB-интерфейса с нуля – далеко не тривиальная задача. Даже написать простой драйвер устройства – и то нелегко.

4.6.2. PCI и PCI Express

Шина связи периферийных устройств (PCI) – стандарт шин расширения, разработанный компанией Intel и получивший широкое распространение с 1994 года. Этот интерфейс использовался для добавления таких плат расширения, как дополнительные порты последовательного ввода-вывода или USB-порты, сетевые карты, звуковые карты, модемы, дисковые контроллеры и видеокарты. Интерфейс представляет собой 32-разрядную параллельную шину, работающую на частоте 33 МГц и обеспечивающую полосу пропускания шириной 133 МБ/с.

Спрос на платы расширения PCI неуклонно падал. В настоящее время в материнскую плату интегрируется больше стандартных портов, таких как Ethernet и SATA. Многие устройства, которые когда-то исполнялись только в виде плат расширения, сейчас можно подключить через быстрые порты USB 2.0 или USB 3.0. А видеокарты требуют гораздо более широкой полосы пропускания, чем может обеспечить PCI.

На современных материнских платах до сих пор часто можно встретить несколько слотов PCI, но высокопроизводительные устройства, в частности видеокарты, подключаются через интерфейс *PCI Express* (PCIe). Слоты PCI Express содержат одну или несколько линий высокоскоростного последовательного интерфейса. Согласно стандарту PCIe 3.0, каждая линия может обеспечить пропускную способность до 8 Гбит/с. На большинстве материнских плат располагается слот с 16 линиями, которые в сумме обеспечивают полосу пропускания шириной 16 ГБ/с для таких ресурсоемких устройств, как видеокарты.

4.6.3. Память DDR3

Динамическое ОЗУ (Dynamic Random Access Memory, DRAM) подключается к микропроцессору по параллельной шине. В 2015 году стандартом являлся DDR3 – третье поколение шин памяти с удвоенной скоростью передачи данных (double-data rate, DDR) с питанием от напряжения 1.5 В. Типичные материнские платы теперь содержат два канала DDR3, поэтому могут одновременно обращаться к двум банкам модулей памяти. На смену DDR3 идет стандарт DDR4, работающий от напряжения 1.2 В и обеспечивающий более высокую скорость.

На **Рис. 4.49** изображен модуль памяти DDR3 с двухрядным расположением выводов (DIMM) емкостью 4 ГБ. С каждой его стороны находится 120 контактов, что в сумме дает 240 контактов, в число которых входят 64-разрядная шина данных, 16-разрядная адресная шина с временным уплотнением, управляющие сигналы и многочисленные контакты для земли и питания. В 2015 году емкость типичного модуля DIMM составляла от 1 до 16 ГБ динамической памяти. Емкость удваивается примерно каждые 2–3 года.

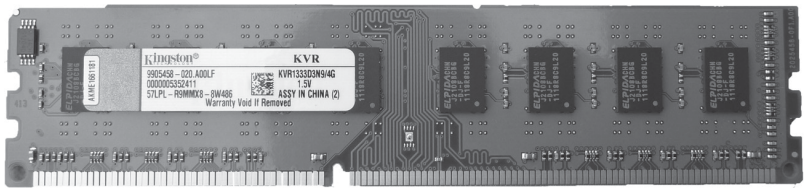


Рис. 4.49. Модуль памяти DDR3

В настоящее время DRAM работают на тактовой частоте 100–266 МГц. DDR3 работает с шиной памяти на тактовой частоте, в 4 раза большей, чем частота сигнала синхронизации DRAM. Кроме того, данные передаются по переднему и заднему фронтам сигнала синхронизации. Следовательно, на каждом такте передается 8 машинных слов. Для 64-битовых слов это соответствует полосе пропускания шириной 6.4–17 ГБ/с. Например, в DDR3-1600 используются тактовая частота памяти 200 МГц и частота ввода-вывода 800 МГц, что позволяет передавать 1600 миллионов слов/сек, или 12 800 МБ/с. Поэтому такие модули также называют PC3-12800. К сожалению, задержка при обращении к DRAM остается высокой – время между отправкой запроса на чтение и прибытием первого слова данных составляет примерно 50 нс.

4.6.4. Сеть

Компьютеры подключаются к Интернету через сетевой интерфейс по протоколу TCP/IP. Физическое соединение может осуществляться посредством Ethernet-кабеля или по беспроводной сети Wi-Fi.

Протокол *Ethernet* определен в стандарте IEEE 802.3. Он был разработан в исследовательском центре Xerox в Пало-Альто (PARC) в 1974 году. Изначально он работал на скорости 10 Мбит/с, теперь – обычно на скорости 100 Мбит/с и 1 Гбит/с. Сигнал передается по кабелю категории 5, содержащему 4 витые пары. 10-гигабитный Ethernet, работающий на оптоволоконных кабелях, становится все популярнее для серверных и других высокопроизводительных вычислений. На подходе уже Ethernet 100 Гбит.

Wi-Fi – популярное название стандарта построения беспроводных сетей IEEE 802.11. Он работает в нелицензионных диапазонах беспроводной связи на частотах 2.4 и 5 ГГц. Это означает, что для передачи в этих диапазонах на низкой мощности пользователю не нужно получать лицензию радиооператора. В **Табл. 4.11** представлены возможности трех поколений Wi-Fi. Появление стандарта 802.11ac обещает увеличение скорости беспроводной передачи данных свыше 1 Гбит/с. Производительность повышена благодаря прогрессу в области модуляции и обработки сигнала, увеличению количества антенн и использованию более широкой полосы пропускания.

Таблица 4.11. Протоколы 802.11 (Wi-Fi)

Протокол	Год	Частота, ГГц	Скорость передачи данных, Мбит/с	Дальность, м
802.11b	1999	2.4	5.5–11	35
802.11g	2003	2.4	6–54	38
802.11n	2009	2.4/5	7.2–150	70
802.11ac	2013	5	433+	переменная

4.6.5. SATA

Внутренние жесткие диски требуют быстрого соединения с ПК. В 1986 году компания Western Digital представила интерфейс IDE (Integrated Drive Electronics), который со временем превратился в стандарт соединения ATA (AT Attachment). В стандарте используется широкий шлейф с 40 или 80 проводниками максимальной длиной 18 дюймов (45 см) для передачи данных со скоростями от 16 до 133 МБ/с.

Стандарт ATA был вытеснен стандартом Serial ATA, подразумевающим использование высокоскоростного последовательного канала, работающего со скоростью 1.5, 3 или 6 Гбит/с, с более удобным 7-жильным кабелем, показанным на **Рис. 4.50**. Самые быстрые твердотельные диски в 2015 году достигали пропускной способности в 500 МБ/с и тем самым полностью раскрыли возможности SATA.

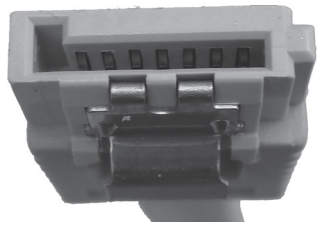


Рис. 4.50. Кабель SATA

С SATA тесно связан стандарт SAS (Serial Attached SCSI) – результат эволюции параллельного интерфейса SCSI (Small Computer System Interface). SAS предлагает сравнимую с SATA производительность и поддерживает более длинные кабели. Этот интерфейс обычно применяется в серверных компьютерах.

4.6.6. Подключение к ПК

Все стандарты ввода-вывода ПК, описанные выше, оптимизированы с учетом высокой производительности и простоты подключения, однако их трудно реализовать аппаратно. Инженерам и научным работникам часто нужен способ подключения к ПК внешних устройств: датчиков, приводов, микроконтроллеров или ПЛИС. Последовательного интерфейса, описанного в **разделе 3.6.3**, достаточно для низкоскоростного подключения к микроконтроллеру с помощью UART. В этом разделе вводится еще два понятия: система сбора данных и USB-подключение.

Системы сбора данных

Система сбора данных (ССД) служит для подключения компьютера к реальному миру посредством нескольких аналоговых и (или) цифровых каналов ввода-вывода. ССД сейчас широко доступны в качестве USB-устройств, что упрощает их установку. Компания National Instruments (NI) – лидер по производству ССД.

Высокопроизводительные ССД стоят тысячи долларов, в основном потому, что рынок узкий и конкуренция на нем ограничена. К счастью, теперь NI продает студентам удобную систему myDAQ с программным обеспечением LabVIEW по цене 200 долларов. На **Рис. 4.51** изображена myDAQ. У нее есть два аналоговых канала ввода-вывода, способных производить до 200 тысяч отсчетов в секунду с 16-битовым разрешением в динамическом диапазоне ±10 В. Эти каналы можно сконфигурировать для работы в качестве осциллографа и генератора сигналов. Устройство имеет также 8 цифровых линий ввода-вывода, совместимых с 3.3 и 5-вольтовыми системами. Кроме того, оно оснащено силовыми выходами с напряжением +5, +15 и −15 В, а также включает цифровой мультиметр, измеряющий напряжение, ток и сопротивление. Таким образом, myDAQ может заменить целый измерительно-испытательный стенд и при этом еще автоматически производит регистрацию данных.

Большинство ССД компании NI управляются с помощью LabVIEW – графического языка для проектирования измерительных и управляющих систем. Некоторыми ССД также можно управлять из программ, написанных на языке C в среде LabWindows, а также из приложений для Microsoft.NET в среде Measurement Studio, или с помощью MatLab, используя Data Acquisition Toolbox.

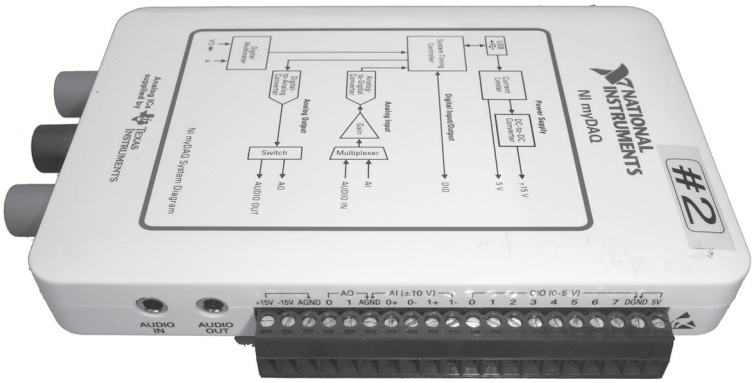


Рис. 4.51. Система сбора данных myDAQ фирмы National Instruments

USB-подключение

Все больше продуктов оснащаются простым и дешевым цифровым USB-интерфейсом. Вместе с этими продуктами поставляются драйверы и библиотеки, позволяющие пользователю написать простую программу на своем ПК, которая обменивается данными с ПЛИС или микроконтроллером.

Компания FTDI занимает лидирующие позиции на рынке таких систем. Например, у кабеля FTDI C232HM-DDHSL USB to MPSSE (Multi-Protocol Synchronous Serial Engine), изображенного на **Рис. 4.52**, на одном конце имеется USB-разъем, а на другом – интерфейс SPI, работающий со скоростью до 30 Мбит/с, а также линия питания 3.3 В и 4 контакта ввода-вывода общего назначения (GPIO). На **Рис. 4.53** изображен пример подключения ПК к ПЛИС с помощью этого кабеля. При необходимости по кабелю можно подавать на ПЛИС напряжение 3.3 В. Три контакта SPI подключены к ведомой ПЛИС, как в **примере 4.4**. На рисунке также показано, что один из контактов GPIO используется для управления светодиодом.

Рис. 4.52. Кабель USB to MPSSE компании FTDI (печатается с разрешения компании FTDI)

На ПК необходимо установить драйвер в виде динамически подключаемой библиотеки D2XX. После этого можно приступать к написанию программы на языке C, используя библиотеку для передачи данных по кабелю.

Для случаев, когда требуется более высокоскоростное соединение, FTDI предлагает модуль UM232H, изображенный на **Рис. 4.54**. Модуль является мостом между USB и 8-битовым синхронным параллельным интерфейсом, работающим со скоростью до 40 МБ/с.

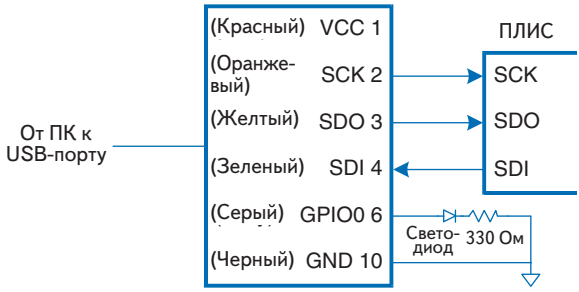


Рис. 4.53. Подключение ПК к ПЛИС USB-кабелем C232HM-DDHSL

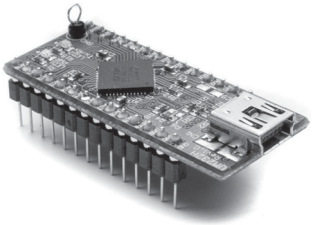


Рис. 4.54. Модуль FTDI UM232H (печатается с разрешения компании FTDI)

4.7. Резюме

В большинстве процессоров для взаимодействия с внешним миром применяется ввод-вывод, отображенный на память. Микроконтроллеры предлагают широкий спектр простых периферийных устройств: ввод-вывод общего назначения, последовательный и аналоговый вводы-выводы, таймеры. ПК и более высокотехнологичные микроконтроллеры поддерживают и более развитые стандарты ввода-вывода, включая USB, Ethernet и SATA.

В этой главе было приведено много конкретных примеров ввода-вывода с применением одноплатного компьютера Raspberry Pi. Проектировщики встраиваемых систем постоянно сталкиваются с новыми процессорами и периферийными устройствами. Общий принцип разработки простой встраиваемой системы ввода-вывода заключается в том, чтобы, ознакомившись с технической документацией, узнать о возможностях устройства и о том, какие у него есть контакты и отображаемые на память регистры. После этого обычно несложно написать простой драйвер, который инициализирует устройство и будет передавать и принимать данные.

В случае более сложных стандартов, например USB, написание драйвера – узкоспециализированная задача, которую лучше оставить специалисту, хорошо знакомому с самим устройством и стеком протоколов USB. Непрофессиональным разработчикам следует выбирать процессор, для которого имеются хорошо протестированные драйверы и примеры кода для представляющих интерес устройств.

Эпилог

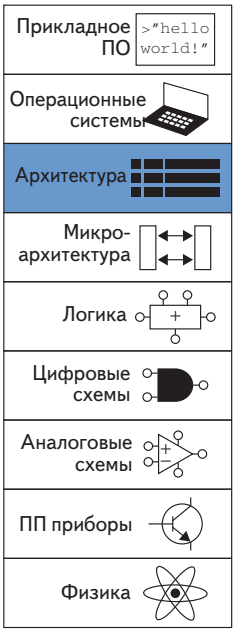
Вот и подошел конец нашего путешествия по миру цифровых систем. Мы надеемся, что в этой книге мы смогли донести не только красоту и увлекательность искусства их проектирования, но и инженерные знания. Вы изучили, как разрабатывать комбинационную и последовательную логику на уровне схем и языков описания аппаратуры. Вы познакомились с более крупными строительными блоками, такими как мультиплексоры, АЛУ и память. Компьютеры – одно из наиболее чарующих приложений цифровых систем. Вы изучили, как программировать процессор ARM на языке ассемблера и как построить процессор и систему памяти из цифровых строительных блоков. В процессе чтения вы наблюдали применение принципов абстракции, дисциплины, иерархии, модульности и регулярности. С их помощью мы сложили пазл внутреннего устройства микропроцессора. От мобильных телефонов до цифрового телевидения, от марсоходов до медицинских систем визуализации – наш мир становится все более и более цифровым.

Представьте, какую цену был бы готов, как Фауст, заплатить Чарльз Бэббидж, чтобы узнать все это полтора столетия назад. Он мечтал всего лишь вычислять математические таблицы с механической точностью. Сегодняшние цифровые системы еще вчера были фантастикой. Мог бы Дик Трейси[5] слушать iTunes на своем телефоне? Запустил бы Жюль Верн в космос навигационные спутники? Мог бы Гиппократ лечить с помощью МРТ? В то же время оруэлловский кошмар повсеместной государственной слежки становится ближе к реальности день ото дня. Хакеры и правительства ведут необъявленные кибервойны, атакуя промышленную инфраструктуру и финансовые системы. Страны-изгои разрабатывают ядерное оружие с помощью ноутбуков, более мощных, чем суперкомпьютеры, занимавшие целые машинные залы и использовавшиеся при расчете бомб времен холодной войны. Микропроцессорная революция продолжает ускоряться. Темп грядущих изменений превзойдет их темп в прошедшие десятилетия. Теперь у вас есть инструменты для разработки и построения систем, которые сформируют наше будущее. С этими знаниями приходит и большая ответственность. Мы надеемся, что вы используете их не только для развлечения и обогащения, но и на пользу человечеству.

[5] Детектив, персонаж комикса 1930-х годов, у которого был телефон в наручных часах. – *Прим. перев.*

Приложение А

Система команд ARM



В этом приложении приведена сводка команд ARMv4, рассмотренных в книге. Коды условий приведены в **Табл. 1.3**.

А.1. Команды обработки данных

Стандартные команды обработки данных кодируются, как показано на **Рис. А.1**. В 4-битовом поле *cmd* задается тип команды, как показано в **Табл. А.1**. Если бит *S* равен 1, то после выполнения команды регистр состояния обновляется в соответствии с выставленными флагами условий. Бит *I*, а также биты 4 и 7 определяют одну из трех кодировок второго операнда-источника, *Src2*, как описано в **разделе 1.4.2**. Поле *cond* определяет, какие коды условий проверять, как описано в **разделе 1.3.2**.

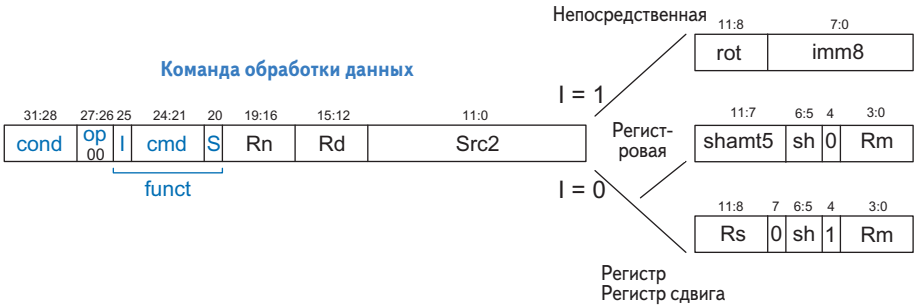


Рис. A.1. Кодирование команд обработки данных

Таблица A.1. Команды обработки данных

cmd	Название	Описание	Операция
0000	`AND Rd, Rn, Src2`	Поразрядное И	`Rd ← Rn & Src2`
0001	`EOR Rd, Rn, Src2`	Поразрядное ИСКЛЮЧАЮЩЕЕ ИЛИ	`Rd ← Rn ^ Src2`
0010	`SUB Rd, Rn, Src2`	Вычитание	`Rd ← Rn - Src2`
0011	`RSB Rd, Rn, Src2`	Инвертированное вычитание	`Rd ← Src2 - Rn`
0100	`ADD Rd, Rn, Src2`	Сложение	`Rd ← Rn+Src2`
0101	`ADC Rd, Rn, Src2`	Сложение с переносом	`Rd ← Rn+Src2+C`
0110	`SBC Rd, Rn, Src2`	Вычитание с переносом	`Rd ← Rn - Src2 -` $\overline{C}$
0111	`RSC Rd, Rn, Src2`	Инвертированное вычитание с переносом	`Rd ← Src2 - Rn -` $\overline{C}$
1000 ($S = 1$)	`TST Rn, Src2`	Тест	`Установить флаги в соответствии с Rn & Src2`
1001 ($S = 1$)	`TEQ Rn, Src2`	Тест на эквивалентность	`Установить флаги в соответствии с Rn ^ Src2`
1010 ($S = 1$)	`CMP Rn, Src2`	Сравнение	`Установить флаги в соответствии с Rn - Src2`
1011 ($S = 1$)	`CMN Rn, Src2`	Сравнение с противоположным	`Установить флаги в соответствии с Rn + Src2`
1100	`ORR Rd, Rn, Src2`	Поразрядное ИЛИ	`Rd ← Rn | Src2`
1101	Сдвиги		
$I = 1$ или ($instr_{11:4} = 0$)	`MOV Rd, Src2`	Перемещение	`Rd ← Src2`
$I = 0$ и ($sh = 00$; $instr_{11:4} \neq 0$)	`LSL Rd, Rm, Rs/shamt5`	Логический сдвиг влево	`Rd ← Rm << Src2`

Табл. А.1. (окончание)

cmd	Название	Описание	Операция
$I = 0$ и ($sh = 01$)	`LSR Rd, Rm, Rs/shamt5`	Логический сдвиг вправо	`Rd ← Rm >> Src2`
$I = 1$ и ($sh = 10$)	`ASR Rd, Rm, Rs/shamt5`	Арифметический сдвиг вправо	`Rd ← Rm>>>Src2`
$I = 0$ и ($sh = 11$; $instr_{11:7,4} = 0$)	`RRX Rd, Rm, Rs/shamt5`	Циклический сдвиг вправо с расширением знака	`{Rd, C} ← {C, Rd}`
$I = 0$ и ($sh = 11$; $instr_{11:7} \neq 0$)	`ROR Rd, Rm, Rs/shamt5`	Циклический сдвиг вправо	`Rd ← Rm ror Src2`
1110	`BIC Rd, Rn, Src2`	Поразрядная очистка	`Rd ← Rn & ~Src2`
1111	`MVN Rd, Rn, Src2`	Поразрядное НЕ	`Rd ← ~Rn`

Команда NOP (нет операции) обычно кодируется как 0xE1A000, что эквивалентно MOV R0, R0.

A.1.1. Команды умножения

Команды умножения кодируются, как показано на **Рис. В.2**. В 3-битовом поле *cmd* задается тип умножения, как показано в **Табл. В.2**.

Рис. А.2. Кодирование команд умножения

Таблица А.2. Команды умножения

cmd	Название	Описание	Операция
000	`MUL Rd, Rn, Rm`	Умножение	`Rd ← Rn × Rm (младшие 32 бита)`
001	`MLA Rd, Rn, Rm, Ra`	Умножение с накоплением	`Rd ← (Rn × Rm)+Ra (младшие 32 бита)`
100	`UMULL Rd, Rn, Rm, Ra`	Длинное умножение без знака	`{Rd, Ra} ← Rn × Rm (все 64 бита, Rm/Rn без знака)`
101	`UMLAL Rd, Rn, Rm, Ra`	Длинное умножение с накоплением без знака	`{Rd, Ra} ← (Rn × Rm)+{Rd, Ra} (все 64 бита, Rm/Rn без знака)`
110	`SMULL Rd, Rn, Rm, Ra`	Длинное умножение со знаком	`{Rd, Ra} ← Rn × Rm (все 64 бита, Rm/Rn со знаком)`
111	`SMLAL Rd, Rn, Rm, Ra`	Длинное умножение с накоплением со знаком	`{Rd, Ra} ← (Rn × Rm)+{Rd, Ra} (все 64 бита, Rm/Rn со знаком)`

A.2. Команды доступа к памяти

Большинство команд доступа к памяти (LDR, STR, LDRB и STRB) оперируют словами или байтами, код операции в них *op* = 01. Дополнительные команды доступа к памяти, оперирующие полусловами или байтами со знаком, имеют код *op* = 00 и обладают меньшей гибкостью в части структуры операнда *Src2*: непосредственное смещение занимает только 8 бит, а регистровое не может быть сдвинуто. В командах LDRB и LDRH для заполнения слова производится дополнение нулями, а в командах LDRSB и LDRSH – расширение знакового бита. См. также режимы индексации, описанные в **разделе 6.3.6**.

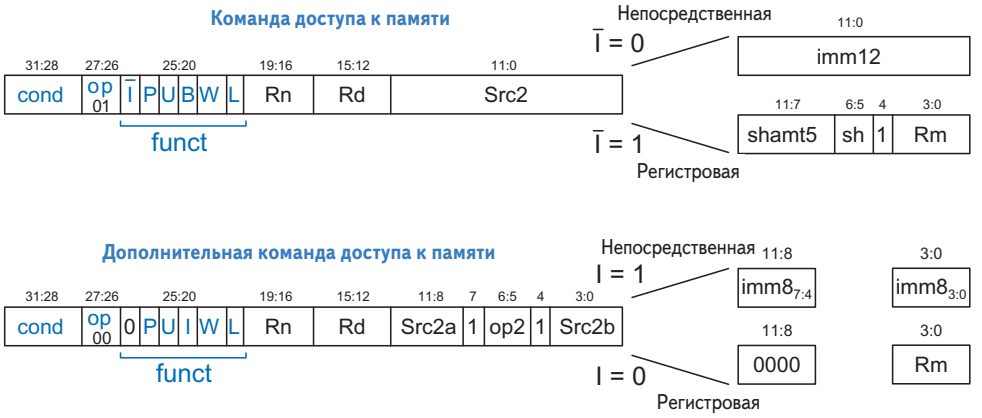


Рис. A.3. Кодирование команд доступа к памяти

Таблица A.3. Команды доступа к памяти

op	B	op2	L	Название	Описание	Операция
01	0	–	0	`STR Rd, [Rn, ±Src2]`	Сохранить регистр	`Mem[Adr]` ← `Rd`
01	0	–	1	`LDR Rd, [Rn, ±Src2]`	Загрузить регистр	`Rd` ← `Mem[Adr]`
01	1	–	0	`STRB Rd, [Rn, ±Src2]`	Сохранить байт	`Mem[Adr]` ← $Rd_{7:0}$
01	1	–	1	`LDRB Rd, [Rn, ±Src2]`	Загрузить байт	`Rd` ← $Mem[Adr]_{7:0}$
00	–	01	0	`STRH Rd, [Rn, ±Src2]`	Сохранить полуслово	`Mem[Adr]` ← $Rd_{15:0}$
00	–	01	1	`LDRH Rd, [Rn, ±Src2]`	Загрузить полуслово	`Rd` ← $Mem[Adr]_{15:0}$
00	–	10	1	`LDRSB Rd, [Rn, ±Src2]`	Загрузить байт со знаком	`Rd` ← $Mem[Adr]_{7:0}$
00	–	11	1	`LDRSH Rd, [Rn, ±Src2]`	Загрузить полуслово со знаком	`Rd` ← $Mem[Adr]_{15:0}$

А.3. Команды перехода

На Рис. А.4 показано, как кодируются команды перехода, а в Табл. А.4 – как они работают.

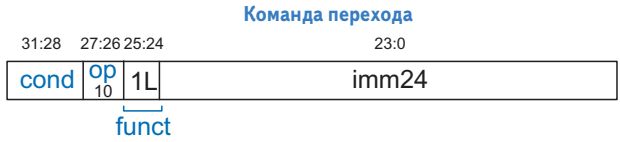


Рис. А.4. Кодирование команд перехода

Таблица А.4. Команды перехода

L	Название	Описание	Операция
0	`B label`	Перейти	`PC ← (PC+8)+imm24 << 2`
1	`BL label`	Перейти и связать	`LR ← (PC+8) – 4; PC ← (PC+8)+imm24 << 2`

А.4. Прочие команды

В набор команд ARMv4 входят также перечисленные ниже команды. Подробные сведения приведены в справочном руководстве по архитектуре ARM (ARM Architecture Reference Manual).

Команды	Описание	Назначение
`LDM, STM`	Загрузить/сохранить несколько	Сохранить или восстановить регистры при вызове функции
`SWP / SWPB`	Обменять (байт)	Атомарная загрузка и сохранения для синхронизации процессов
`LDRT, LDRBT, STRT, STRBT`	Загрузить/сохранить слово/байт с трансляцией	Разрешить операционной системе доступ к памяти в виртуальном адресном пространстве
`SWI`[1]	Программное прерывание	Возбудить исключение, часто используется для вызова операционной системы
`CDP, LDC, MCR, MRC, STC`	Доступ к сопроцессору	Взаимодействие с факультативным сопроцессором
`MRS, MSR`	Скопировать в регистр состояния или из него	Сохранить регистр состояния на время обработки исключения

[1] В наборе команд ARMv7 команда SWI переименована в SVC (вызов супервизора).

A.5. Флаги состояния

Флаги состояния изменяются командами обработки данных, в машинном коде которых бит $S = 1$. Чтобы любая команда, кроме `CMP`, `CMN`, `TEQ` и `TST`, поднимала бит S, в конец ее мнемонического обозначения нужно дописать знак «`S`». В **Табл. А.5** для каждой команды показано, какие флаги состояния она изменяет.

Таблица А.5. Команды, влияющие на флаги состояния

Тип	Команды	Флаги состояния
Сложение	`ADDS, ADCS`	*N, Z, C, V*
Вычитание	`SUBS, SBCS, RSBS, RSCS`	*N, Z, C, V*
Сравнение	`CMP, CMN`	*N, Z, C, V*
Сдвиги	`ASRS, LSLS, LSRS, RORS, RRXS`	*N, Z, C*
Поразрядные	`ANDS, ORRS, EORS, BICS`	*N, Z, C*
Тест	`TEQ, TST`	*N, Z, C*
Перемещение	`MOVS, MVNS`	*N, Z, C*
Умножение	`MULS, MLAS, SMLALS, SMULLS, UMLALS, UMULLS`	*N, Z*

Книги издательства «ДМК Пресс» можно заказать в торгово-издательском холдинге «Планета Альянс» наложенным платежом, выслав открытку или письмо по почтовому адресу: **115487, г. Москва, 2-й Нагатинский пр-д, д. 6А**.

При оформлении заказа следует указать адрес (полностью), по которому должны быть высланы книги; фамилию, имя и отчество получателя. Желательно также указать свой телефон и электронный адрес.

Эти книги вы можете заказать и в интернет-магазине: **www.a-planeta.ru**.

Оптовые закупки: **тел. +7 (499) 782-38-89**.

Электронный адрес: **books@alians-kniga.ru**.

Дэвид М. Харрис, Сара Л. Харрис

Цифровая схемотехника и архитектура компьютера
Дополнение по архитектуре ARM

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Научный редактор *Косолобов Д. А.*
Перевод с английского *Слинкин А. А.*
Корректор *Синяева Г. И.*
Верстка *Паранская Н. В.*
Дизайн обложки *Мовчан А. Г.*

Формат 70×100 1/16. Гарнитура «QuantAntiqua».
Печать офсетная. Усл. печ. л. 28,93.
Тираж 200 экз.

Веб-сайт издательства: www.dmkpress.com