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 MIPS

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's 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's ARM Microcontroller Development Kit (MDK-ARM)

Kiel's 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.

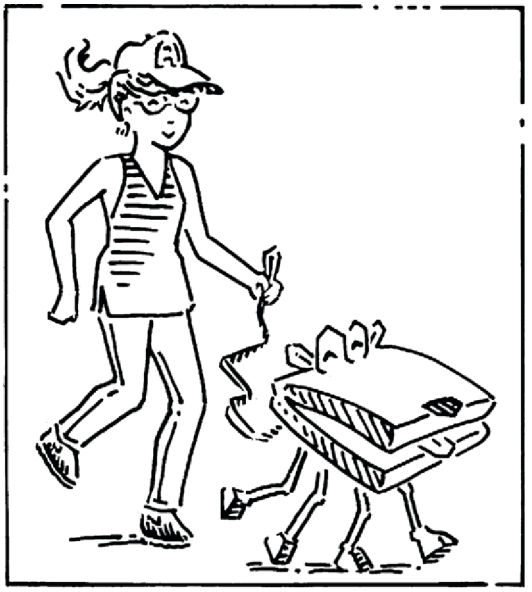
Мы хотели бы поблагодарить Мэтью Уоткинса (Matthew Watkins), который помог написать раздел о гетерогенных многопроцессорных системах в **главе 2**. Мы очень ценим работу Джошуа Васкеза (Joshua Vasquez), разработавшего код для Raspberry Pi из **главы 4**. Мы также благодарны Джозефу Спьюту (Josef Spjut)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

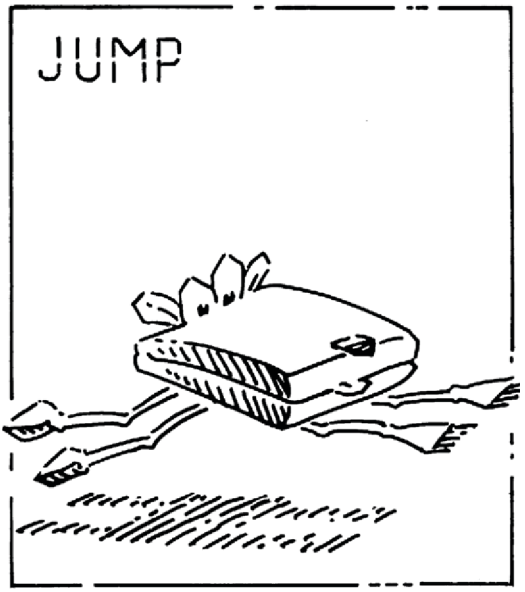
Огромный вклад в улучшение качества книги внесли многочисленные рецензенты, среди которых Бойанг Ванг (Boyang Wang), Джон Барр (John Barr), Джэк Брайнер (Jack V. Briner), Эндрю Браун (Andrew C. Brown), Карл Баумгартнер (Carl Baumgaertner), Утку Дирил (A. Utku Diril), Джим Френцель (Jim Frenzel), Джаэха 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.

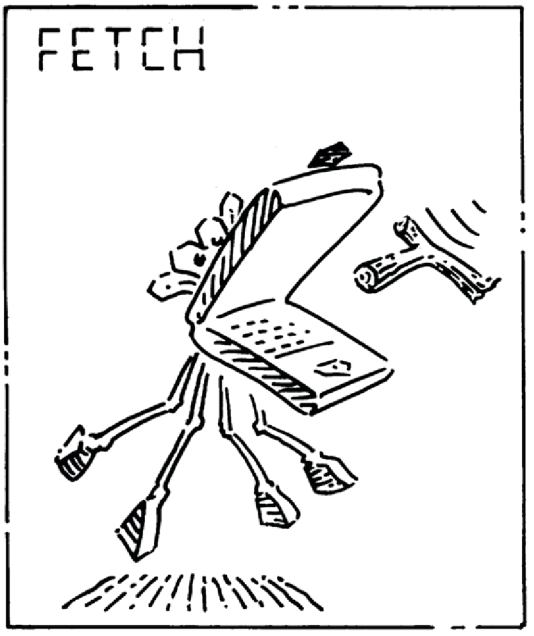
[1] Это относится лишь к ошибкам в исходном англоязычном издании. Ошибки в переводе присылайте на адрес dmkpress@gmail.com (хотя в этом случае издательство доллар не выдает). – *Прим. перевод.*

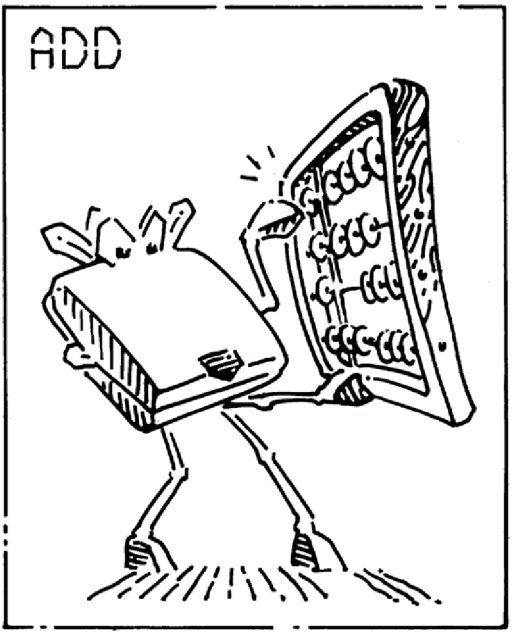
Мы также признательны нашим студентам из колледжа Harvey Mudd и UNLV, которые дали полезные отзывы на черновики этого учебника. Отдельного упоминания заслуживают Клинтон Барнс (Clinton Barnes), Мэтт Вайнер (Matt Weiner), Карл Уолш (Carl Walsh), Эндрю Картер (Andrew Carter), Кейси Шиллинг (Casey Schilling), 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.



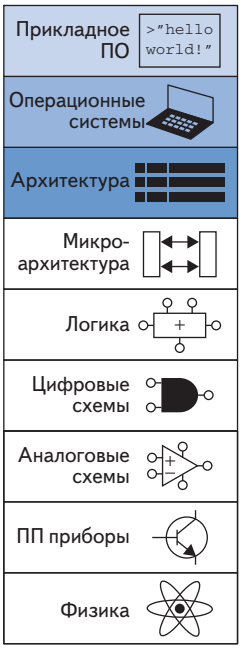

JUMP


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.

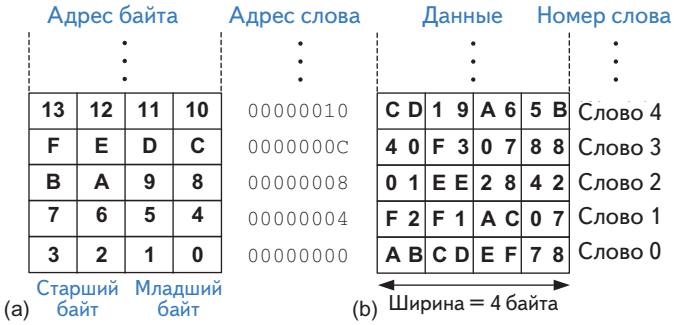


Рис. 1.1. Память в ARM с побайтовой адресацией: (a) адреса байтов, (b) данные

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.

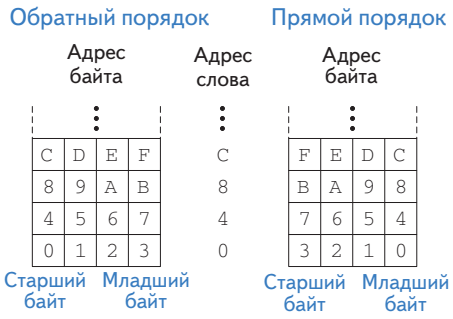


Рис. 1.2. Адресация данных с прямым и обратным порядком байтов

1.3. Программирование

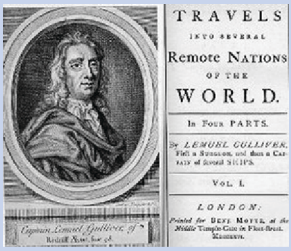
Языки типа C и Java называются языками программирования высокого уровня, поскольку они находятся на более высоком уровне абстракции, по сравнению с языком ассемблера. Во многих языках высокого уровня используются такие распространенные программные конструкции, как арифметические и логические операции, 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

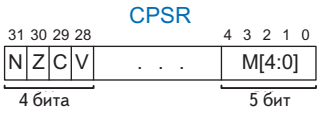


Рис. 1.6. Регистр текущего состояния программы (CPSR)

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).

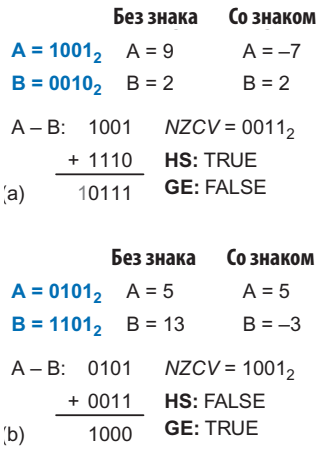


Рис. 1.7. Сравнение чисел со знаком и без знака: HS и GE

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*.

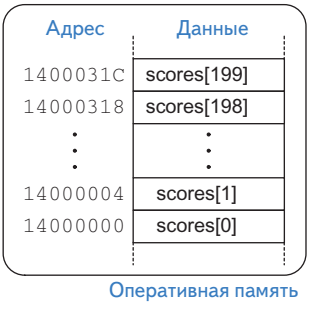


Рис. 1.8. Область памяти, содержащая массив `scores[200]` и начинающаяся с базового адреса 0x14000000

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.