Текст из pdf файла

Для получения дополнительных бесплатных электронных книг посетите http://ali-almukhtar.blogspot.com

Руководство разработчиков систем ARM по разработке и оптимизации системного программного обеспечения

О авторах Эндрю Н. Сloss Эндрю Сloss получил степень бакалавра в области компьютерных наук в Университете Хэферфордшира (Великобритания) в 1992 году и получил сертификат Chartered Engineer от Британского компьютерного общества (C.Eng, MBCS). Он получил большой опыт разработки широкого спектра приложений, работающих на процессоре ARM. Он разработал первые системы редактирования как для китайских, так и для египетских иероглифов, выполняющих на процессорах ARM2 и ARM3 для Emerald Publishing (Великобритания). В настоящее время он является техническим инженером продаж, консультирующим и поддерживающим компании, разрабатывающие новые продукты. Он работает в рамках организации продаж США и базируется в Лос-Гатосе, Калифорния. Он впервые программировал ARM в 1989 году и особенно интересуется алгоритмами и методами оптимизации. В то время как в Advanced Marine Systems он написал небольшие программные системы управления для использования на семействе микроконтроллеров Intel 8051. Крис Райт в последнее время провёл несколько лет в группе поддержки клиентов в ARM Inc., обучая и поддерживая партнерские компании, разрабатывающие новые продукты на основе ARM. Крис Райт в настоящее время является директором по поддержке клиентов в Ultimodule Inc. в Саннивеле, Калифорния. Джон Рейфилд Джон Рейфилд, независимый консультант, был бывшим вице-президентом по маркетингу в США в ARM. В этой должности он отвечал за определение стратегического направления маркетинга ARM в США и выявление возможностей для новых технологий для обслуживания ключевых сегментов рынка. До прихода в ARM Джон занимал несколько инженерных и управленческих ролей в области цифровой обработки сигналов, программного обеспечения, аппаратного обеспечения, ASIC и системного проектирования.

Руководство разработчиков систем ARMСценарий разработки и оптимизации системного программного обеспечения Эндрю Н. Сloss Доминик Саймс Крис Райт С участием Джона Рейфилда АМСТЕРДАМ • Бостон • Хайдельберг • ЛОНДОН НЬЮ-ЙОРК • Оксфорд • Париж • Сан-Диего Сан-Франциско • Сингапур • СИДНЕЙ • Токио Морган Кауфманн является отпечатка Elsevier

Старший редактор Денис Э.М. Пенроуз Менеджер издательских услуг Саймон Крумп Менеджер проекта Сара М. Редактор Hajduk Developmental Editor Belinda BreyerРедакционный помощник Летний блок-конверт Дик Ганус Cover Image Red Wing No.6 Charles Biederman Collection Walker Art Center, Minneapolis Подарок художника через программу покупки Фонда Форда, 1964 Техническая иллюстрация издательство ДартмутComposition Cepha Imaging, Ltd. Копиэтор Кен Деллапента Профредер Ян Кокер Индекс Феррера Индексирование Интерьерный принтер Печатная группа изготовления Maple-Vail Book Cover принтер Phoenix Color Morgan Kaufmann Publishers является отпечаткой Elsevier.500 Sansome Street, Suite 400, San Francisco, CA 94111 Эта книга напечатана на кислотной бумаге. © 2004 от Elsevier Inc. Все права защищены. Программы, примеры и приложения, представленные в этой книге и на веб-сайте издателя, включены для их учебной ценности. Издатель и авторы не предлагают никаких подразумеваемых или выраженных гарантий, включая, но не ограничиваясь подразумеваемые гарантии пригодности или коммерциализации для какой-либо конкретной цели, и не несут ответственности за любые потери или ущерб, возникающие в результате использования любой информации в этой книге, или любой ошибки или упущения в такой информации, или любого неправильного использования этих программ, процедур и приложений. Во всех случаях, когда Morgan Kaufmann Publishers знает о претензии, названия продуктов появляются в первом капитале или во всех капитальных буквах. Однако читатели должны обратиться в соответствующие компании для получения более полной информации о товарных знаках и регистрации. Ни одна часть этой публикации не может быть воспроизведена, храниться в системе поиска или передана в любой форме или любым способом электронно, механически, фотокопировано, сканировано или иным образомбез предварительного письменного разрешения издателя. Вы также можете заполнить свой запрос онлайн через домашнюю страницу Elsevier (http://elsevier.com) b y выбирая Помощь клиентов, а затем Объявление разрешений. Библиотека Конгресса Каталогирование в публикации данных Sloss, Эндрю Н. См. Содержит библиографические ссылки и индекс. ISBN 1-55860-874-5 (ал. бумага) 1. Компьютерное программное обеспечение Разработка. 2. Микропроцессоры RISC. 3. Компьютерная архитектура. I. Symes, Доминик II. Wright, Chris, 1953- III. Заголовок. QA76.76.D47S565 2004 005.1dc22 2004040366 ISBN: 1-55860-874-5 Для получения информации о всех публикациях Моргана Кауфмана посетите наш веб-сайт www.mkp.com.

Содержание о авторах ii Превод XI Глава 1 ARM Встроенные системы 3 1.1 Философия проектирования RISC 4 1.2 Философия проектирования ARM 5 1.3 Встроенное оборудование 6 1.4 Встроенное программное обеспечение 12 1.5 Резюме 15 Глава 2 ARM процессор Основы 19 2.1 Регистры 21 2.2 Регистр текущего состояния программы 22 2.3 Пиплин 29 2.4 Исключения, прерывания и векторная таблица 33 2.5 Основные расширения 34 2.6 Архитектура Ревизии 37 2.7 ARM процессорные семьи 38 2.8 Резюме 43 Глава 3 Введение в инструкцию ARM Набор 47 3.1 Инструкции по обработке данных 50 58 3.2 Инструкции 3.3 Загрузка-положение 3.4 Инструкции 60 73 3.5 Регистрные инструкции 3.6 Констанция в 787 787 ARM 795 Исполнение программного обеспечения 3.8

vi Содержание Глава 4 Введение в Инструкцию по большому пальцу Сет 87 4.1 Реестр по большому пальцу Использование 89 4.2 ARM-Рюбиковый взаимодействие 90 4.3 Другие отраслевые инструкции 92 4.4 Инструкции по обработке данных 93 4.5 Инструкции по единому регистру загрузка-магазин Инструкции 96 4.6 Многократные инструкции по загрузке-магазин 97 4.7 Инструкции по стеку 98 4.8 Программное обеспечение прерывать Инструкция 99 4.9 Резюме 100 Глава 5 Эффективное программирование 103 5.1 Обзор компиляторов и оптимизации данных 104 5.2 Основные типы C Регистрации 105 5.3 C Лопичные структуры 113 5.4 Разделение 120 5.5 Функции Звонки 122 5.6 Поискание 5.7 Структура 130 5.8 Битовые поля 133 5.9 Неуровневаемая собрания и конфигурация 153 153 5.11 Дивизионный код 5.12 ПО 5.12 ПО 6.12 ПО 6.12 6.12 6.12 6.12 6.13 6.13 6.13 6.13 6.14 6.14 6.14 6.14 6.14 6.14 6.14 6.14 6.14 6.14 6.14 6.14 6.14 6.15 6.14 6.14 6.14 6.15 6.14 6.15 6.15 6.15 6.14 6.14 6.15 6.15 6.14 6.14 6.14 6.15 6.14 6.14 6.15 6.15 6.15 6.15 6.15 6.15 6.15 6.15 6.15 6.15 6.15 6.15 6.15 6.15 6.15 6.15 6.15 6.15 6.15 6.15 6.15 6.15 6.15 6.15 6.15 6.15 6.15 6.15 6.15 6.15 6.15 6.15 6.15 6.15 6.15 6.15 6.15 6.15 6.15 6.15 6.15 6.15 6.15 6.15 6.15 6.15 6.15 6.15 6.15 6.15 6.15 6.15 6.15 6.15 6.15 6.15 6.15 6.15 6.15 6.15 6.15 6.15 6.15 6.15 6.15 6.15 6.15 6.15 6.15 6.15 6.15 6.15 6.15 6.15 6.15 6.15 6.15 6.15 6.15 6.15 6.15 6.15 6.15 6.15 6.15 7.1.

Содержание vii 6.9 Обработка несовместимых данных 201 6.10 Резюме 204 Глава 7 Оптимизированные примитивы 207 7.1 Двойное точное целое число Умножение 208 7.2 Цельное нормализация и подсчет ведущих нулей 212 7.3 Раздел 216 7.4 Квадраты Корни 238 7.5 Трансцендентные функции: логик, эксп, грех, кос 241 7.6 Индийская обратная и битная операции 248 7.7 Насыщенная и округлая арифметика 253 7.8 Случайное число 255 7.9 Резюме 256 Глава 8 Обработка цифровых сигналов 259 8.1 Представление цифрового сигнала 260 8.2 Введение в DSP на ARM 269 8.3 FIR 280 8.4 IIR фильтры 294 8.5 Резюме разграничения Формировать 303 8.6 Резюме 10.4 Глава 9 314 Исключение и перелом ручной фирмы 9.7 9.7 9.2 9.3 9.3 9.3 9.3 9.3 9.3 9.3 9.3 9.3 9.3 9.3 9.3 9.3 9.3 9.3 9.3 9.3 9.3 9.3 9.3 9.3

viii Содержание Глава 11 Встроенные операционные системы 381 11.1 Основные компоненты 381 11.2 Пример: Простая маленькая операционная система 383 11.3 Резюме 400 Глава 12 Кэш 403 12.1 Память и кеши памяти 404 12.2 Кэш архитектура 408 12.3 Кэш Политика 418 12.4 Копроцессор 15 и кеши 423 12.5 Кэш 423 12.6 Кэш блокировка 443 12.7 Кеши и программное обеспечение 456 12.8 Резюме 457 Глава 13 Части защиты памяти 461 13.1 Защищенные регионы 463 13.2 Интифицирование MPU, кеши и записывание кеши 465 13.3 демонстрация системы MPU 478 13.4 Резюме 487 14 Меморийное управление 491 Единицами памяти 14.1 14.1 14.2 14.2 14.2 14.2 14.2 14.2 15.2 15.3 15.3 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 14.2 14.2 14.2 14.2 14.2 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5 15.5

Содержание ix Глава 15 Будущее архитектуры Джон Рейфилд 549 15.1 Развитый DSP и SIMD Поддержка в ARMv6 550 15.2 Система и многопроцессорная поддержка Добавления к ARMv6 560 15.3 ARMv6 Implementations 563 15.4 Будущие технологии за пределами ARMv6 563 15.5 Резюме 566 Приложение A к ARM и указаниям на устройство с помощью устройства с помощью устройства с помощью устройства с помощью устройства с помощью устройства с помощью устройства с помощью устройства с помощью устройства с помощью устройства с помощью устройства с помощью устройства с помощью устройства с помощью устройства с помощью устройства с помощью устройства с помощью устройства с помощью устройства с помощью устройства с помощью устройства с помощью устройства с помощью устройства с помощью устройства с помощью устройства с помощью устройства с помощью устройства с помощью устройства с помощью устройства с помощью устройства с помощью устройства с помощью устройства с помощью устройства с помощью устройства с помощью устройства с помощью устройства с помощью устройства с помощью устройства с помощью устройства с помощью устройства с помощью устройства с помощью устройства с помощью устройства с помощью устройства с помощью устройства с помощью устройства с помощью устройства с устройства с помощью устройства с помощью устройства с устройства с помощью устройства с устройства с учетом устройства с учетом устройства с учетом устройства с учетом устройства с учетом устройства с учетом устройства с учетом устройства с учетом устройства с учетом устройства с учетом устройства с учетом устройства с учетом устройства с учетом устройства с устройства с учетом устройства с устройства с учетом устройства с устройства с учетом устройства с устройства с учетом устройства с устройства с учетом устройства с устройства с учетом устройства с устройства с учетом устройства с учетом устройства с устройства с устройства с учетом устройства с устройства с учетом устройства с устройства с учетом устройства с устройства с устройства

x Содержание Приложение E Предлагаемое чтение 667 E.1 ARM Ссылки 667 E.2 Алгоритм Ссылки 667 E.3 Управление памятью и архитектура кеша (Обзор и ссылка на аппаратные средства) 667 E.4 Ссылки на операционную систему 668 Индекс 669

Все чаще разработчики встроенных систем и конструкторы систем на чипе выбирают конкретные ядра микропроцессора и семейство инструментов, библиотек и непредназначенных компонентов для быстрого разработки новых продуктов на основе микропроцессоров. За последние 10 лет архитектура ARM стала самой распространенной 32-битарной архитектурой в мире, с более чем 2 миллиардами процессоров на основе ARM, поставленных на момент написания этой статьи. В мире развилось сообщество партнеров ARM и сторонних поставщиков среди компаний по разработке полупроводников и продуктов, включая инженеров-һадировщиков, системных дизайнеров и разработчиков программного обеспечения. Этот текст заполняет этот пробел.Наша цель состояла в том, чтобы описать работу ядра ARM с точки зрения разработчика продукта с четким акцентом на программное обеспечение. Поскольку мы написали эту книгу специально для инженеров, которые имеют опыт разработки встроенных систем, но не знакомы с архитектурой ARM, мы не предполагали, что у нас был бывший опыт ARM. Чтобы помочь нашим читателям стать продуктивными как можно быстрее, мы включили набор примеров программного обеспечения ARM, которые могут быть интегрированы в коммерческие продукты или использованы в качестве шаблонов для быстрого создания продуктивного программного обеспечения. Примеры пронумерованы так, чтобы читатели могли легко найти исходный код на веб-сайте издателя. Примеры также ценны для людей с опытом проектирования ARM, которые хотят максимально эффективно использовать встроенную систему на основе ARM. Организация книги Книга начинается с краткого обозначения философии проектирования процессора ARM и обсуждения того, как и почему она отличается от традиционной философии RISC. В первой главе также представлена простая встроенная система, основанная на процессоре ARM. Второй главе углубляется в аппаратное обеспечение, сосредоточившись на ядре процессора ARM и представлен обзор ядров ARM, находящихся на рынке. В объяснениях ключевых инструкций содержатся полные примеры, поэтому эти главы также служат учебным пособием по наборам инструкций.

xii Префаза и правила написания кода C, которые будут эффективно компилироваться на архитектуре ARM, и помогают определить, какой код должен быть оптимизирован. В главе 6 подробно описываются лучшие практики для написания и оптимизации кода сборки ARM, которые имеют решающее значение для улучшения производительности путем снижения потребления энергии системы и скорости часа. В нем представлены оптимизированные справочные реализации общепримитивных и более сложных математических операций для тех, кто хочет принять быстрый справочный подход. Они требуют возможности цифровой обработки сигналов (DSP), которые до недавнего времени были бы предоставлены отдельным процессором DSP. Однако теперь архитектура ARM предлагает более высокую пропускную способность памяти и более быстрые операции, позволяющие поддерживать эти приложения одним дизайном ARMcore. В главе 8 рассматривается, как максимизировать эффективность ARM для приложений цифровой обработки и как внедрить алгоритмы DSP. Глава 9 охватывает теорию и практику обработки исключений и прерываний на процессоре ARM с помощью множества подробных примеров. В главе также рассматриваются популярные отраслевые пакеты прошивки, доступные для ARM. Глава 11 демонстрирует внедрение встроенных операционных систем посредством примера операционной системы, которую мы разработали, называемой Simple Little Operating System. В главе 12 рассматриваются различные технологии кеша, которые окружают ядра ARM, демонстрируя рутины управления кешем на конкретных процессорах ARM с поддержкой кеша. В главе 13 обсуждается единица защиты памяти, а в главе 14 обсуждается единица управления памятью. Наконец, в главе 15 мы рассмотрим будущее архитектуры ARM, подчеркивая новые направления на наборе инструкций и новые технологии, которые ARM внедряет в ближайшие несколько лет. В приложениях даются подробные ссылки на наборы инструкций, сроки цикла и конкретные продукты ARM. Примеры в Интернете Как мы отметили ранее, мы создали обширный набор проверенных практических примеров для усиления концепций и методов. Эти книги можно найти на сайте издателя www.mkp.com/companies/1558608745.

Предыдущая статья XIII Признания Первый, конечно, наши жены Шау Чин Симес и Юлиан Ян и семьи, которые очень поддерживают и терпеливо терпели, что мы проводим большую часть нашего домашнего времени на этом проекте. Написание технической книги требует большого внимания к деталям, поэтому большое спасибо всем рецензентам, которые потратили время и усилия на чтение и предоставление отзывов. Рецензентами, работавшими с издателем во время процесса разработки, были Джим Турли (Silicon-Insider), Питер Малой (CodeSprite), Крис Ларсен, Питер Харрод (ARM, Ltd.), Гэри Томас (MLB Associates), Уэйн Вольф (Princeton University), Скотт Руннер (Qualcomm, Inc.), Ниалл Мерфи (PanelSoft) и Доминик Сьюитман (Algorithmics, Ltd.). Спасибо также Роду Кроуфорду, Эндрю Кэм-Минсу, Дэйву Флинну, Джейми Смиту, Уильяму Рису и Энне Руни за помощь в содействии. Спасибо группе стратегической поддержки ARM Хуард Хо, Джону Арчибальду, Мигелу Эчаваррии, Роберту Аллену и Иану Филд за чтение и предоставление быстрой местной обратной связи. Мы хотим поблагодарить Джона Рейфилда за инициирование этого проекта и внесение вклад в Глава 15. Мы также хотим поблагодарить Дэвида Браш за обзор рукописи и за то, что он позволил нам включить в эту книгу материал ARMv6.

Эта страница умышленно осталась пустой

1.1 Философия проектирования RISC 1.2 Философия проектирования ARM 1.2.1 Комплект инструкций для встроенных систем 1.3 Встроенная система аппаратное обеспечение 1.3.1 Технология автобуса ARM 1.3.2 Протокол автобуса AMBA1.3.3 Память 1.3.4 Периферические устройства 1.4 Встроенное программное обеспечение системы 1.4.1 Код инициирования (загрузки) 1.4.2 Операционная система1.4.3 Приложения 1.5 Резюме

Глава ARM EmbeddedSystems1 Ядро процессора ARM является ключевым компонентом многих успешных 32-битных встроенных систем. Вы, вероятно, владеете одним из них и даже не можете понять этого! Ядра ARM широко используются в мобильных телефонах, портативных организаторах и множестве других повседневных портативных потребительских устройств. К концу 2001 года по всему миру было отправлено более миллиарда процессоров ARM. Компания ARM основывает свой успех на простом и мощном оригинальном дизайне, который продолжает улучшаться и сегодня благодаря постоянным техническим инновациям. На самом деле ядро ARM не является одним ядром, а целым семейством конструкций, разделяющих схожие принципы проектирования и общий набор инструкций. Он обеспечивает до 120 Dhrystone MIPS 1 и известен своей высокой плотностью кода и низким энергопотреблением, что делает его идеальным для мобильных встроенных устройств. Затем мы представляем пример встроенного устройства и обсуждаем типичные аппаратные и программные технологии, которые окружают процессор ARM. 1.Dhrystone MIPS версия 2.1 - это небольшая программа для сравнения. 3

4 Глава 1 ARM Embedded Systems 1.1 Философия проектирования RISC Ядро ARM использует архитектуру RISC. RISC - это философия проектирования, направленная на предоставление простых, но мощных инструкций, которые выполняются в течение одного цикла с высокой скоростью часов. В результате, дизайн RISC ставит большие требования к компилятору. На рисунке 1.1 изображены эти основные различия. Философия RISC реализуется четырьмя основными правилами проектирования: 1.Инструкции РИСС процессоры имеют уменьшенное количество классов инструкций. Эти классы обеспечивают простые операции, которые каждый из них может выполнять в одном цикле. Компилятор или программист синтезирует сложные операции (например, операция деления) путем сочетания нескольких простых инструкций. Каждое указание имеет фиксированную длину, чтобы трубопровод мог получить будущие инструкции, прежде чем декодировать текущую инструкцию. В отличие от этого, в процессорах CISC инструкции часто имеют переменный размер и требуют много циклов для выполнения. В идеале трубопровод продвигается на один шаг на каждый цикл для максимальной пропускной способности.Инструкции могут быть декодированы на одном этапе трубопроводного процесса.Не требуется, чтобы инструкция выполнялась минипрограммой под названием микрокод, как на процессорах CISC. 3.Регистры РИСК машины имеют большой набор реестров общего назначения. Регистры выступают как быстрый локальный хранилище памяти для всех данных CISC RISC Компилятор Компилятор Процессор Генерация кода БольшаяComплексивность БольшаяКомплексивность Кодгенерация Рисунок 1.1 CISC против RISC. CISC подчеркивает аппаратную сложность. RISC подчеркивает сложность компилятора.

1.2 Операции обработки ARM Design Philosophy 5. В отличие от этого процессоры CISC имеют специальные реестры для конкретных целей. 4.Архитектура хранилища грузов. Отдельные инструкции по загрузке и хранению передают данные между банком регистра и внешней памятью. Доступ к памяти дорого, поэтому разделение доступа к памяти от обработки данных представляет собой преимущество, поскольку можно многократно использовать элементы данных, хранящиеся в банке регистра, без необходимости использования нескольких доступа к памяти. В отличие от этого, при разработке CISC процессоры данных могут действовать непосредственно на память. Эти правила проектирования позволяют RISC процессору быть проще, и, таким образом, ядро может работать на более высоких частотах часов. Однако в течение двух десятилетий различие между RISC и CISC стало туманным, поскольку процессоры CISC внедрили больше концепций RISC. 1.2 Философия проектирования ARM Существует ряд физических особенностей, которые обусловливали проектирование процессора ARM. Во-первых, портативные встроенные системы требуют некоторой формы питания батареей. Процессор ARM специально разработан для того, чтобы он был небольшим, чтобы уменьшить потребление энергии и расширить работу батареи, необходимых для таких приложений, как мобильные телефоны и персональные цифровые помощники (PDA). Высокая плотность кода полезна для приложений, которые имеют ограниченную бортовую память, например, мобильные телефоны и устройства массового хранения. Возможность использования недорогих устройств памяти обеспечивает значительную экономию.Еще одним важным условием является уменьшение площади дия, занятой встроенным процессором. Это, в свою очередь, снижает стоимость проектирования и производства, поскольку для конечного продукта требуется меньше дискретных чипов. ARM включила в процессор технологию аппаратного дебъга, чтобы инженеры программного обеспечения могли просматривать, что происходит, пока процессор выполняет код. С более высокой видимостью инженеры программного обеспечения могут быстрее решать проблемы, что напрямую влияет на время выхода на рынок и снижает общие затраты на разработку. В современных системах ключом является не скорость процессора, а эффективная производительность системы и расход энергии.

6 Глава 1 Встроенные системы ARM 1.2.1 Комплект инструкций для встроенных систем Комплект инструкций ARM отличается от чистого RISC определения несколькими способами, которые делают комплект инструкций ARM подходящим для встроенных приложений: ■Изполнение переменного цикла для определенных инструкций Не все инструкции ARM выполняются в одном цикле. Например, инструкции загрузки-хранилища-множества варьируются в количестве циклов выполнения в зависимости от количества переводимых реестров. Также повышается плотность кода, поскольку многочисленные перечисления реестра являются общими операциями в начале и конце функций. ■Встроенный сдвигатель ведет к более сложным инструкциям Встроенный сдвигатель ветвей является аппаратным компонентом, который предусматривает обработку одного из входящих реестра до того, как он будет использован инструкцией. Мы подробно объясняем эту функцию в главах 2, 3 и 4. ■ 16-битный набор инструкций по "Тумбу" ARM улучшил ядро процессора, добавив второй 16-битный набор инструкций под названием "Тумб", который позволяет на ядре ARM выполнять 16- или 32-битные инструкции. 16-битные инструкции повышают плотность кода примерно на 30% по сравнению с 32-битными инструкциями фиксированной длины. Эта функция улучшает производительность и плотность кода путем сокращения показаний к ветви. ■Улучшенные инструкцииУлучшенные инструкции для цифрового сигнального процессора (DSP) были добавлены в стандартный набор инструкций ARM для поддержки быстрых операций и насыщения 16-х16-битного мультипликатора. Эти инструкции позволяют более быстрым процессору ARM в некоторых случаях заменить традиционные комбинации процессора плюс DSP. Эти дополнительные функции сделали процессор ARM одним из наиболее часто используемых 32-битных встроенных процессоров. 1.3 Встроенная система аппаратное обеспечение Встроенные системы могут управлять многими различными устройствами, начиная от небольших датчиков, найденных на производственной линии, и заканчивая системами управления в режиме реального времени, используемыми на космической зонде НАСА. Все эти устройства используют сочетание программного обеспечения и аппаратных компонентов. Каждый компонент выбирается по эффективности и, если это применимо, предназначен для дальнейшего расширения и расширения.

1.3 Встроенная система Аппаратное обеспечение 7 процессора ARM АХБ арбитрПрерывный контроллер АХБ-АПБ мост Реальный час Серийные UARTsПамятьный контроллер АХБ-внешний мост Этернет счетчик/временщикиROMSRAMFLASHROM DRAMВнешний автобус Физический драйвер Этернет Консоль контроллеры ARM Периферические контроллеры Автобусная фигура 1.2 Пример встроенного устройства на базе ARM, микроконтроллер. Каждый ящик представляет собой функцию или функцию. линии, соединяющие ящики, представляют собой автобусы, перевозящие данные. Мы можем разделить устройство на четыре основных компонента оборудования: Процессор ARM включает ядро (двигатель выполнения, который обрабатывает инструкции и манипулирует данными), а также окружающие его компоненты, которые взаимодействуют с автобусом. Два обычно встречающихся контроллера - контроллер перерыва и контроллер памяти. ■Периферические устройства обеспечивают все возможности ввода и вывода вне чипа и отвечают за уникальность встроенного устройства. ■Abusis используется для связи между различными частями устройства.

8 Глава 1 ARM Embedded Systems 1.3.1 ARM Bus Technology Embedded systems используют разные технологии автобуса, чем те, которые предназначены для x86-х компьютеров. Этот тип технологии является внешним или вне чипа (т. е. автобус предназначен для механического анделектрического подключения к устройствам, находящимся вне чипа) и встроен в материнскую плату ПК. В отличие от этого, встроенные устройства используют на чипе автобус, который является внутренним для чипа и который позволяет различным периферийным устройствам быть взаимосвязаны с ядром ARM. Ядро процессора ARM - это логическое устройство, способное инициировать передачу данных с другим устройством через тот же автобус. Периферические устройства, как правило, являются логическими устройствами, способными отвечать только на запрос передачи от магистрального устройства автобуса. Второй уровень касается протокола - логических правил, регулирующих связь между процессором и периферийным устройством. В нем редко реализуются электрические характеристики автобуса, но он регулярно уточняет протокол автобуса. 1.3.2 Протокол автобуса AMBA Продвинутая микроконтроллерная архитектура автобуса (AMBA) была введена в 1996 году и широко принята в качестве архитектуры на чипе автобуса, используемой для процессоров ARM. Первыми автобусами AMBA, введенными были ARM System Bus (ASB) и ARM Peripheral Bus (APB). Позже ARM представила другой дизайн автобуса, названный ARM High Performance Bus ((AHB). Используя AMBA, дизайнеры периферийных устройств могут повторно использовать тот же дизайн на нескольких проектах. Поскольку существует большое количество периферийных устройств, разработанных с интерфейсом AMBA, дизайнеры аппаратных средств имеют широкий выбор проверенных и проверенных периферийных устройств для использования в устройстве. Этот интерфейс для разработчиков аппаратного обеспечения улучшает доступность и время выхода на рынок. AHB обеспечивает более высокую пропускную способность данных, чем ASB, потому что он основан на централизованной схеме мультиплексированного автобуса, а не на дизайне двунаправленного автобуса ASB. Это изменение позволяет автобусу AHB работать на более высоких скоростях часов и быть первым автобусом ARM, поддерживающим ширину 64 и 128 бит. ARM представила два варианта автобуса AHB: многослойный AHB и AHB-Lite. В отличие от оригинального AHB, который позволяет одному мастеру автобуса быть активным в любое время, многослойный AHB-автобус позволяет несколько активных мастеров автобуса. Этот автобус был разработан для конструкций, которые не требуют полных функций стандартного автобуса AHB. AHB и многослойный AHB поддерживают один и тот же протокол для хозяина и раба, но имеют разные взаимосвязи. Новые взаимосвязи в многослойном AHB хороши для систем с несколькими процессорами. Они позволяют выполнять операции параллельно и обеспечивают более высокие скорости пропуска.

1.3 Встроенное системное оборудование 9 Примерное устройство, показано на рисунке 1.2, имеет три автобуса: автобус AHB для периферийных устройств высокой производительности, автобус APB для более медленных периферийных устройств и третий автобус для внешних периферийных устройств, собственный на это устройство. Внешний автобус требует специального моста для подключения к автобусу AHB. 1.3.3 Память Встроенная система должна иметь какую-то форму памяти для хранения и исполнения кода. Если память должна работать вдвое быстрее, чтобы поддерживать желаемую полосу пропускания, то потребность в энергии памяти может быть выше. 1.3.3.1 Иерархия Все компьютерные системы имеют память, расположенную в какой-то форме иерархии. Внутри процессора есть вариант кеша (не показано на рисунке 1.2) для улучшения производительности памяти. Рисунок 1.3 показывает компромиссы памяти: самый быстрый кеш памяти находится физически ближе к ядре процессора ARM, а самая медленная вторичная память размещается дальше. Он используется для ускорения передачи данных между процессором и главной памятью. Каш обеспечивает общее увеличение производительности, но с потерей предсказуемого времени выполнения.

10 Глава 1 ARM Embedded Systems общая производительность системы, она не помогает в режиме реального времени отклика системы. Обратите внимание, что многие небольшие встроенные системы не требуют преимуществ производительности кеша. Указы по загрузке и хранению доступны к основной памяти, если только значения не были сохранены в кеше для быстрого доступа. В настоящее время вторичное хранилище может варьироваться от 600 МБ до 60 ГБ. 1.3.3.2 Ширина Памяти - это количество бит, которые память возвращает на каждый доступобычно 8, 16, 32 или 64 бит. Ширина памяти напрямую влияет на общий показатель производительности и стоимости. Если у вас есть нескейзная система с использованием 32-битных инструкций ARM и 16-битных чипов памяти, то процессору придется делать два загрузки памяти на инструкцию. Это, очевидно, уменьшает производительность системы, но преимущество в том, что 16-битная память дешевле. В отличие от этого, если ядро выполняет 16-битные указания на палец, оно достигнет лучших результатов с 16-битной памятью. Таким образом, использование указаний на палец с 16-битовыми устройствами памяти обеспечивает как улучшение производительности, так и снижение затрат. В этом разделе мы описываем некоторые из наиболее популярных устройств памяти, найденных в встроенных системах на базе ARM. Память для чтения только (ROM) является наименее гибкой из всех типов памяти, потому что она содержит изображение, которое постоянно устанавливается во время производства и не может быть перепрограммировано. Многие устройства также используют ROM для хранения загрузочного кода. Таблица 1.1 Привлечение инструкций из памяти. Размер инструкции 8-битная память 16-битная память 32-битная память ARM 32-битная 4 цикла 2 цикла 1 цикл Дюйм 16-битный 2 цикла 1 цикл 1 цикл

1.3 Встроенный аппаратный аппарат 11 Flash ROM можно писать и читать, но его запись медленна, поэтому его не следует использовать для хранения динамических данных. Удаление и запись флэш-ROM полностью управляются программным обеспечением без необходимости дополнительной аппаратной схемы, что снижает затраты на производство. Flash ROM стал самым популярным из типов памяти для чтения и в настоящее время используется в качестве альтернативы массовому или вторичному хранению. Динамическая память случайного доступа (DRAM) является наиболее часто используемой памятью для устройств. DRAM динамично, поэтому требуется обновление камер хранения и новый электронный заряд каждые несколько миллисекунд, поэтому перед использованием памяти необходимо установить контроллер DRAM. Время доступа для SRAM значительно короче, чем для эквивалентного DRAM, поскольку SRAM не требует паузы между доступом к данным. Из-за его более высокой стоимости он используется в основном для небольших высокоскоростных задач, таких как быстрая память и кеширование. SDRAM синхронизирует себя с процессорным автобусом, потому что он содержит часовые блоки. Внутренне данные извлекаются из клеток памяти, проводятся в трубопроводы и, наконец, выводятся на автобус в разрыве. Старый стиль DRAM не синхронен, поэтому не взрывается так эффективно, как SDRAM. 1.3.4 Периферические Встроенные системы, которые взаимодействуют с внешним миром, нуждаются в некоторой форме периферического устройства. Периферическое устройство выполняет входные и выходные функции для чипа путем подключения к другим устройствам или датчикам, которые не находятся на чипе. Периферальные устройства варьируются от простого серийного устройства связи до более сложного беспроводного устройства 802.11. Все периферии ARM имеют карту памяти. Интерфейс программирования представляет собой набор реестров с адресами памяти. Адрес этих реестров является компенсированным от конкретного периферийного базового адреса. Контролеры являются специализированными перифериями, которые реализуют более высокий уровень функциональности в рамках встроенной системы. В процессоре управления используются два типа контроллеров: контроллеры памяти и контроллеры перерывов. 1.3.4.1 Контролеры памяти Подключают к процессору различные типы памяти. Некоторые устройства памяти должны быть установлены программным обеспечением; например, при использовании DRAM, прежде всего необходимо установить сроки памяти и скорость обновления, прежде чем к ней можно получить доступ.

12 Глава 1 ARM Embedded Systems 1.3.4.2 Перерывные контроллеры Когда периферий или устройство требует внимания, оно поднимает перерыв процессора. Контролер перерыва обеспечивает программируемую управляющую политику, которая позволяет программному обеспечению определить, какой периферий или устройство может перерывать процессор в любое конкретное время, установив соответствующие биты в реестрах контроллеров перерыва. Он может быть запрограммирован для игнорирования или маскирования отдельного устройства или набора устройств. Управляющий перерывом определяет, какое устройство требует обслуживания, читая регистра битмапы устройства в контроллере перерыва. ВИК более мощный, чем стандартный контроллер перерыва, потому что он приоритетирует перерывы и упрощает определение того, какое устройство вызвало перерыв. После объединения приоритета и адреса обработки с каждым перерывом, VIC утверждает сигнал перерыва в ядро только в том случае, если приоритетность нового перерыва выше, чем текущий, выполняющий перерывный обработчик. В зависимости от типа VIC будет либо вызвать стандартный управляющий исключением прерывания, который может загружать адрес управляющего устройства из VIC, либо заставить ядро перепрыгнуть к управляющему устройству напрямую. Каждый компонент программного обеспечения в стеке использует более высокий уровень абстракции, чтобы отделить код от аппаратного устройства. Он устанавливает минимальные части доски перед передачей контроля операционной системе.

1.4 Встроенное программное обеспечение системы 13 Операционная система обеспечивает инфраструктуру для управления приложениями и управления аппаратными ресурсами системы. Многие встроенные системы не требуют полной операционной системы, а просто простого расписателя задач, которое будет управляться либо событием, либо опросом. Драйверы устройств являются третьей компонентой, показанной на рисунке 1.4. Они обеспечивают последовательный программный интерфейс для периферийных устройств на аппаратном устройстве.В конце концов, приложение выполняет одну из задач, необходимых для устройства.Например, мобильный телефон может иметь приложение-дневник.На одном устройстве могут быть несколько приложений, управляемых операционной системой.Софт. компоненты могут работать из ROM или RAM. Ром-код, который фиксируется на устройстве (например, код инициализации), называется прошивкой. 1.4.1 Код инициализации (загрузка) Код инициализации (или код загрузки) забирает процессор из состояния ресета в состояние, в котором может работать операционная система. В простой системе операционная система может быть заменена простой программист или дебгу-монитор. Мы можем группировать эти различные задачи на три этапа: начальная аппаратная конфигурация, диагностика и загрузка. Например, система памяти обычно требует реорганизации карты памяти, как показано в Примере 1.1. Диагностика часто встроена в код инициализации. Диагностический код тестирует систему, используя аппаратную цель для проверки, в состоянии ли цель работать. Этот тип тестирования важен для производства, поскольку он происходит после завершения разработки программного продукта. Сам процесс загрузки может быть сложным, если система должна загрузить разные операционные системы или разные версии одной и той же операционной системы. Загрузка изображения является последним этапом, но сначала вы должны загрузить изображение. Загрузка изображения включает в себя от копирования всей программы, включая код и данные в RAM, до простой копирования зоны данных, содержащей переменные, в RAM. После запуска система передает контроль, изменяя счетчик программы, чтобы указывать на начало изображения. Пример 1.1 Инициализация или организация памяти является важной частью кода инициализации, потому что многие операционные системы ожидают известного макета памяти, прежде чем они могут начать.

14 Глава 1 ARM Embedded Systems Before 0xffffffffff 0x00000000I/O Regs I/O Regs FAST SRAM FAST SRAMBoot ROM Boot ROMDRAM largecontiguousblockDRAM largecontiguousblockDRAM после рисунка 1.5 Remapping памяти. В системе ARM встроенная система обычно предусматривает ремейпинг памяти, поскольку он позволяет системе запускать код инициализации из ROM при включении. В разделе 2.4 мы подробно обсудим векторную таблицу. ■ 1.4.2 Операционная система Процесс инициализации готовит аппаратное обеспечение для операционной системы для управления. При управлении этими ресурсами операционная система может эффективно использоваться различными приложениями, работающими в среде операционной системы. Процессоры ARM поддерживают более 50 операционных систем. Мы можем разделить операционные системы на две основные категории: операционные системы в режиме реального времени (RTOS) и операционные системы платформы. RTOS обеспечивают гарантированные сроки отклика на события. Различные операционные системы имеют различный контроль за временем отклика системы.Трудные приложения в режиме реального времени требуют гарантированного ответа на работу вообще. Системы, работающие на RTOS, обычно не имеют вторичного хранилища. Платформа операционных систем требует единой единицы управления памятью для управления крупными, не в режиме реального времени приложениями и, как правило, имеют вторичное хранилище.

1.5 Сводка 15 Эти две категории операционных систем не являются взаимовыключающими: существуют операционные системы, которые используют ядро ARM с блоком управления памятью и имеют характеристики в режиме реального времени. ARM разработала набор процессорных ядров, которые специально ориентированы на каждую категорию. 1.4.3 Приложения Операционная система распределяет приложения с кодом, предназначенным для выполнения конкретной задачи.Приложение выполняет задачу обработки; операционная система контролирует окружающую среду. Встроенная система может иметь одно или несколько активных приложений, работающих одновременно. В каждом сегменте процессоры ARM можно найти в нескольких приложениях. Например, процессор ARM встречается в сетевых приложениях, таких как домашние шлюзы, модемы DSL для высокоскоростной интернет-коммуникации и беспроводная связь 802.11. Процессоры ARM также встречаются в устройствах массового хранения, таких как жесткие диски, и изобразительных продуктах, таких как чернильные принтеры, которые являются чувствительными к затратам и высоким объемам. Поскольку эти приложения имеют тенденцию к низкому объему и высокой стоимости, ARMhas решил не сосредоточиться на проектах на этих типах приложений. 1.5 Резюме ПУРНЫЙ РИСК направлен на высокую производительность, но ARM использует модифицированную философию RISC, которая также ориентирована на хорошую плотность кода и низкое потребление энергии. Встроенная система состоит из процессора ядра, окруженного кешами, памятью и перифериями. Система управляется программным обеспечением операционной системы, которое управляет задачами приложения. Ключевые моменты в философии проектирования RISC заключаются в улучшении производительности путем уменьшения сложности инструкций, ускорении обработки инструкций с использованием трубопровода, предоставлении большого реестра для хранения данных вблизи ядра и использовании архитектуры хранения грузов.

16 Глава 1 Встроенные системы ARM ■Улучшает плотность кода и производительность посредством условного выполнения инструкций. ■Включает улучшенные инструкции для выполнения функций типа цифровой обработки сигналов.Встроенная система включает следующие аппаратные компоненты: в чипах находятся процессоры ARM.Програмисты получают доступ к периферийным устройствам через регистры, отображаемые в памяти. Есть специальный тип периферийного устройства, называемый контроллером, который используется встроенными системами для конфигурации функций более высокого уровня, таких как память и прерывания. После конфигурации операционные системы могут быть загружены и выполнены. Операционные системы обеспечивают общую программирующую среду для использования аппаратных ресурсов и инфраструктуры. Драйверы устройств обеспечивают стандартный интерфейс для периферийных устройств. Приложение выполняет задачи, специфические для встроенной системы.

Эта страница умышленно осталась пустой

2.1 Регистры 2.2 Текущий статус программы Регистр 2.2.1 Режим процессора 2.2.2 Банковские регистры2.2.3 Состояние и наборы инструкций 2.2.4 Перерыв масок 2.2.5 Состояние флаги 2.2.6 Условное исполнение 2.3 Трубопровод 2.3.1 Трубопровод Исполнение характеристик 2.4 Исключения, перерывы и векторная таблица 2.5 Основные расширения 2.5.1 Кэш и тесно соединенная память 2.5.2 Управление памятью2.5.3 Копроцессоры 2.6 Архитектура Ревизии 2.6.1 Номенклатура2.6.2 Архитектура Эволюция 2.7 ARM процессорные семьи 2.7.1 ARM7 семья2.7.2 ARM9 семья 2.7.3 ARM10 семья 2.7.4 ARM112.7.5 Специализированные процессоры 2.8 Резюме семья

Глава ARM ProcessorFundamentals2 Глава 1 охватила встроенные системы с процессором ARM. В этой главе мы будем сосредоточиться на самом процессоре. Мы опишем модель программиста с точки зрения разработчика программного обеспечения процессора ARM, который покажет вам функции процессора и взаимодействие различных частей.Мы также рассмотрим основные расширения, которые составляют процессор ARM. Затем мы будем освещать пересмотры основной архитектуры ARM, описывая основные конвенции ARM для их идентификации и хронологические изменения в архитектуре наборов инструкций ARM. В заключительном разделе представлены архитектурные реализации, подразделяя их на конкретные семейства ядренных процессоров ARM. Программист может рассматривать ядро ARM как функциональные единицы, связанные с базами данных, как показано на рисунке 2.1, где стрелы представляют поток данных, линии представляют автобусы, а ящики представляют либо операционный блок, либо область хранения. На фиг. 2.1 показано выполнение Фон Нейманомских элементов ARMdata и инструкции разделяют один и тот же автобус. В отличие от Гарвардских программ, внедряемых в ARM, используются два различных автобуса. Дикодер инструкций переводит инструкции до их выполнения. Каждая выполняемая инструкция относится к определенному набору инструкций. Это означает, что он имеет два типа инструкций для передачи данных в процессор и из него: инструкции загрузки копировать данные из памяти в реестры в ядре, и наоборот хранилище 19

20 Глава 2 Основы процессора ARM Примечание данных Расширить Пишите Читайте r15 pcRd Rn RmA A B AccB NResultРегистр файла r0r15 ALUMAC Адресный реестр АдресИнкрементаторБаррель сдвигИнструкционный декодер Рисунок 2.1 ARM модели основного потока данных. инструкции копировать данные из реестров в память. Нет инструкций по обработке данных, которые непосредственно манипулируют данными в памяти. Таким образом, обработка данных осуществляется исключительно в реестрах. Данные помещаются в файл реестра в банке хранения, состоящем из 32-битных реестра. Знак расширяет аппаратные конверты, подписанные 8-битными и 16-битными номерами на 32-битные значения, поскольку они читаются из памяти и помещаются в реестр. Источник операндах читаются из файла реестра с использованием внутренних автобусов A и B, соответственно. ALU (арифметическая логическая единица) или MAC (множественно накапливаемая единица) берет регистрационные значения Rnand Rm из A и Bbuses и вычисляет результат. Указы по загрузке и хранению используют ALU для создания адреса, который будет помещен в адресный реестр и транслироваться на маршрутизатореAddress.

2.1 Регистры 21 Одной из важных особенностей ARM является то, что регистра RAlternatively может быть заранее обработана в смесителе баррелей, прежде чем он входит в ALU. Вместе смеситель баррелей и ALU могут рассчитывать широкий спектр выражений и адресов. После прохождения через функциональные единицы, результат в Rdis записывается обратно в файл реестра с использованием автобуса Результата. Для загрузки и хранения инструкций инкрементатор обновляет адресный реестр, прежде чем ядро читает или пишет следующее значение реестра из или в следующее место последовательной памяти. Процессор продолжает выполнять инструкции, пока исключение или прерывание не изменят нормальный поток выполнения. Теперь, когда у вас есть обзор ядра процессора, мы рассмотрим некоторые из ключевых компонентов процессора: реестры, текущий реестр состояния программы (cpsr) и трубопровод. Например, реестр 4 имеет этикетку r4. На рисунке 2.2 показаны активные реестры, доступные в режиме пользователязащищенный режим обычно r0 r1r2 r3 r4r5 r6 r7r8r9 r10 r11r12 r13 sp r14 lr r15 pc cpsr- На рисунке 2.2

22 Глава 2 Основы процессора ARM, используемые при выполнении приложений. Процессор может работать в семи различных режимах, которые мы вскоре введем. Все показаные реестры имеют размер 32 бит. Регистры данных видны программисту как r0tor15. Реестр r14 называется реестр ссылок (lr) и находится в центре, где адрес возвращения, когда он обращается к подпрограмме. ■Реестр r15 - счетчик программы (PC) и содержит адрес следующей инструкции, которую должен получить процессор. В зависимости от контекста, реестры r13 и r14 также могут быть использованы в качестве реестра общего назначения, что может быть особенно полезно, поскольку эти реестры банковствуются во время изменения процессового режима. Однако, когда процессор работает на любой форме операционной системы, использование r13 в качестве общего реестра опасно, поскольку операционные системы часто предполагают, что r13 всегда указывает на действительную раму стека. Однако есть инструкции, которые обращаются с r14 и r15 особым образом. Помимо 16 реестров данных, есть два реестра состояния программы: cpsr и spsr (текущие и сохраненные реестры состояния программы, соответственно). 2.2 Реестр текущего состояния программы Ядро ARM использует cpsr для мониторинга и контроля внутренних операций. cpsr является специальным 32-битным реестре и находится в файле реестра. В настоящее время в процессоре ARM используются дополнительные биты, которые могут быть использованы в процессоре, в процессоре ARM, в процессоре ARM и в процессоре ARM. Например, Jbit, который можно найти в поле флаги, доступен только на процессорах с поддержкой Jazelle, которые выполняют

2.2 Реестр текущего состояния программы 23 поля бит Функция состояния флагиПерерыв маски Режим процессора Дюйм-стат31 30 29 28 NZCV7654 0 IF T РежимФлаги управления расширением состояния Рисунок 2.3 Обычный регистр состояния программы (psr). 8-битные инструкции. Мы обсудим Jazelle в разделе 2.2.3. Вполне вероятно, что в будущем дизайне будут присвоены дополнительные биты для мониторинга и контроля новых функций. Каждый режим процессора является либо привилегированным, либо не привилегированным: привилегированный режим позволяет полный доступ к cpsr с чтением и написанием. В отличие от этого, режим непривилегированного режима позволяет только читать доступ к поле управления в cpsr, но все же позволяет читать-писывать доступ к знакам состояния. В общей сложности существует семь режимов процессора: шесть привилегированных режимов (отключение, быстрое запрос перерыва, запрос перерыва , супервайзер, система и неопределенный) и один режим непривилегированный (пользователь). Моды быстрого перерыва и перерыва запроса соответствуют двум уровням перерыва, доступным на процессоре ARM. Мод супервайзера - это режим, в котором процессор находится после ресета и, как правило, режим, в котором работает ядро операционной системы. Неопределенный режим используется, когда процессор сталкивается с инструкцией, которая не определена или не поддерживается реализацией. Пользовательский режим используется для программ и приложений. 2.2.2 Банковские реестры Рисунок 2.4 показывает все 37 реестра в файле реестра. Эти реестры называются банковскими реестрами и идентифицируются оттенком на диаграмме.

24 Глава 2 Основы процессора ARM Пользователь и система Быстрое запрос перерыва Перерыв запрос Надзорный запрос Неопределенный Abortr0 r1 r2 r3 r4 r5 r6 r7 r8 r9 r10 r11 r12 r13 sp r14 lr r15 pc cpsr-r8\_ fiqr9\_ fiqr10\_ fiq r11\_ fiq r12\_ fiqr13\_ fiq r14\_ fiq spsr\_ fiqr13\_ irq r14\_ irq spsr\_ irqr13\_ svcr14\_ svcr13\_ spundefr14\_undef spsr\_ abtr13\_ abtr14\_ abt spsr\_ abt полный реестр ARM набор реестра; например, режим заброшен реестр r\_ abt\_ r13\_ abt\_ r14\_ abt\_ rsp\_ abt\_ и Figure 2.4 Банковские реестры определенного режима обозначаются подчеркнутым символом, фиксированным после режима мнемонического или \_мода. Каждый режим процессора, кроме режима пользователя, может изменить режим, написав непосредственно в биты режима cpsr. Все режимы процессора, кроме режима системы, имеют набор связанных банковских реестра, которые являются подмножеством основных 16 реестра. Если вы меняете режим процессора, банковский реестр из нового режима заменит существующий реестр. Например, когда процессор находится в режиме запроса на перерыв, инструкции, которые вы выполняете, включают регистра доступа, названные r13 и r14. Однако эти реестры являются банковскими реестрами r13\_irq и r14\_irq. Регистраторы пользовательского режима r13\_usr и r14\_usr не влияют на инструкции, ссылающиеся на эти регистры. Программа все еще имеет нормальный доступ к другим регистрам r0tor12. Режим процессора может быть изменен программой, которая пишет непосредственно в cpsr (ядра процессора должна быть в привилегированном режиме) или аппаратным обеспечением, когда ядро отвечает на

2.2 Регистр текущего состояния программы 25 Пользовательский режим Перерыв запроса moder0 r1 r2 r3 r4 r5 r6 r7 r8 r9 r10 r11 r12 r13 sp r14 lr r15 pc cpsr-r13\_irqr14\_irq spsr\_irq Рисунок 2.5 Изменение режима при исключении. исключение или перерыв. Следующие исключения и прерывания вызывают изменение режима: перезагрузка, запрос прерывания, запрос быстрого прерывания, программное прерывание, отключение данных, префетч-отключение и неопределенное указание. На рисунке показано изменение ядра с режима пользователя на режим запроса прерывания, что происходит, когда происходит запрос на прерывание из-за внешнего устройства, который поднимает прерывание к ядру процессора. Это изменение приводит к тому, что регистра пользователя r13 и r14 будут зафиксированы. Регистры пользователя заменяются регистрами r13\_irq и r14\_irq соответственно. Примечание r14\_irq содержит адрес возврата и r13\_irq содержит указатель на стек для режима запроса перерыва. На рисунке 2.5 также показано новое реестр, появляющийся в режиме запроса перерыва: реестр состояния программы (spsr), который хранит предыдущий режим cpsr. Вы можете увидеть на диаграмме cpsr, который копируется в spsr\_irq. Для возвращения в режим пользователя используется специальная инструкция возвращения, которая приказывает ядро восстанавливать оригинальный cpsr из spsr\_irq и банка в пользовательских реестрах r13 и r14.

26 Глава 2 Основы процессора ARM Таблица 2.1 Режим процессора. Режим сокращение Привилегированный режим[4:0] Abort abt да 10111 Быстрое перерывное запрос фик. да 10001 Перерывное запрос irq да 10010 Supervisor svc да 10011 Система sys да 11111 Неопределенный и да 11011 Пользовательская usr no 10000 Еще одна важная особенность, которую следует отметить, заключается в том, что cpsr не копируется в ссср, когда изменение режима вынуждено из-за программы, записывающейся непосредственно в cpsr. Сбережение cpsr происходит только при возникновении исключения или перерыва. Начало в привилегированном режиме полезно, поскольку код инициализации может использовать полный доступ к cpsr для настройки стеков для каждого из других режимов. В последней колонке таблицы приведены битовые шаблоны, которые представляют каждый из режимов процессора в cpsr. 2.2.3 Состояние и наборы инструкций Состояние ядра определяет, какой набор инструкций выполняется. Подобным образом набор инструкций "Тумб" активен только тогда, когда процессор находится в состоянии "Тумб". После того, как он находится в состоянии "Тумб", процессор выполняет 16-битные инструкции "Тумб". Вы не можете смешивать последовательные инструкции "АРМ", "Тумб" и "Джазель". Тбит "Тумб" "Джазель" в cpsr отражает состояние процессора. Это происходит, когда на процессор применяется мощность. Когда бит T составляет 1, то процессор находится в состоянии Дюймс. Для изменения состояния ядро выполняет специализированную инструкцию по ветку. Jazelle выполняет 8-битовые инструкции и представляет собой гибридную смесь программного обеспечения и оборудования, предназначенную для ускорения выполнения байткодов Java. Для выполнения байткодов Java требуется технология Jazelle и специально модифицированная версия виртуальной машины Java. Важно отметить, что аппаратная часть Jazelle поддерживает только подмножество байткодов Java; остальные моделируются в программном обеспечении.

2.2 Реестр текущего состояния программы 27 Таблица 2.2 ARM и Фундальный набор инструкций. ARM (cpsr T =0) Фундальный (cpsr T =1) Размер инструкции 32-бит 16-битные основные инструкции 58 30 Условное выполнениеначально только ветвей инструкции Обработка данныхИнструкции доступ к смещателю баррелей и ALUОсобный смещатель баррелей и ALU инструкции Реестр состояния программы чтение-пись в привилегированном режиме без прямого доступа использование 15 регистра общего назначения Реестры 8 регистра общего назначения +pc +7Смотрите высокие регистра +pc a Раздел 2.2.6. Таблица 2.3 Jazelle инструкции набор функций Jazelle (cpsr T =0,J=1) Размер инструкции 8-битные инструкции Более 60% инструкций Java-Core доступны в аппаратном обеспечении; остальные коды реализуются в программном обеспечении. Набор инструкции Jazelle является закрытым и не открытое набор инструкции. В таблице 2.3 приведены функции набора инструкций Jazelle. 2.2.4 Перерыв масок Перерыв масок используются для предотвращения конкретных запросов перерыва от перерыва процессора. На основе процессора ARM имеются два уровня запросов перерыва (IRQ) и запрос быстрого перерыва (FIQ). Ибит маскирует IRQ при установке на бинарный 1, и аналогично Fbit маскирует FIQ при установке на бинарный 1. 2.2.5 Условия флаги Условия флаги обновляются путем сравнений и результатов операций ALU, которые указывают на суфикс инструкции. Например, если инструкция SUBS-отчёта приводит к регистральному значению нуля, то Zflag в cpsr устанавливается.

28 Глава 2 Основы процессора ARM Таблица 2.4 Знак знаки состояния. Название знака Назначенное при Q Насыщение результат вызывает переполнение и/или насыщение V oOVerflow результат вызывает подписанное переполнение C Перемещение результатом вызывает неподписанное carry Z Zero результат - нуль, часто используется для указания равенства N Отрицательный бит 31 результата является бинарным 1 При процессорных ядрах, которые включают расширения DSP, Qbit указывает, если в расширенной инструкции DSP произошло переполнение или насыщение. Флаг sticky в том смысле, что аппаратное обеспечение устанавливает только этот флаг. Чтобы очистить флаг, необходимо написать в cpsr напрямую. В процессорах с поддержкой Jazelle Jbit отражает состояние ядра; если он установлен, ядра находится в состоянии Jazelle. Jbit не используется в целом и доступен только на некоторых процессорных ядрах. Чтобы воспользоваться Jazelle, дополнительное программное обеспечение должно быть лицензировано как от ARM Limited, так и от Sun Microsystems. Большинство инструкций ARM могут выполняться условно по значению знаменных условий. Эти знаки расположены в самых значимых бит в cpsr. Эти бит используются для условного исполнения. На рисунке 2.6 показано типичное значение для cpsr как с расширениями DSP, так и Jazelle. В этой книге мы используем обозначение, которое представляет данные cpsr в более человечески читаемой форме. Когда бит является бинарным 1, мы используем большую букву; когда бит является бинарным 0, мы используем нижнюю казелетру. Для флагов условий большая буква показывает, что флаг установлен. Для прерывания большая буква показывает, что прерывание отключено. В примере cpsre показано на рисунке 2.6, Cflag является единственным набором флагов условий. Остальные флагов nzvq все чисты. Процессор находится в состоянии ARM, потому что ни один из Jazelle jor Thumb t битов не установлен. IRQ прерывания включены, а FIQ прерывания отключены. Наконец, вы nzCvq j iF SVC t31 30 29 28 27 24 00100 0765 4 0 0 1 0 10011 Рисунок 2.6 Пример: cpsr=nzCvqjiFt\_SVC.

2.3 Трубопровод 29 Таблица 2.5 Состояние мнемоники. Мнемонические Флаги Состояние EQ равно Z NE не равно z CS HS carry set/unsigned higher or the same C CC LO carry clear/unsigned lower c MI minus/negative N PL plus/positive or zero n VS overflow V VC no overflow v HI unsigned higher zC LS unsigned lower or the same Zorc GE signed greater or equal NVornv LT signed less than NvornV GT signed greater than NzV ornzv LE signed less or equal ZorNvornV AL always (unconditional) ignored можно увидеть по фигуре процессор находится в режиме надзора (SVC) поскольку режим[4:0] равен бинарному 10011.6 Условное исполнение Условное исполнение Контролируют исполнение или не будет выполнять инструкцию. Указания имеют условие, которое на основе основного атрибута выполнения на основе основного настройства флага будет определено, будет ли оно выполняться в соответствии с условием выполнения. Перед выполнением процессор сравнивает атрибут условия с знаками условия в cpsr. Если они совпадают, то инструкция выполняется; в противном случае инструкция игнорируется. Атрибут условия закрепляется за инструкцией мнемоникой, которая кодируется в инструкции. Когда условие мнемоническое отсутствует, дефолтным поведением является установка его на всегда (AL) выполнять. 2.3 Pipeline A - это механизм, который используется процессором RISC для выполнения инструкций. Один из способов увидеть трубопровод - это представить его как линию сборки автомобилей, на каждом этапе выполняется определенная задача по изготовлению транспортного средства.

30 Глава 2 ARM Processor Fundamentals Execute Decode Fetch Фигура 2.7 ARM7 Триэтапный трубопровод. В первом цикле ядро получает инструкцию ADD из памяти. Во втором цикле ядро получает инструкцию SUB и декодирует инструкцию ADD. В третьем цикле инструкции SUB и ADD перемещаются вдоль трубопровода. По мере увеличения длины трубопровода количество выполненной работы на каждом этапе сокращается, что позволяет процессору достигать более высокой частоты работы. Это, в свою очередь, увеличивает производительность. Увеличение длины трубопровода также означает, что между определенными этапами может быть зависимость от данных. Вы можете написать код для уменьшения этой зависимости с помощью графики инструкций (для получения дополнительной информации о графике инструкций см. Глава 6).

2.3 Трубопровод 31 Исполнение памяти записывать декодировать затягивать рисунок 2.9 ARM9 пятиступенчатый трубопровод. ARM9 добавляет стадию памяти и записывания, что позволяет ARM9 обрабатывать в среднем 1,1 Dhrystone MIPS на МГц увеличение пропускной способности инструкций примерно на 13% по сравнению с ARM7. Максимальная частота ядра, достижимая с помощью ARM9, также выше. ARM10 может обрабатывать в среднем 1,3 Dhrystone MIPS на МГц, примерно на 34% больше пропускной способности, чем ядро процессора ARM7, но снова при более высокой стоимости задержки. 2.3.1 Характеристики выполнения трубопровода ARM не обработал инструкцию, пока она не прошла полностью через этап выполнения. Например, трубопровод ARM7 (с тремя этапами) выполняет инструкцию только после получения четвертой инструкции. Инструкция MSR используется для включения IRQ-перерывов, которые происходят только после завершения MSR-инструкции этапа выполнения трубопровода. После того, как инструкция ADD вошла в этап выполнения трубопровода, включены прерывания IRQ. На рисунке 2.12 показано использование трубопровода и счетчика программы pc. На этапе выполнения pcalways указывают на адрес инструкции плюс 8 байтов. Это важно, когда PCS, используемый для расчета относительного офорса, является

32 Глава 2 Основы процессора ARM MSRExecute Decode Fetch cpsr IFt\_SVC cpsr IFt\_SVC cpsr IFt\_SVCMSR ADD MSR ADD ANDCycle 1 Цикл 2 Временный цикл 3 ADD И SUB Цикл 4 Рисунок 2.11 последовательность инструкций ARM. LDR NOP DCDExecute pc+ 8 (0x8000 + 8)Decode Fetch0x8000 LDR pc, [pc,#0] 0x8004 NOP 0x800 DCDAddress Figure 2.12 Пример: pc=адрес +8. архитектурная характеристика по всем трубопроводам. Обратите внимание, что, когда процессор находится в состоянии "Путь" (Pumb state), pcis - адрес инструкции плюс 4. Во-первых, выполнение инструкции по ветку или разветвление путем прямой модификации pccauses ядро ARM, чтобы выбросить его трубопровода. Во-вторых, ARM10 использует прогнозирование ветки, который уменьшает эффект от потока трубопровода, предсказывая возможные ветки и загружая новый адрес ветки до выполнения инструкции. Другие инструкции в трубопроводе будут оставлены, и процессор начнет заполнять трубопровод из соответствующего входа в векторной таблице.

2.4 Исключения, прерывания и векторная таблица 2.4 Исключения, прерывания и векторная таблица При возникновении исключения или прерывания процессор устанавливает пcto определенный адрес памяти. Вкладки в векторной таблице представляют собой инструкции, которые подключаются к конкретным рутинам, предназначенным для обработки конкретного исключения или прерывания. Адрес карты памяти 0x00000000 зарезервирован для векторной таблицы, набора 32-битных слов. На некоторых процессорах векторная таблица может быть выборным образом размещена на более высоком адресе в памяти (начиная с офсета 0xffff0000). Операционные системы, такие как Linux и встроенные продукты Microsoft, могут воспользоваться этой функцией. Каждый вклад в векторную таблицу содержит форму отраслевой инструкции, указывающей на начало конкретной рутины: ■Вектор восстановления - местонахождение первой инструкции, выполняемой процессором при нанесении питания. Эта инструкция передается коду инициализации. ■Неопределенный вектор инструкции используется, когда процессор не может расшифровать инструкцию. ■Программный вектор прерывания вызван, когда вы выполняете SWI-инструкцию. Инструкция SWI часто используется в качестве механизма для призыва к рутинной работе операционной системы. ■Вектор префетча происходит, когда процессор пытается получить инструкцию из адреса без правильных разрешений на доступ. Фактический аборт происходит на этапе декодирования. ■Вектор аборта данных похож на префетч-автор, но поднимается, когда инструкция пытается получить доступ к памяти данных без правильных разрешений на доступ. ■Вектор запроса прерывания используется внешним оборудованием для прерывания нормального потока исполнения процессора. Он может быть поднят только в том случае, если IRQ не маскированы в cpsr. Таблица 2.6 Векторовая таблица. Исключение/перерыв Короткий адрес Высокий адрес Ресет 0x00000000 0xffff0000 Неопределенная инструкция UNDEF 0x00000004 0xffff0004 ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО ПО П

34 Глава 2 Основы процессора ARM ■Вектар запроса быстрого перерыва похож на запрос перерыва, но предназначен для оборудования, требующего более быстрых времен отклика. Он может быть поднят только в том случае, если FIQ не скрыты в cpsr. Они улучшают производительность, управляют ресурсами и обеспечивают дополнительную функциональность и предназначены для обеспечения гибкости в обработке конкретных приложений. Каждое семейство ARM имеет различные расширения. Есть три аппаратных расширения, которые ARM заворачивает вокруг ядра: кеш и тесно соединенная память, управление памятью и интерфейс сопроцессора. 2.5.1 Кеш и тесно соединенная память. При кеше ядро процессора может работать в течение большинства времени, не дожидаясь данных из медленной внешней памяти. Большинство встроенных систем ARM используют одноуровневый кеш внутри процессора. Он объединяет как данные, так и инструкции в единый единый кеш, как показано на рисунке 2.13. Для простоты мы назвали логику клея, которая соединяет систему памяти с логикой и управлением автобуса AMBA.

2.5 Core Extensions 35 ARM Core Data TCMInstruction TCMLogic и Control On-chip AMBA busD+IDI DI Main memoryAMBA bus interface unit Рисунок 2.14 Упрощенная архитектура Гарварда с TCM. Напротив, вторая форма, прикрепленная к ядрам Гарвардского стиля, имеет отдельные кеши для данных и инструкций. Но для систем в режиме реального времени крайне важно, чтобы исполнение кода было детерминистичным. Время, необходимое для загрузки и хранения инструкций или данных, должно быть предсказуемо. TCM - это быстрый SRAM, расположенный рядом с ядром и гарантирует циклы часов, необходимые для получения инструкций или данных, критических для алгоритмов реального времени, требующих детерминистического поведения. Пример процессора с ТКМ показано на рисунке 2.14: Сочетанием обеих технологий процессоры ARM могут иметь как улучшенную производительность, так и предсказуемый ответ в режиме реального времени. Обычно необходимо иметь метод, который поможет организовать эти устройства и защитить систему от приложений, пытающихся сделать неуместный доступ к аппаратному обеспечению. Это достигается с помощью аппаратного обеспечения для управления памятью.Ядра ARM имеют три различных типа аппаратного обеспечения для управления памятьюне имеют расширений, обеспечивающих защиту, единица защиты памяти (MPU) обеспечивает ограниченную защиту, а единица управления памятью (MMU) обеспечивает полную защиту: ■Незащищенная память фиксируется и обеспечивает очень мало гибкости. Обычно он используется для небольших, простых встроенных систем, которые не требуют защиты от вредоносных приложений.

36 Глава 2 ARM Processor Fundamentals ARM Core Data TCMData cacheИнструкция кэш Инструкция ТСМЛОГИКА И управление На чипе AMBA busD+IDIDI DI Главная памятьАмба bus interface unit Рисунок 2.15 Упрощенная архитектура Гарвард с кэшами и ТКМ. ■MPU используют простую систему, которая использует ограниченное количество областей памяти. Эти регионы управляются наборами специальных реестров сопроцессора, и каждый регион определяется с конкретными разрешениями доступа. Этот тип управления памятью используется для систем, которые требуют защиты памяти, но не имеют сложной карты памяти. MMU использует набор таблиц перевода для обеспечения тщательного контроля над памятью. Эти таблицы хранятся в основной памяти и предоставляют виртуальную карту адресов с физического на физический, а также разрешения на доступ. MMU предназначены для более сложных платформенных операционных систем, которые поддерживают многозадачные работы. MMU объясняется в главе 14.2.5.3 Копроцессоры Копроцессоры могут быть присоединены к процессору ARM. Копроцессор расширяет функции обработки ядра путем расширения набора инструкций или предоставления регистраторов конфигурации. Более одного копроцессора можно добавить в ядро ARM через интерфейс копроцессора. К копроцессору можно получить доступ через группу специальных инструкций ARM, которые обеспечивают интерфейс типа клавиша для хранения грузов. Возьмем, например, копроцессор 15: TheARM-процессор использует реестры копроцессора 15 для управления кешем, ТКМ и управлением памятью.

2.6 Архитектура Ревизии 37 добавляются к стандартным инструкциям ARM, установленным для обработки операций вектора плавающей точкой (VFP). Если этап декодирования видит инструкцию сопроцессора, то она предлагает ее соответствующему сопроцессору. Но если сопроцессор не присутствует или не распознает инструкцию, то ARM принимает неопределенное инструкционное исключение, которое позволяет имитировать поведение сопроцессора в программном обеспечении. 2.6 Архитектурные пересмотры Каждая реализация процессора ARM выполняет специфическую архитектуру наборов инструкций (ISA), хотя в ревизии ISA может быть более одной реализация процессора. ISA развилась для усовершенствования требований встроенного рынка. Эта эволюция была тщательно управлена ARM, так что код, написанный для выполнения на более ранней архитектурной ревизии, также будет выполняться на более поздней ревизии архитектуры. Номенклатура идентифицирует отдельные процессоры и предоставляет базовую информацию о наборе функций. 2.6.1 Номенклатура ARM использует номенклатуру, показанную на рисунке 2.16 для описания процессора. Буквы и цифры после слова ARM указывают на характеристики процессора ARM{x}{y}{z}{T}{D}{M}{I}{E}{J}{F}{-S} xfamily ymemory management/protection unitzcache TThumb 16-bit decoder DJTAG debugMmultiplicator IEmbeddedICE macrocell Eenhanced instructions (assumes TDMI) JJazelleFvector floating-point unit Ssynthesizable version Figure 2.16 Номенклатура ARM.

38 Глава 2 Основы процессора ARM могут быть. Обратите внимание, что номенклатура не включает информацию об архитектурном пересмотре. Есть несколько дополнительных моментов, которые следует сделать о номенклатуре ARM: ■Все ядра ARM после ARM7TDMI включают функции TDMI, хотя они могут не включать эти буквы после этикета ARM. ■Семейство процессоров представляет собой группу процессоров, которые имеют одни и те же аппаратные характеристики. Например, ARM7TDMI, ARM740T и ARM720T все имеют одинаковые характеристики семейства и принадлежат к семейству ARM7. ■JTAG описывается стандартом IEEE 1149.1 Standard Test Access Port и архитектурой граничного сканирования. Это серийный протокол, используемый ARM для отправки и получения информации о дебоге между процессорным ядром и испытательным оборудованием. ■EmbeddedICE макроцель - это аппаратное обеспечение для дебога, встроенное в процессор, которое позволяет устанавливать точки перерыва и точки наблюдения. ■Синтезируемость означает, что ядро процессора поставляется в качестве исходного кода, который может быть составлен в форму, которую легко используют инструменты EDA. 2.6.2 Архитектура Эволюция Архитектура продолжает развиваться с момента внедрения первой реализации процессора ARM в 1985 году. В таблице 2.7 показаны значительные улучшения архитектуры от оригинальной архитектуры версии 1 до текущей архитектуры версии 6. Одним из наиболее значительных изменений в ISA было введение инструкции "Тюбль" в процессоре ARMv4T (processor ARM7TDMI). All относится к архитектуре ARMv4 и выше. 2.7 ARM Processor Families ARM разработала ряд процессоров, которые группируются в разные семьи в соответствии с используемым ядром. ARM8 был разработан, но вскоре был заменен.В таблице 2.9 представлена приблизительная сравнение атрибутов между ядрами ARM7, ARM9, ARM10 и ARM11.Приведенные цифры могут сильно варьироваться и напрямую зависят от типа и геометрии производственного процесса, что напрямую влияет на частоту (MHz) и потребление энергии (Вт).

2.7 ARM процессор семьи 39 Таблица 2.7 Ревизия история. Пример основного внедрения ISA улучшение ARMv1 ARM1 Первый ARM процессор 26-разрядный адресованный ARMv2 ARM2 32-разрядный мультипликатор32-разрядный сопроцессор поддержка ARMv2a ARM3 На чипе кеши Копроцессор 15 для управления кешем ARMv3 ARM6 и ARM7DI 32-разрядный адресс отдельных cpsr и spsr Новые режимы Неопределенные инструкции и поддержка MMU виртуальная память ARMv3M ARM7M Подписанные и неподписанные длинные инструкции ARMv4 ARM4 кешем Крепкий кешем Загрузочный магазин инструкции для подписанных и неподписанных полусловий Новый режим/Сервсистема СРЕЗИ для архитектурного управления пространством Недостаточное расширенное управление ARMv4 и ARM5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5T5 ARM продолжает расширять количество доступных семейств и различные вариации внутри каждой семьи. Вы можете найти другие процессоры, которые выполняют ISA ARM, такие как StrongARM и XScale. Эти процессоры уникальны для конкретной компании полупроводников, в данном случае Intel. Следующие подразделы описывают семейства ARM более подробно, начиная с семейства ARM7.

40 Глава 2 ARM Процессор Основы Таблица 2.8 Описание cpsr. Части Бит архитектуры Описание Режим 4:0 все процессорный режим T 5 ARMv4T Дюйм-статус I&F 7:6 все прерывающие маски J 24 ARMv5TEJ Jazelle состояние Q 27 ARMv5TE состояние флаг V 28 все состояние флаг C 29 все состояние флаг Z 30 все состояние флаг N 31 все состояние флаг Таблица 2.9 АРМААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААААА Ядро имеет трехступенчатый трубопровод и выполняет набор инструкций по архитектуре ARMv4T. ARM7TDMI был первым из новой линейки процессоров, введенных в 1995 году ARM. В настоящее время это очень популярный ядром и используется во многих 32-разрядных встроенных процессорах. Ядро процессора ARM7TDMI было лицензировано многими ведущими компаниями по полупроводникам по всему миру и является первым ядром, включающим набор инструкций Thumb, быстрое умножение инструкций и технологию дебгу EmbeddedICE.

2.7 ARM процессорные семьи 41 Таблица 2.10 Варианты процессора ARM. CPU core MMU/MPU cache Jazelle Thumb ISA Ea ARM7TDMI none none none none none none yes yes v4T no ARM7EJ-S none none none yes yes v5TEJ yes v5TEJ yes ARM720T MMU unified8K cache no yes v4T no ARM920T MMU separate16K /16K D+I cacheeno yes v4T no ARM922T MMU separate8K/8K D+I cacheeno yes v4T no ARM926EJ-S MMU separate cache yes v5TEJ v5TEJ v5TEJ v5TEJ v5TEJ v5TEJ v5TEJ v5TEJ v5TEJ v5TEJ v5TEJ v5TEJ v5TEJ v5TEJ v5TEJ v5TEJ v5TEJ v5TEJ v5TEJ v5TEJ v5TEJ v5TEJ v5TEJ v5J v5J v5J v5J v5J v5J v5J v5J v5J v5J v5J v5J v5J v5J v5J v5J v5J v5J v5J v5j v5j v5j v5j v5j v5v v5j v5j v5v v5j v5v v5j v5v v5j v5v v5v v5v v5v v5v v5v v5v v5v v5v v5v v5v v5v v5v v5v v5v v5v v5v v5v v5v v5v v5v v5v v5v v5v v5v v5v v5v v5v v5v v5v v5v v5v v5v v5v v5v v5v v5v v5v v5v v5v v5v v5v v5v v5v v5v v5v v5v v5v v5v v5v v5v v5v v5v v5v v ARM7TDMI-S имеет те же характеристики, что и стандартный ARM7TDMI, но также синтезируемый. ARM720T является самым гибким членом семейства ARM7, потому что он включает MMU. Наличие MMU означает, что ARM720T способен обрабатывать встроенные платформы Linux и Microsoft. Процессор также включает единый кеш 8K. Векторная таблица может быть перенесена на более высокий адрес, установив реестр 15 сопроцессора. Другой вариант - процессор ARM7EJ-S, который также синтезируется. ARM7EJ-S совершенно отличается, поскольку он включает в себя пятиступенчатый трубопровод и выполняет инструкции ARMv5TEJ. Эта версия ARM7 является единственной, которая обеспечивает как ускорение Java, так и улучшенные инструкции, но без каких-либо защитных данных памяти.

Глава 2 Основы процессора ARM 2.7.2 Семейство ARM9 Семейство ARM9 было объявлено в 1997 году. Благодаря своему пятиступенчатому трубопроводу, процессор ARM9 может работать на более высоких частотах часов, чем семья ARM7. Система памяти была переработана в соответствии с архитектурой Гарварда, которая разделяет данные и инструкции Ibuses. Первым процессором в семье ARM9 был ARM920T, который включает в себя отдельный D+Icache и MMU. Этот процессор может быть использован операционными системами, требующими поддержки виртуальной памяти. ARM922T является вариантом ARM920T, но с половиной размера кеша D+I. ARM940T включает меньшую D+Icache и MPU. ARM940T предназначен для приложений, которые не требуют платформы операционной системы. Это ядро является синтезируемой версией ARM9 core с расширениями E. Существуют два варианта: ARM946E-S и ARM966E-S. Оба выполняют инструкции архитектуры v5TE. Они также поддерживают опциональную встроенную макроцель следа (ETM), которая позволяет разработчику отслеживать инструкцию и выполнение данных в режиме реального времени на процессоре. Это важно при дебгуге приложений с критически важными сегментами времени. ARM946E-S включает TCM, кеш и MPU. Размеры TCM и кешей конфигурируются. Этот процессор предназначен для использования в встроенных приложениях управления, которые требуют детерминистического ответа в режиме реального времени. В отличие от ARM966E, он не имеет MPU и расширений кэш, но имеет конфигурируемые TCM. Последним ядром в линейке продуктов ARM9 является синтезируемый процессор ARM926EJ-S, объявленный в 2000 году. Он предназначен для использования в небольших портативных устройствах с поддержкой Java, таких как телефоны 3G и персональные цифровые помощники (PDA). ARM926EJ-S является первым ядром ARM-процессора, включающим технологию Jazelle, которая ускоряет выполнение байткода Java. Он имеет MMU, конфигурируемые TCM и D+Icaches с нулевым или не нулевым состоянием ожидания. 2.7.3 ARM10 Family ARM10, объявленный в 1999 году, был разработан для производительности. Он расширяет трубопровод ARM9 до шести этапов. Он также поддерживает опциональный векторный плавучий пункт (VFP), который добавляет седьмую стадию трубопровода ARM10. VFP значительно увеличивает производительность плавучей точки и соответствует стандарту IEEE 754.1985 плавучей точки. ARM1020E является первым процессором, использующим ядро ARM10E. Как и ARM9E, он включает улучшенные инструкции E. Он имеет отдельные 32K D+Icaches, опциональный векторный блок плавающей точки и MMU. ARM1020E также имеет двойный 64-битный автобусный интерфейс для повышенной производительности. ARM1026EJ-S очень похож на ARM926EJ-S, но с MPU и MMU. Этот процессор имеет производительность ARM10 с гибкостью ARM926EJ-S.

2.8 Резюме 43 2.7.4 ARM11 Семейство ARM11S ARM1136J-S, объявленное в 2003 году, было разработано для высокопроизводительных и энергоэффективных приложений. ARM1136J-S был первым процессором, который выполнял инструкции по архитектуре ARMv6. В инструкции ARMv6 включены расширения для однополых инструкций для обработки данных (SIMD), специально разработанные для повышения производительности обработки видео. ARM1136JF-S является ARM1136J-S с добавлением векторной блоки плавающей точки для операций с быстрыми плавающими точками. 2.7.5 Специализированные процессоры StrongARM первоначально были разработаны совместно с Digital Semiconductor и теперь лицензированы исключительно Intel Corporation. Он был популярным для PDA и приложений, требующих производительности с низким энергопотреблением. Это архитектура Гарварда с отдельными кешами D + I. StrongARM был первым высокопроизводительным процессором ARM, который включал в себя пятиступенчатый трубопровод, но он не поддерживает набор инструкций Thumb. Intels XScale является продолжением StrongARM и предлагает резкое повышение производительности. В момент написания XScale был цитирован как способный работать до 1 ГГц. XScale выполняет архитектуру v5TE инструкции. Это архитектура Гарварда и похожа на StrongARM, так как она также включает MMU. SC100 находится на другом конце спектра производительности. Он разработан специально для низкоэнергетических приложений безопасности. SC100 является первым SecurCore и основан на ARM7TDMI ядром с MPU. В этой главе мы сосредоточились на аппаратных основах фактического процессора ARM.Процессор ARM может быть абстрагирован в восемь компонентов: ALU, barrel shifter, MAC, реестрный файл, инструкционный декодер, адресный реестр, инкрементер и расширенный знак. ARM имеет три набора инструкций: ARM, Thumb и Jazelle. В файле реестра 37 реестра, но в любой момент времени доступно только 17 или 18 реестра; остальные хранятся в банке в зависимости от режима процессора. Состояние определяет, какой набор инструкций выполняется. Процессор ARM включает ядро плюс окружающие его компоненты, которые взаимодействуют с автобусом.

44 Глава 2 Основы процессора ARM ■Управление памятью используется для организации памяти и защиты системных ресурсов. ■Копроцессоры используются для расширения наборов инструкций и функциональности. Копроцессор 15 контролирует кеширование, ТКМ и управление памятью. Процессор ARM представляет собой реализацию специфической архитектуры наборов инструкций (ISA). ISA постоянно улучшается с первого дизайна процессора ARM. Процессоры группируются в семейства реализаций (ARM7, ARM9, ARM10 и ARM11) с аналогичными характеристиками.

Эта страница умышленно осталась пустой

3.1 Инструкции по обработке данных 3.1.1 Перемещение Инструкции 3.1.2 Перемещатель баррелей3.1.3 Арифметические инструкции 3.1.4 Использование перемещателя баррелей с арифметическими инструкциями 3.1.5 Логические инструкции 3.1.5 Инструкции 3.1.6 Сравнение Инструкции 3.1.7 Многократные инструкции 3.2 Инструкции отрасль 3.3 Инструкции по загрузке-магазина 3.3.1 Единорегистрированная регистрация 3.3.2 Единорегистрированная регистрация загрузки-магазина обращение с режимами3.3.3.3 Многократная регистрация загрузки 3.3.4 Срубка Инструкция 3.4 Процессорный регистрационный инструкция 3.1.5 Инструкция по состоянию программы 3.5.1 Инструкция по совместному процессору 3.5.2 Инструкция по совместному процессору 15 Инструкция по синтаксису 3.6 Константы по загрузке 3.7 ARMv5 ТипЭ Расширения 3.7.1 Счет инструкции по загрузке 3.7.23.3 Армв3.3 3.8 Условия выполнения 3.8 См.

Глава Введение в систему инструкций ARM3 Введение в систему инструкций ARM является основной главой, поскольку представленная здесь информация используется в течение всей остальной части книги. В этой главе представлены наиболее распространенные и полезные инструкции ARM и основные принципы процессора ARM, которые были рассмотрены в последней главе. В главе 4 представлен набор инструкций Thumb, а в приложении A представлен полный описание всех инструкций ARM. Различные архитектурные пересмотры ARM поддерживают разные инструкции. Однако новые пересмотры обычно добавляют инструкции и остаются обратно совместимыми. Код, который вы пишете для архитектуры ARMv4T, должен выполняться на процессоре ARMv5TE. В таблице 3.1 представлен полный список инструкций ARM, доступных в архитектуре набора инструкций ARMv5E (ISA). Эта ISA включает в себя все основные инструкции ARM, а также некоторые новейшие функции на наборе инструкций ARM. В колонке ARM ISA перечислены пересмотры ISA, в которой была введена инструкция. Некоторые инструкции расширили функциональность в более поздних архивах; например, инструкция CDP имеет вариант ARMv5 под названием CDP2. Аналогичным образом, такие инструкции, как LDR, имеют дополнения ARMv5, но не требуют новой или расширенной мнемоники.

48 Глава 3 Введение в ARM Инструкционный набор Таблица 3.1 Сатрудная инструкция ARM набор инструкций. Mnemonics ARM ISA Описание ADC v1 добавлять два 32-битных значения и переносить ADD v1 добавлять два 32-битных значения И v1 логический битовый путь И двух 32-битных значений B v1 ветви относитель +/−32 MB BIC v1 логический бит чистый (И НЕ) двух 32-битных значений BKPT v5 прорывной точке инструкции BL v1 относительный ветв с ссылкой BLX v5 ветв с ссылкой и обменом BX v4T ветв с 32-битной CDP CDP2 v2 v5 интегральный процессор обработки данных операции CLZ v5 сопоставить два 32-битных нуля CMN v1 ссопоставленные 32-битные значения CMP v1 ссопоставленные значения V1 ссопоставленные значения V1 ссопоставленные значения V1 ссопоставленные значения V1 ссопоставленные значения V1 ссососоединенные знания V1 с 32-битные значения 32-битные значения V1 ссопоставленные значения V2 ссопоставленные значения V2 ссоединенные битовые значения LB2 v2 v2 v2 v2 v2 v2 v2 v2 v2 v2 v2 v2 v2 v2 v2 v2 v2 v2 v2 v2 v3 v3 v3 v3 v3 v3 v3 v3 v3 v3 v3 v3 v3 v3 v3 v3 v3 v3 v3 v3 v3 v3 v3 v3 v3 v3 v3 v3 v3 v3 v3 v3 v3 v3 v3 v3 v3 v3 v3 v3 v3 v3 v3 v3 v3 v3 v3 v3 v3 v3 v3 v3 v3 v3 v3 v3 v3 v3 v3 v3 v3 v3 v3 v3 v3 v3 v3 v3 v3 v3 v3 v3 v3 v3 v3 v3 v3

Глава 3 Введение в набор инструкций по ARM 49 Таблица 3.1 набор инструкций по ARM. (Продолжение) Mnemonics ARM ISA Описание SMULxy v5E подписанное множественное указание (16 ×16=32 бита) SMULWy v5E подписанное множественное указание ((32 ×16)/greatermuch16=32 бита) STC STC2 v2 v5 хранить в памяти одно или несколько 32-битных значений от сопроцессора STM v1 множественные 32-битные регистраторы хранить в памяти STR v1 v4 v5E хранить регистрацию в виртуальный адрес в памяти SUB v1 вычитать два 32-битных значения SWI v1 программное обеспечение прерывает SWP v2a обмен словом/байтом в памяти с регистратором, без прерыва TEQ v1 тест на равенство двух 32-битных значений TST v1 тест для битов в 32-битном значении UMLAL v3M неподписанный накопившийся в 32-битный v32 ×642 +642 длинный UML3 ×642 = 642 длинный UML3 = 642 Мы представим шестидесятые числа с предварительным 0x и двоичные числа с предварительным 0b. Примеры следуют следующему формату: PRE<предварительные условия> <инструкция/s> POST<последующие условия> В предварительных и последующих условиях память обозначается как mem<data \_size>[адрес]. Это относится к битам памяти data\_size, начинающимся с данного байт-адреса. Например, mem32[1024] - 32-битный размер, начинающийся с адреса 1 КБ. Инструкции ARM обрабатывают данные, хранящиеся в реестрах, и получают доступ только к памяти с указаниями на загрузку и хранение. Инструкции ARM обычно принимают два или три операнда. Например, инструкция ADD ниже добавляет два значения, хранящихся в реестрах r1 и r2 ((источник реестра). Инструкция СинтаксисДестинационный регистр (Rd) Sourceregister 1 (Rn) Sourceregister 2 (Rm) ADD r3, r1, r2 r3 r1 r2 В следующих разделах мы изучаем функцию и синтаксис инструкций ARM по классу инструкцийинструкции по обработке данных, инструкции по отраслям,

50 Глава 3 Введение в инструкцию ARM Набор инструкций для хранения нагрузки, инструкции для прерывания программного обеспечения и инструкции по регистру состояния программы. 3.1 Инструкции по обработке данных Инструкции по обработке данных манипулируют данными в реестрах. Большинство инструкций по обработке данных могут обрабатывать один из своих операторов с помощью смещения барреля. Если вы используете Ssuffix на инструкции по обработке данных, то он обновляет флаги в cpsr. Перемещение и логические операции обновляют флаг переноса C, отрицательный флаг N и нулевой флаг Z. Флаг переноса устанавливается из результата смещения барреля, когда последний бит переносится. Zflag устанавливается, если результат 0 . 3.1.1 Переместить инструкции Переместить - это самая простая инструкция ARM. Она копирует N в реестр назначения Rd, где Nis реестр или непосредственное значение. Данное руководство полезно для установки начальных значений и передачи данных между реестрами. Синтаксис: <инструкция>{<cond>}{S} Rd, N MOV Переместить 32-битный значение в реестр Rd=N MVN переместить НОТ 32-битного значения в реестр Rd=∼ N Таблица 3.3, представленная в разделе 3.1.2, дает полный описание допустимых значений для второго операнда N для всех инструкций по обработке данных. Обычно это реестр Rmor постоянная, предшествующая #. Пример 3.1В этом примере показано простое руководство к движению.

3.1 Инструкции по обработке данных 51 3.1.2 Перемещатель баррелей В примере 3.1 мы показали инструкцию MOV, где Nis является простой регистром. Но N может быть не просто регистра или непосредственное значение; это также может быть регистра Rm, который был заранее обработан сместителем барреля до использования инструкцией по обработке данных. Это изменение увеличивает мощность и гибкость многих операций обработки данных. Существуют инструкции по обработке данных, которые не используют смещение барреля, например, инструкции по MUL (множение), CLZ (счет ведущих нулей) и QADD (подписанный насыщенный 32-битный добавление). Это особенно полезно для загрузки констант в реестр и достижения быстрых умножений или деления на мощность 2. Нет предварительной обработки ПреобработкаRn Rm Barrel Shifter Результат N RdАрифметическая логическая единица Рисунок 3.1 Barrel Shifter и ALU. Для иллюстрации Barrel Shifter мы возьмем пример из рисука 3.1 и добавим операцию смещения к примеру инструкции движения. Регистр Rnntrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrr Инструкция MOV копирует результат оператора смена в реестр Nd.NОтражает результат операции LSL, описанной в таблице 3.2.

52 Глава 3 Введение в наборы инструкций ARM MOV r7, r5, LSL #2; пусть r7 = r5\*4 = (r5<<2) POSTr 5=5 r7 = 20 Пример умножает реестр r5 на четыре, а затем помещает результат в реестр r7. Например, содержимое бита 0 переносится на бит 1. Бит 0 очищается. Cflag обновляется с последним битом, перенесенным из реестра. Это бит (32 −y) из первоначального значения, где yis сумма переноса. Если yis больше, чем одна, то смена по позициям равно сменам по одной позиции выполненным ytimes. Таблица 3.2 Описание Мнемонический Описание Результат смена смена объем y LSL логический смена левый xLSL y x/lessmuchy #031 или Rs LSR логический смена права xLSR y (неподписанный) xLSR y (неподписанный) xx /greatermuchy #132 или Rs ASR арифметический смена правой xASRags y (подписанный) xx /greatermuchy #132 или Rs ROR вращаются xROR y ((неподписанный) xROR y ((xx /greatermuchy) ((xmuch\_lessmuch\_32y) #100000000 или Rs RRX вращаются правой расширенный xRRX (cfxmuch\_greatermuch\_031) (((неподписанный) 0x\_rex\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x\_x

3.1 Инструкции по обработке данных 53 Таблица 3.3 Синтаксис работы с смены барреля для инструкций по обработке данных. Синтаксис операций смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены смены с Это умножает регистра r1с значением 21. Как вы можете видеть, Cflag обновляется в cpsr, потому что Ssuffix присутствует в инструкции mnemonic. PREcpsr = nzcvqiFt\_USER r0 = 0x00000000 r1 = 0x80000004 MOVS r0, r1, LSL #1 POSTcpsr = nzCvqiFt\_USERr0 = 0x00000008 r1 = 0x80000004 ■ В таблице 3.3 перечислено синтаксис для различных операций смещения баррелей, доступных на инструкции по обработке данных. Второй операнд N может быть непосредственной постоянной, предшествующей #, регистральным значением Rm, или значением Rm, обработанным смесью. 3.1.3 Арифметические инструкции Арифметические инструкции осуществляют добавление и вычитание 32-битных подписанных и не подписанных значений.

54 Глава 3 Введение в синтаксис набора инструкций ARM: <instruction>{<second>}{S} Rd, Rn, N ADC добавляют два 32-битных значения и переносят Rd=Rn+N+carry ADD добавляют два 32-битных значения Rd=Rn+N RSB обратный вычеты два 32-битных значения Rd=N−Rn RSC обратный вычеты с переносом двух 32-битных значений Rd=N−Rn!(переносительный флаг) SBC вычеты с переносом двух 32-битных значений Rd=Rn−N!(переносительный флаг) SUB вычеты два 32-битных значения Rd=Rn−N результат операции Shifter. Синтаксис операции сдвиг показана в таблице 3.3. Пример 3.4Эта простая инструкция по вычитанию вычитает значение, хранящееся в реестре r2 из значения, хранящегося в реестре r1. Результат хранится в реестре r0. PREr0 = 0x00000000 r1 = 0x00000002 r2 = 0x00000001 SUB r0, r1, r2 POSTr0 = 0x00000001 ■ Пример 3.5Эта инструкция об обратном вычитании (RSB) вычитает r1 из постоянного значения #0, записывая результат на r0. Вы можете использовать эту инструкцию для отрицания числа. PREr0 = 0x00000000 r1 = 0x00000077 RSB r0, r1, #0 ; Rd = 0x0 - r1 POSTr0 = -r1 = 0xffffffff89 ■ Пример 3.6Инструкция SUBS полезна для декрементирования счетчиков петли. В этом примере мы вычитаем непосредственное значение одно из значения, хранящегося в реестре r1. Cpsr обновляется с установкой ZCflags. PREcpsr = nzcvqiFt\_USER r1 = 0x00000001SUBS r1, r1, #1

3.1 Инструкции по обработке данных 55 POSTcpsr = nZCvqiFt\_USER r1 = 0x00000000 ■ 3.1.4 Использование смесителя бочки с арифметическими инструкциями Широкий спектр второго операдного смещения, доступный на арифметических и логических инструкциях, является очень мощной особенностью набора инструкций ARM. Пример 3.7 иллюстрирует использование смесителя бочки с встроенной арифметической инструкцией. Пример 3.7 Регистр r1 первоначально переносится в одно место налево, чтобы дать значение в два раза r1. Конечный результат, переданный в реестр r0=три раза больше значения, хранящегося в реестре r1. PREr0 = 0x00000000 r1 = 0x00000005 ADD r0, r1, r1, LSL #1 POSTr0 = 0x0000000f r1 = 0x00000005 ■ 3.1.5 Логические инструкции выполняют битовые логические операции на двух исходных реестрах. Синтаксис: <инструкция>{<вторых>}{S} Rd, Rn, AND логическая битовая и N двух 32-битных значений Rd=Rn&N ORR логическая битовая или двух 32-битных значений Rd=RnNOR EOR логическая битовая или двух 32-битных значений Rd=RnN BIC логическая битовая (Rd=RnR) Rd=Rn0808080R080R080R080R080R080R080R080R080R080R080R080R080R080R080R080R080R080R080R080R080R080R080R080R080R080R080R080R080R080R080R080R080R080R080R00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000

56 Глава 3 Введение в наборы инструкций ARM ORR r0, r1, r2 POSTr0 = 0x12345678 ■ Пример 3.9В этом примере показана более сложная логическая инструкция под названием BIC, которая выполняет логический бит четкий. PREr1 = 0b1111 r2 = 0b0101 BIC r0, r1, r2 POSTr0 = 0b1010 Это эквивалентно R d=R nAND NOT(N) В этом примере реестр r2 содержит бинарный шаблон, где каждый бинарный 1 в r2 очищает соответствующее местоположение бита в реестре r1. Эта инструкция особенно полезна при очистке битов состояния и часто используется для изменения масок прерывания в cpsr. ■ Логические инструкции обновляют знаки cpsr только при наличии ссыфкса. Эти инструкции могут использовать операторы второго значения с перемещением бареля таким же образом, как и арифметические инструкции. 3.1.6 Инструкции по сравнению Инструкции по сравнению используются для сравнения или тестирования реестра с 32-битным значением. Они обновляют биты флага cpsr в соответствии с результатом, но не влияют на другие реестры. После установки битов информация может быть использована для изменения потока программы с использованием условного исполнения. Для получения дополнительной информации о условном исполнении см. раздел 3.8. Для обновления флагов не нужно применять Ssuffix для сравнительных инструкций. Синтаксис: <instruction>{<cond>} Rn, N CMN сравнить отрицательные флагов, установленные в результате Rn+N CMP сравнить флагов, установленные в результате Rn−N TEQ-теста на равенство двух 32-битных значений флагов, установленных в результате RnN TST-тест-битов 32-битных знаков, установленных в результате Rn&N

3.1 Инструкции по обработке данных 57 Нис результат операции сдвиг. Синтаксис операции сдвиг показан в таблице 3.3. Пример 3.10В этом примере показана инструкция по сравнению CMP. Вы можете видеть, что оба реестра, r0 и r9, равны перед выполнением инструкции. Значение zflag перед выполнением 0 и представлено малым буквой z. После выполнения флаг z изменяется на 1 или в большом букве Z. Это изменение указывает на равенство. PREcpsr = nzcvqiFt\_USER r 0=4 r 9=4 CMP r0, r9 POSTcpsr = nZcvqiFt\_USER TheCMP является фактически инструкцией для вычитания с результатом, отброшенным; аналогично инструкция TST является логической операцией AND, а TEQ является логической эксклюзивной операцией OR. Для каждого из них результаты отброшены, но биты условий обновляются в cpsr. Важно понимать, что инструкции по сравнению изменяют только знаки условий cpsr и не влияют на регистри, которые сравниваются. ■ 3.1.7 Умножение инструкций Умножение инструкций умножается содержание пары реестров и, в зависимости от инструкции, накапливается результаты в другом реестре. Конечный результат помещается в реестр назначения или пары реестров. Синтаксис: MLA{<cond>}{S} Rd, Rm, Rs, Rn MUL{<cond>}{S} Rd, Rm, Rs MLA умножить и накопить Rd=(RmRs) +Rn MUL умножить Rd=RmRs Синтаксис: <инструкция>{<cond>}{S} RdLo, RdHi, Rm, Rs SMLAL подписан умножить накопиться длинный [RdHi , RdLo]=[RdHi , RdLo]+RmRs) SMULL умножить длинный [RdHi , RdLo]=RmRs UML неописанный [RdHi , RdLo] МОГОМНОМНОМНОМНОМНОМНОМНОМНОМНОМНОМНОМНОМНОМНОМНОМНОМНОМНОМНОМНОМНОМНОМОМНОМОМНОМОМНОМОМОМНОМОМОМОМОМОМОМОМОМОМОМОМОМОМОМОМОМОМОМОМОМОММОМОМОММОМОМОМММОМОМОМОМОМОММОМОММ [RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR

58 Глава 3 Введение на наборы инструкций ARM Количество циклов, необходимых для выполнения инструкции умножения, зависит от реализации процессора. Для получения дополнительных подробностей о времени цикла см. Приложение D. Пример 3.11В этом примере показана простая инструкция умножения, которая умножает реестры r1andr2 вместе и помещает результат в реестр r0. В этом примере реестр r1 равен значению 2, а r2 равен 2. Результат, 4, затем помещается в реестр r0. PREr0 = 0x00000000 r1 = 0x00000002 r2 = 0x00000002 MUL r0, r1, r2 ; r0 = r1\*r2 POSTr0 = 0x00000004 r1 = 0x00000002 r2 = 0x00000002 ■ Длинные умножительные инструкции (SMLAL, SMULL, UMLAL и UMULL) дают 64-битный результат. Результат слишком большой, чтобы поместить один 32-битный реестр, поэтому результат помещается в два реестра с маркировкой RdLo и RdHi. RdLo содержит нижние 32 бита 64-битного результата, а RdHi содержит более высокие 32 бита 64-битного результата. Пример 3.12 показывает пример длинной неподписанной инструкции умножить. Пример 3.12Инструкция умножает реестры r2 и r3 и помещает результат в реестр r0 и r1.Реестр r0 содержит нижние 32 бита, а реестр r1 содержит верхние 32 бита 64-бита.PREr0 = 0x00000000 r1 = 0x00000000 r2 = 0xf0000002 r3 = 0x00000002 UMULL r0, r1, r2, r3 ; [r1, r0] = r2\*r3 POSTr0 = 0xe0000004 ; = RdLo r1 = 0x00000001 ; = RdHi ■ 3.2 Приведенные инструкции Раздел Приведенное приложение изменяет поток выполнения или используется в рутинной форме. Этот тип инструкции позволяет программам иметь субпрограмы, структуры, если-то-то-другие, и петли.

3.2 Инструкции 59 В соответствии с инструкцией ARMv5E, набор инструкций включает в себя четыре различных инструкции к отряду. Синтаксис: Б{<вторых>} накладка BL{<вторых>} накладка BX{<вторых>} Rm BLX{<вторых>} накладка Rm B branch pc=label BL branch с ссылкой pc=label lr=адрес следующей инструкции после BL BX накладка pc=Rm &0xfffffffe, T=Rm&1 BLX накладка с ссылкой pc=label, T =1 pc=Rm & 0xfffffffe, T=Rm&1 lr=address следующей инструкции после BLX Накладка адреса хранится в соотношении как инструкция к отряду с pc-относителем в пределах 32 MB и должна быть подписана на отряду с отрядом. Приведенный пример показывает направление вперед и впоследствии. Поскольку эти петли являются адресовыми, мы не включаем предварительные и последующие условия. Вперед ADD r1, r2, #4ADD r0, r6, #2 ADD r3, r7, #4 вперед SUB r1, r2, #4 назад ADD r1, r2, #4SUB r1, r2, #4 ADD r4, r6, r7B Вперед ADD r2, r7B Вперед ADD r1, r2, r6 и r7B Вперед ADD r2, r6, r6 и r7B Вперед ADD r2, r2, r2, r2, r2, r2, r2, r2, r2, r4 и r4R, r2, r2, r7B Вперед ADD r2, r2, r2, r2, r2, r7B, r7B, r7B, r7B, r7B, r7B, r7B, r7B, r7B, r7B, r7B, r7B, r7B, r7B, r7B, r7B, r7B, r7B, r7B, r7B, r7B, r7B, r7B, r7B, r7B, r7B, r7B, r7B, r7B, r7B, r7B, r7B, r7B, r7B, r7B, r7B, r7B, r7B, r7B, r7B, r7B, r7B, r7B, r7B, r7B, r7B, r7B, r7B, r7B, r7B, r7B, r7B, r7B, r7B, r7B, r7B, r7B, r7B, r7B, r7B, r7B, r7B, r7B, r7B, r7B, r7B, r7B, r7B, r7B, r7B, r7B, r7B, r7B, r7B, r7B, r7B, r7B, r7B, r7B, r7B, r7B, r7B, r7B, r7B, r7B, r7B, r7B, r7B, r7B, r7B, r7B, r7B В этом примере этикетки наперед и назад. Этикетки ветвей размещаются в начале строки и используются для обозначения адреса, который может быть использован монтажчиком для расчета распределения ветвей. ■

60 Глава 3 Введение в наборы инструкций ARM Пример 3.14Указание с ветви, или BL, похоже на Binstruction, но переписывает реестр ссылок с возвращением адреса. Он выполняет подпрограммный вызов. Для возвращения из субрутины вы копируете реестр ссылок на компьютер. Подрутины BL; отрасль к подрутине CMP r1, #5; сравните r1 с 5 MOVEQ r1, #0; если (r1==5) то r1 = 0: подрутины <код подрутины> MOV pc, lr; возвращение с движением pc = lr Обмен ветвями (BX) и обмен ветвями с ссылкой (BLX) являются третьим типом инструкции ветвя. BXinstruction использует абсолютный адрес, хранящийся в регистре Rm.It используется в первую очередь для подключения к и от кода Пальца, как показано в главе 4. Tbit в cpsr обновляется наименее значимым битом в регистре подключения. Аналогично инструкция BLX обновляет Tbit в cpsr с наименее значимым битом и дополнительно устанавливает реестр ссылки с возвращающим адресом. 3.3 Инструкции для загрузки: Инструкции для загрузки: передача данных между реестрами памяти и процессора. Поддерживаемые типы данных - подписанные и неподписанные слова (32-битные), полусловы (16-битные) и байты. Вот различные инструкции по передаче одного реестра для кладовых магазинов. Синтаксис: <LDR vyiSTR>{<cond>}{B} Rd, адресуя1 LDR{<cond>}SBH в том случае, чтобы сделать Rd, адресуя2 STR{<cond>}H Rd, адресуя2 LDR загружают слово в реестр Rd <-mem32[адрес] STR save byte или слово из реестра Rd -> mem32[адрес] LDRB load byte в реестр Rd <-mem8[адрес] STRB save byte из реестра Rd ->8[адрес]

3.3 Загрузка-указание 61 LDRH загрузка полуслова в реестр Rd <- mem16[адрес] STRH сохранение полуслова в реестр Rd -> mem16[адрес] LDRSB загрузка подписанная байт в реестр Rd <- SignExtend (mem8[адрес]) LDRSH загрузка подписанного полуслова в реестр Rd <- SignExtend (mem16[адрес]) Таблицы 3.5 и 3.7, которые должны быть представлены, являются Раздел 3.3.2, описывают синтаксис адресования1 и адресования2. Пример 3.15 LDRandSTR инструкции могут загружать и хранить данные на границе сровки, которая является той же, что размер загружаемого или храняемого типа данных. Например, LDR может загружать только 32-битные слова на адрес памяти, который является кратным из четырех байтов0, 4, 8 и так далее. В этом примере показано загрузка от адреса памяти, содержащегося в регистре r1, затем хранилище обратно в тот же адрес в памяти. ; ; загрузка регистра r0 с содержанием; адреса памяти, указанного в регистре r1. ; LDR r0, [r1] ; = LDR r0, [r1, #0] ; ; ; хранение содержания регистра r0 до ; адрес памяти, указанный в регистре r1.; STR r0, [r1] ; = STR r0, [r1, #0] Первая инструкция загружает слово из адреса, хранящегося в регистре r1 и помещает его в регистр r0. Второе указание идет в обратном направлении, храняя содержимое реестра r0 на адрес, содержащийся в реестре r1. Эти режимы включают в себя один из методов индексации: предварительный индекс с отписью, предварительный индекс и постиндекс (см. Таблица 3.4).

62 Глава 3 Введение в Таблица наборов инструкций ARM Таблица 3.4 Индексные методы. Метод индекса адреса базы Регистр данных Пример Преиндекс с записью mem[base + offset] base + offset LDR r0,[r1,#4]! Преиндекс mem[base + offset] не обновлен LDR r0,[r1,#4] Postindex mem[base] base + offset LDR r0,[r1],#4 Примечание: ! Пример 3.16Preindex с записью заново рассчитывает адрес из базового реестра плюс адресный оффсет, а затем обновляет адресный реестр с новым адресом. Напротив, преиндекс-оффсет является тем же, что и преиндекс с записью, но не обновляет реестр базы адресов. Postindex обновляет реестр базы адресов только после использования адреса. Постиндекс и предварительный индекс с режимами записывания полезны для пересечения массива. PRE r0 = 0x00000000 r1 = 0x00090000 mem32[0x00009000] = 0x01010101 mem32[0x00009004] = 0x02020202 LDR r0, [r1, #4]! Предварительный индекс с записью: POST(1) r0 = 0x02020202 r1 = 0x00009004 LDR r0, [r1, #4] Предварительный индекс: POST(2) r0 = 0x02020202 r1 = 0x00009000 LDR r0, [r1], #4 Постиндекс: POST(3) r0 = 0x0100000101 r1 = 0x9004

3.3 Инструкции по загрузке 63 Таблица 3.5 Единый реестр загрузки-магазина адресование, слово или незаписанный байт. Адрес1мода и индекс метода Адрес1синтаксис Преиндекс с немедленным отставанием [Rn, #+/-оффсет\_12] Преиндекс с регистрационным отставанием [Rn, +/-Rm] Преиндекс с масштабированным отставанием [Rn, +/-Rm] Преиндекс с масштабированным отставанием [Rn, +/-Rm] Преиндекс с масштабированным отставанием [Rn, +/-Rm, +/-Rm, #shift\_imm]! Преиндекс с масштабированным отставанием [Rn, +/-Rm], Пример #rm, +/-Rm, +/-Rm, +/-Rm, +/-Rm, +/-Rm, +/-Rm, +/-Rm, +/-Rm, +/-Rm, +/-Rm, +/-Rm, +/-Rm, +/-Rm, +м, +м, +м, +м, +м, +м, +м, +м, +м, +м, +м, +м, +м, +м, +м, +м, +м, +м, +м, +м, +м, +м, +м, +м, +м, +м, +м, +м, +м, +м, +м, +м, +м, +м, +м, +м, +м, +м, +м, +м, +м, +м, +м, +м, +м, +м, +м, +м, +м, +м, +м, +м, +м, +м, +м, +м, +м, +м, +м, +м, +м, +м, +м, +м, +м, +м, +м, +м, +м, +м, +м, +м, +м, +м, +м, +м, +м, +м, +м, +м, +м, В этом примере показано, как каждый метод индексации влияет на адрес, содержащийся в регистре r1, а также на данные, загруженные в регистр r0. Каждое указание показывает результат метода индекса с одним и тем же предварительным условием. ■ Режим адресования, доступный при определённой нагрузке или инструкции хранения, зависит от класса инструкции. В таблице 3.5 показаны режимы адресования, доступные для загрузки и хранения 32-битного слова или незаписанного байта. Реестр базовых адресов - это указатель на байт в памяти, а офсет указывает количество байтов. Немедленно означает, что адрес рассчитывается с использованием регистра базовых адресов и 12-битного офсета, кодированного в инструкции. Скаленный означает, что адрес рассчитывается с использованием базового регистра адресов и операции смены барреля. В таблице 3.8 показаны вариации инструкций STRH. 3.3.3 Многочисленные регистрационные перечисления Загрузка многочисленные инструкции хранилища могут передавать несколько регистра между памятью и процессором в одной инструкции. Передача происходит из базового регистра адресов Rnpointing в память. Инструкции по передаче нескольких реестров более эффективны, чем передачи одного реестра для перемещения блоков данных вокруг памяти и сохранения и восстановления контекста и стеков.

64 Глава 3 Введение в наборы инструкций ARM Таблица 3.6 Примеры инструкций LDR с использованием различных режимов адресования. Инструкция r0= r1+= Преиндекс LDR r0,[r1,#0x4]! mem32[r1+0x4] 0x4 с записью LDR r0,[r1,r2]! mem32[r1+r2] r2 LDR r0,[r1,r2,LSR#0x4]! mem32[r1+(r2 LSR 0x4)] (r2 LSR 0x4) Преиндекс LDR r0,[r1,#0x4] mem32[r1+0x4] не обновлен LDR r0,[r1,r2] mem32[r1+r2] не обновлен LDR r0,[r1,-r2,LSR #0x4] mem32[r1-(r2 LSR 0x4)] не обновлен Postindex LDR r0,[r1],#0x4 mem32[r1] 0x4 LDR r0,[r1],r2 mem32[r1] r2 LDR r0,[r1],r2,LSR #0x4 mem32[r1] (r2 LSR 0x4) Таблица 3.7 Единый реестр-адрессчет, полузаданный, полузаданный, подписанный и подписанный по адресу, реестр-адрессчет, реестр-адресчет, реестр-адресчет, реестр-адресчет, реестр-адресчет, реестр-адресчет, реестр-адресчет, реестр-адресчет, реестр-адрес-адресчет, реестр-адресчет, реестр-адрес-адрес-адрес-адрес-адрес-адрес-адрес-адрес-адрес-адрес-адрес-адрес-адрес-адрес-адрес-адрес-адрес-адрес-адрес-адрес-адрес-адрес-адрес-адрес-адрес-адрес-адрес-адрес-адрес-адрес-адрес-адрес-адрес-адрес-адрес-адрес-адрес-адрес-адрес-адрес-адрес-адрес-адрес-адрес-адрес-адрес-адрес-адрес-адрес-адрес-адрес-адрес-адрес-адрес-адрес-адрес-адрес-адрес-адрес-адрес-адрес-адрес-адрес-адрес-адрес-адрес-адрес-адрес-адрес-адрес-адрес-адрес mem16[r1+0x4]=r0 0x4 запись STRH r0,[r1,r2]! mem16[r1+r2]=r0 r2 Преиндекс STRH r0,[r1,#0x4] mem16[r1+0x4]=r0 не обновлен STRH r0,[r1,r2] mem16[r1+r2]=r0 не обновлен Postindex STRH r0,[r1],#0x4 mem16[r1]=r0 0x4 STRH r0,[r1],r2 mem16[r1]=r0 r2

3.3 Инструкции для загрузки 65 Многочисленные инструкции для загрузки могут увеличить латентность перерыва. Реализация ARM обычно не прерывает инструкции во время выполнения. Например, на ARM7 многочисленные инструкции для загрузки занимают 2 + Ntcycles, где Nis количество реестров для загрузки и количество циклов, необходимых для каждого последовательного доступа к памяти. Если перерыв был вызван, то он не имеет никакого эффекта до тех пор, пока не будет завершена инструкция по многочисленным записям нагрузки. Компиляторы, такие как armcc, обеспечивают переключатель для управления максимальным количеством реестров, перемещаемых на хранилище, что ограничивает максимальную задержку перерыва. Синтаксис: <LDMSTM>{<cond>}<адресный режим> Rn{!},<registers>{ˆ} LDM загружают несколько реестров {Rd}N<-mem32[адрес запуска + 4N] optional Rn обновленный STM множество реестров {Rd}N-> save32[start + 4N] optional Rn обновленная таблица 3.9 показывает различные режимы адресования для инструкций нагрузки на хранилище. Здесь определяется количество реестров в списке. Любой подреестров текущих реестров может быть передан из нескольких реестров памяти или банков. Базовый реестр RndОпределяет исходный или адрес назначения для многократной инструкции для хранения грузов.Этот реестр может быть обновлен после передачи.Это происходит при регистрации Rnis, за которым следует ! Пример 3.17 В этом примере реестр r0 является базовым реестре Rn и последует за ним !, что указывает на то, что реестр обновляется после выполнения инструкции. В инструкции по загрузке множества вы заметите, что реестры не перечислены отдельно. Вместо этого для идентификации диапазона реестра используется знак -. В этом случае диапазон является от реестра r1tor3inclusive. Каждый реестр также может быть перечислен, используя перерывку, чтобы отделить каждый реестр в скобках { и }.

66 Глава 3 Введение в набор инструкций ARM mem32[0x80010] = 0x01 r0 = 0x00080010r1 = 0x00000000 r2 = 0x00000000 r3 = 0x00000000 LDMIA r0!, {r1-r3} POSTr0 = 0x0008001c r1 = 0x00000001 r2 = 0x00000002 r3 = 0x00000003 Рисунок 3.3 показывает графическое представление. Адреса памяти 0x80010, 0x80014 и 0x80018 содержат значения 1, 2 и 3 соответственно. После загрузки множественная инструкция выполняет реестры r1, r2, и r3 содержат эти значения, как показано на рисунке 3.4. Теперь базовый реестр указывает на адрес памяти 0x8001c после последнего загруженного слова. Теперь заменить инструкцию LDMIA на кратную загрузку и увеличение до инструкции LDMIB и использовать те же условия PRE. После выполнения реестр r0now указывает на место последней загрузки памяти. Это отличается от примера LDMIA, который указывал на место последующей памяти. ■ В вариантах DA и DB для декремента множественных инструкций для хранения загрузки декрементируется стартовый адрес, а затем хранится в местах восходящей памяти. Они также позволяют выполнять операции толкания и вытягивания на стеке, которые будут проиллюстрированы позже в этом разделе. 0x80020 0x8001c 0x80018 0x80014 0x80010 0x8000c0x000000050x00000004 0x00000003 0x00000002 0x00000001 0x00000000r3 = 0x00000000 r2 = 0x00000000r1 = 0x00000000 r0 = 0x80010Памятник адреса Память Показатель адреса Данные Фигура 3.3 Предварительный условий для инструкции LDMIA.

3.3 Нагрузка-указание магазина 67 0x80020 0x8001c 0x80018 0x80014 0x80010 0x8000c0x000000050x000000040x000000030x00000002 0x000000000001 0x000000000000r3 = 0x00000003 r2 = 0x00000002r1 = 0x00000001r0 = 0x8001cПомноситель адреса Память Указатель адреса Данные Фигура 3.4 Последующее состояние для инструкции LDMIA. 0x80020 0x8001c0x80018 0x80014 0x800100x800000c0x000000050x0000000040x00000003 0x00000002 0x0000000000000000x0000r3 = 0x00000004 r2 = 0x000000001r2 LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDMIA LDM Если вы используете магазин с базовой обновлением, то парагрузочная инструкция того же числа реестров перезагрузит данные и восстановит указатель базового адреса. Это полезно, когда вам нужно временно сохранить группу реестров и восстановить их позже.

68 Глава 3 Введение на наборы инструкций ARM Пример 3.18В этом примере показывается STMincrement перед инструкцией, а затем LDMdecrement после инструкции. PRE r0 = 0x00009000 r1 = 0x00000009r2 = 0x00000009r2 = 0x00000008 r3 = 0x00000007 STMIB r0!, {r1-r3} MOV r1, #1 MOV r2, #2 MOV r3, #3 PRE((2) r0 = 0x0000900c r1 = 0x00000001 r2 = 0x00000002 r3 = 0x00000002 r3 LDMDA r0!, {r1-r3} POST r0 = 0x00009000 r1 = 0x00000009 r2 = 0x00000008 r3 = 0x0000000000 The instruction stores the 7, 8, 9 to 9 to memory. Затем мы повреждаем реестр r1tor3. TheLDMDA перезагружает исходные значения и восстанавливает базовый указатель r0. ■ Пример 3.19Мы иллюстрируем использование множественных инструкций для хранения загрузки примером копирования блока памяти. Этот пример - простая рутина, которая копирует блоки из 32 байтов от местонахождения исходного адреса к местонахождению адресной адреса. Пример имеет две инструкции для загрузки-укладки множества, которые используют один и тот же инкремент после обращения режима. ; r9 точки для начала исходных данных; r10 точки для начала данных направления; r11 точки для конца источника петли; загрузка 32 байтов из источника и обновление r9 указателя LDMIA r9!, {r0-r7}

3.3 Загрузка-устройство инструкции 69; хранить 32 байта на место назначения и обновлять указатель r10 STMIA r10!, {r0-r7}; и хранить их; мы достигли окончания циркуляции CMP r9, r11BNE Эта рутина основана на реестрах r9, r10 и r11, которые были установлены до выполнения кода. Реестры r9 и r11 определяют данные, которые должны быть скопированы, и регистрируют r10-точки в память для данных. LDMIA загружает данные, указанные в регистре r9 в реестры r0tor7. Он также обновляет r9 для указания на следующий блок данных, который должен быть скопирован. STMIA копирует содержание реестра r0tor7 в адрес памяти назначения, указанный в регистре r10. Он также обновляет r10 для указания на следующее место назначения. CMPandBNE сопоставляют указатели r9 и r11 для проверки того, достигнут ли конец копии блока. Если блок-копия завершена, то рутинная работа завершается; в противном случае петля повторяется с обновленными значениями реестра r9 и r10. Если предыдущая инструкция сравнения устанавливает, что флаги условия не равны, выполняется инструкция к ветку. На рисунке 3.6 показана карта памяти копии блока памяти и то, как рутина перемещается через память. Теоретически эта петля может передавать 32 байта (8 слов) в двух инструкциях, для максимально возможного пропускной способности 46 МБ/секунда, передаваемого на 33 МГц. Эти цифры предполагают идеальную систему памяти с быстрой памятью. ■ Высокая память Низкая память11 r9 r10ИсточникМедирия местонахождениеКопия памяти Рисунок 3.6Облокировать копию памяти на карте памяти.

70 Глава 3 Введение в набор инструкций ARM 3.3.3.1 Операции на стеке Архитектура ARM использует множественные инструкции для выполнения операций на стеке. Поп-операция (удаление данных из стека) использует инструкцию по загрузке множества; аналогично, операция толчка (вмещение данных на стек) использует инструкцию по хранению множества. При использовании стека вы должны решить, вырастет ли стек или снизится в памяти. Повышающие стаки растут в сторону более высоких адресов памяти; напротив, нисходящие стаки растут в сторону более низких адресов памяти. Когда вы используете полный стак (F), указатель стака указывает на адрес, который является последним использованным или полным местом (т.е. указывает на последний пункт на стаке). В отличие от этого, если вы используете пустой стек (E), то точки сплинтов указывают на адрес, который является первым неиспользованным или пустым местом (т.е. указывает на последний пункт на стеке). Например, для полного восходящего стека будет прилагаться обозначение FA к инструкции по загрузке множества LDMFA. Это будет переведено в инструкцию LDMDA. ARM определил стандарт вызова процедуры ARM-Thumb (ATPCS), который определяет, как называются рутины и как распределяются реестры. Таким образом, инструкции LDMFD и STMFD обеспечивают функции появления и спуска соответственно. Пример 3.20Инструкция STMFD толкает реестры на стек, обновляя сп. Вы можете видеть, что при росте стека указатель стека указывает на последний полный вход в стеке. PREr1 = 0x00000002 r4 = 0x00000003sp = 0x00080014 STMFD sp!, {r1,r4} Таблица 3.11 Методы обращения к операциям стека. Режим обращения Pop =LDM Push =STM FA полный подъем LDMFA LDMDA STMFA STMIB FD полный подъем LDMFD LDMIA STMFD STMDB EA пустой подъем LDMEA LDMDB STMEA EDEA STMEA STMEA LDMED LDMED LIB STMED STMDA

3.3 Инструкции по загрузке в магазине 71 0x80018 0x80014 0x80010 0x8000c0x000010x00000002 Пустые пустые адреса PRE Данные 0x80018 0x80014 0x80010 0x8000c0x000000010x00000002 0x00000003 0x00000002 spАдрес POST Данные Фигура 3.7 СTMFD инструкцияполная работа толчка. POSTr1 = 0x00000002 r4 = 0x00000003 sp = 0x0008000c ■ Пример 3.21В рисунке 3.8 показана контрастная операция на пустой стаке с использованием инструкции STMED толчка. Инструкция STMED толкает реестры на станок, но обновляет реестр, чтобы указать на следующее пустое место. PREr1 = 0x00000002r4 = 0x00000003sp = 0x00080010 STMED sp!, {r1,r4} POSTr1 = 0x00000002 r4 = 0x00000003 sp = 0x00080008 ■ 0x800180x800140x80010 0x8000c 0x80080x000000010x00000002Empty Empty Empty DataPress POST 0x800180180x80014 0x80010 0x8000c8000x80x00000000x00000000x00000000x0000000000x00000000003 0x00000002 SpAddress POST Figurine 3.8 STMED Инструкция ПОСТИ ПОСТИ ПОСТИ ПОСТИ ПОСТИ ПОСТИ ПОСТИ ПОСТИ ПОСТИ

72 Глава 3 Введение на наборы инструкций ARM При обработке проверенной пачки необходимо сохранить три атрибута: основа пакета, указатель пакета и предел пакета. Показатель стека изначально указывает на базу стека; по мере того, как данные наталкиваются на стек, указатель стека снижает память и постоянно указывает на верхнюю часть стека. Вот небольшой код, который проверяет ошибки переполнения стака для нисходящего стака: ; проверяет переполнение стака SUB sp, sp, #size CMP sp, r10BLLO \_stack\_overflow ; условие ATPCS определяет регистра r10 как лимит стака или sl. Это является необязательным, поскольку используется только при включении проверки стака. Инструкция BLLO - это ветвь с инструкцией ссылки плюс условием mnemonic LO.I fsp меньше, чем регистра r10 после того, как новые элементы выталкиваются на станок, то произошла ошибка переполнения стака. Если указатель стака возвращается за основу стака, то произошла ошибка подтока стака. 3.3.4 Инструкция свапа Инструкция свапа является специальным случаем инструкции для кладовых. Он обменивает содержимое памяти содержимым реестра. Эта инструкция является атомной операцией она читает и пишет местоположение в той же операции автобуса, препятствуя любой другой инструкции от чтения или написания в этом месте, пока она не завершится. Синтаксис: SWP{B}{<second>} Rd,Rm,Rn] SWP обмен словом между памятью и реестре tmp = mem32[Rn] mem32[Rn] = Rm Rd = tmp SWPB обмен байтом между памятью и реестре tmp = mem8[Rn] mem8[Rn] = Rm Rd = tmp Swap не может быть прерван каким-либо другим инструкцией или любым другим доступом к автобусу. Мы говорим, что система держит автобус до завершения транзакции. Пример 3.22Инструкция swap загружает слово из памяти в реестр r0 и переписывает память реестр r1.

3.4 Инструкция по перерыву программного обеспечения 73 PREmem32[0x9000] = 0x12345678 r0 = 0x00000000 r1 = 0x11112222 r2 = 0x00009000 SWP r0, r1, [r2] POSTmem32[0x9000] = 0x11112222 r0 = 0x12345678 r1 = 0x11112222 r2 = 0x00009000 Эта инструкция особенно полезна при реализации семафоров и взаимоисключения в операционной системе. Вы можете видеть из синтаксиса, что эта инструкция также может иметь квалификатор размером байтов B, поэтому эта инструкция позволяет как слово, так и обмен байтами. ■ Пример 3.23В этом примере показывается простой защитник данных, который может быть использован для защиты данных от написания другой задачей. Инструкция SWP держит автобус до завершения транзакции. spin MOV r1, =semaphore MOV r2, #1 SWP r3, r2, [r1]; держите автобус до завершения CMP r3, #1 BEQ spin Адрес, указанный семафором, содержит значение 0 или 1. Рутинная схема будет продолжать циркулировать, пока услуга не будет выпущена другим процессомдругими словами, когда местоположение семафора адреса содержит значение 0. ■ 3.4 Инструкция для прерывания программного обеспечения Инструкция для прерывания программного обеспечения (SWI) вызывает исключение для прерывания программного обеспечения, которое обеспечивает механизм для приложения для вызова рутин операционной системы. Синтаксис: SWI{<second>} SWI\_number SWI программное обеспечение прерывает lr\_svc=адрес инструкции по SWI spsr\_svc =cpsr pc=векторы +0x8 cpsr mode=SVC cpsr I=1 (маска IRQ прерывает)

74 Глава 3 Введение на наборы инструкций ARM Когда процессор выполняет SWI-инструкцию, он устанавливает счетчик программы к офсету0x8 в векторной таблице. Инструкция также вынуждает режим процессора работать в SVC, что позволяет вызвать рутину операционной системы в привилегированном режиме. У каждой инструкции SWWI есть связанный номер SWI, который используется для представления конкретного вызова или функции. Пример 3.24Вот простый пример вызова SWI с номером SWWI0x123456, используемый инструментами ARM в качестве дебгугирующего SWI. Обычно SWI-инструкция выполняется в режиме пользователя. PREcpsr = nzcVqift\_USER pc = 0x00008000 lr = 0x003fffff; lr = r14 r0 = 0x12 0x00008000 SWI 0x123456 POSTcpsr = nzcVqIft\_SVC spsr = nzcVqift\_USER pc = 0x00000008 lr = 0x00008004 r0 = 0x12 Поскольку инструкции SWI используются для вызова операционной системы, вам нужна какая-то форма прохождения параметров. В этом примере реестр r0 используется для передачи параметра 0x12. Возвратные значения также передаются через реестры. ■ Для обработки SWI-звона требуется код, называемый оператором SWI. Управляющий получает SWInumber с помощью адреса выполненной инструкции, которая рассчитывается из реестра ссылок lr. Номер SWI определяется SWI\_Number = <SWI instruction> И NOT(0xff000000) Здесь SWI instruction является фактической 32-битной SWI instruction выполненной процессором. Пример 3.25Этот пример показывает начало реализации SWIhandler. Фрагмент кода определяет, какое число SWI называется, и помещает его в реестр r10. Вы можете видеть из этого примера, что инструкция загрузки сначала копирует полную инструкцию SWI в реестр r10. Предполагаем, что SWI вызван из состояния ARM. SWI\_handler ;; Магазин регистра r0-r12 и регистра ссылок

3.5 Инструкции по регистру статуса программы 75 ; STMFD sp!, {r0-r12, lr}; Читайте инструкцию SWI LDR r10, [lr, #-4] ; Маскируйте верхние 8 бит BIC r10, r10, #0xff000000; r10 - содержит номер SWI BL service\_routine; возвращайтесь из оператора SWI LDMFD sp!, {r0-r12, pc}ˆ Номер в регистре r10 затем используется оператором SWI для вызова соответствующей рутины SWI. ■ 3.5 Инструкции по регистру статуса программы Комплект инструкций ARM предоставляет две инструкции для непосредственного управления регистра статуса программы (psr). Инструкция MRS переносит содержание либо cpsr, либо spsr в регистр; в обратном направлении инструкция MSR переносит содержание регистра в cpsrorspsr. В совокупности эти инструкции используются для чтения и написания cpsrandspsr. В синтаксисе вы можете увидеть этикетку, называемую полями. Это может быть любая комбинация контроля (c), расширения (x), статуса (s) и флагов (f). Эти поля относятся к отдельным районам байтов в apsr, как показано на рисунке 3.9. Синтаксис: MRS{<second>} Rd,<cpsr khipsspsr> MSR{<second>} <cpsr khipsspsr>\_<fields>,Rm MSR{<second>} <cpsr khipsspsr>\_<fields>,#immediate Fields Bit 31 30 29 28 NZCV7654 0 IF T ModeFlags [24:31] Статус [16:23] Экстенсия [8:15] Контроль [0:7] Рисунок 3.9 fields psrbyte.

76 Глава 3 Введение в ARM Инструкция набор MRS копировать реестр статуса программы в регистр общего назначения Rd=psr MSR переместить реестр общего назначения в регистр статуса программы psr[field] =Rm MSR переместить немедленное значение в регистр статуса программы psr[field] =immediate Cfield контролирует маски прерывания, состояние пальца и режим процессора. Пример 3.26 показывает, как активировать прерывания IRQ путем очистки Imask. Пример 3.26В случае, если MRSR первым копирует cpsr в реестр r1, то инструкция BIC очищает бит 7 r1. Вы можете видеть из этого примера, что этот код сохраняет все другие настройки в cpsr и изменяет только Ibit в поле управления. PREcpsr = nzcvqIFt\_SVC MRS r1, cpsr BIC r1, r1, #0x80 ; 0b01000000 MSR cpsr\_c, r1 POSTcpsr = nzcvqiFt\_SVC Этот пример находится в режиме SVC. В режиме пользователя вы можете прочитать все биты cpsr, но вы можете только обновить поле флага условия f. ■ 3.5.1 Инструкции для сопроцессора Инструкции для сопроцессора используются для расширения набора инструкций. Копроцессор может либо обеспечивать дополнительные вычислительные возможности, либо использоваться для управления подсистемой памяти, включая кеши и управление памятью. Инструкции для сопроцессора включают обработку данных, передачу реестра и инструкции по передаче памяти. Мы предоставим только краткий обзор, поскольку эти инструкции являются специфичными для сопроцессора. Обратите внимание, что эти инструкции используются только ядрами с сопроцессором. Синтаксис: CDP{<cond>} cp, opcode1, Cd, Cn {, opcode2} <MRCMCR> <<cond>} cp, opcode1, Rd, Cn, Cm {, opcode2} <LDCSTC>{<cond>} cp, Cd, адресующие

3.5 Инструкции по регистру статуса программы 77 обработка данных сопроцессором CDPвыполнять операцию в копроцессоре MRC MCR перевод реестра сопроцессораперемещение данных в/из реестра сопроцессора LDC STC перевод памяти сопроцессоранагрузка и хранение блоков памяти в/из копроцессора В синтаксисе инструкций сопроцессора cpfield представляет число сопроцессора между p0 и p15. Поле опкода описывают операцию, которая должна быть выполнена на сопроцессоре. Поле Cn, Cm и Cd описывают реестры внутри сопроцессора. Операции сопроцессора и реестры зависят от конкретного сопроцессора, который вы используете. Копроцессор 15 (CP15) предназначен для системного управления, таких как управление памятью, управление буферным записью, контроль на кеше и регистра идентификации. Пример 3.27Этот пример показывает, что реестр CP15 копируется в реестр общего назначения. ; передача содержания реестра CP15 c0 для регистрации r10 MRC p15, 0, r10, c0, c0, 0 Здесь реестр CP15-0 содержит идентификационный номер процессора. Этот реестр копируется в реестр общего назначения r10. ■ 3.5.2 Копроцессор 15 Инструкция Синтаксис CP15 конфигурирует ядро процессора и имеет набор специальных реестра для хранения конфигурационной информации, как показано в Примере 3.27. При этом, как для MRC, так и для MCR, инструкции используются для чтения и записи в CP15, где регистрация Rdis является основным регистратом назначения, Cnis - первичным регистратом, Cm - вторичным регистратом, а opcode2 - вторичным регистратором модификатора. Вы можете иногда слышать вторичные реестры, называемые расширенные реестры. Например, вот инструкция для перемещения контента реестра управления CP15 c1 в реестр r1 процессора ядра: MRC p15, 0, r1, c1, c0, 0 Мы используем сокращенную обозначение для ссылки CP15 , которая облегчает следование ссылкам на конфигурационные реестры.

78 Глава 3 Введение на наборы инструкций ARM Первый термин, CP15, определяет его как сопроцессор 15. Второй термин, после разделяющего колонна, является первичным реестре. Первичный реестр Xcan имеет значение от 0 до 15. Третий термин - вторичный или расширенный реестр. Последний термин, opcode2, является модификатором инструкций и может иметь значение от 0 до 7. Некоторые операции также могут использовать не нулевое значение wofopcode1.W и записывать их как CP15:w:cX:cY:Z. 3.6 Загрузочные постоянные Вы, возможно, заметили, что нет инструкции ARM для перемещения 32-битной постоянной в реестр. Поскольку инструкции ARM имеют размер 32 бит, они, очевидно, не могут указать общую 32-битную постоянную. Для помощи программированию есть две псевдоинструкции для перемещения 32-битного значения в реестр. Синтаксис: LDR Rd, =константа ADR Rd, набор LDR нагрузки постоянная псевдоинструкция Rd=32 битная постоянная ADR нагрузка адрес псевдоинструкция Rd=32 битный относительный адрес Первая псевдоинструкция пишет 32-битную постоянную в реестр с использованием любых доступных инструкций. Второй псевдоинструкция записывает относительный адрес в реестр, который будет кодироваться с использованием компьютерно-относительного выражения. Пример 3.28Этот пример показывает LDR инструкцию загрузки 32-битная постоянная 0xff00ff в реестр r0. LDR r0, [pc, #constant\_number-8-{PC}]: constant\_number DCD 0xff00ffff ■ Пример 3.29 показывает альтернативный метод загрузки той же постоянной в реестр r0by с использованием инструкции MVN.

3.7 ARMv5E Расширения 79 Таблица 3.12 Псевдоинструкция конверсии LDR. Псевдоинструкция Фактическая инструкция LDR r0, =0xff MOV r0, #0xff LDR r0, =0x55555555 LDR r0, [pc, #offset\_12] Пример 3.29 Загрузка постоянной 0xff00ff с использованием MVN. PREnone... MVN r0, #0x00ff0000 POSTr0 = 0xff00ff ■ Как вы можете видеть, есть альтернативы доступу к памяти, но они зависят от постоянной, которую вы пытаетесь загрузить. Компиляторы и монтажники используют умные методы, чтобы избежать загрузки постоянной из памяти. Эти инструменты имеют алгоритмы для поиска оптимального количества инструкций, необходимых для создания постоянной в регистре и широкого использования смесителя баррелей. Псевдоинструкция LDR либо вставляет инструкцию MOV или MVN для создания значения (если это возможно), либо генерирует инструкцию LDR с компьютерным адресом для чтения постоянной из буквального пуля данных, встроенного в код. Мы рекомендовали использовать эту псевдоинструкцию для загрузки постоянной. Чтобы увидеть, как монтажник обрабатывает определенную постоянную нагрузку, вы можете проложить выход через демонтажник, который перечислит инструкцию, выбранную инструментом для загрузки постоянной. В данной инструкции адрес данной этикетки помещается в регистр Rd, используя компьютерную добавку или вычитание. 3.7 ARMv5E Расширения ARMv5E предоставляют множество новых инструкций (см. Таблица 3.13). Эти операции являются одноциклом на многих реализациях ARMv5E. ARMv5E обеспечивает большую гибкость и эффективность при манипулировании 16-битными значениями, что важно для таких приложений, как 16-битная цифровая аудиопереработка.

80 Глава 3 Введение в наборы инструкций ARM Таблица 3.13 Новые инструкции, предоставленные расширениями ARMv5E. Описание инструкции CLZ {<cond>} Rd, Rm подсчет ведущих нулей QADD {<cond>} Rd, Rm, Rn подписанный насыщенный 32-битный добавление QDADD{<cond>} Rd, Rm, Rn подписанный насыщенный двойной 32-битный добавление QDSUB{<second>} Rd, Rm, Rn подписанный насыщенный двойной 32-битный вычеты QSUB{<second>} Rd, Rm, Rn подписанный насыщенный 32-битный вычеты SMLAxy{<second>} Rd, Rm, Rn, Rn, Rn, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd Пример 3.30 показывает пример инструкции CLZ. Пример 3.30 Вы можете видеть из этого примера, что первый бит, установленный на 1, имеет 27 нулей, предшествующих ему. CLZ полезен в рутинах, которые должны нормализовать числа. PREr1 = 0b0000000000000000000000000000000010000 CLZ r0, r1 POSTr0 = 27 ■ 3.7.2 Насыщенная арифметическая нормальная арифметическая инструкция ARM обернуться вокруг, когда вы переполните цельное значение. Например, 0x7ffffffff+1=-0x80000000. Таким образом, при разработке алгоритма, вы должны быть осторожны, чтобы не превышать максимально представляемый значение в 32-битном целом число. Пример 3.31Этот пример показывает, что происходит, когда максимальное значение превышается. PREcpsr = nzcvqiFt\_SVC r0 = 0x00000000 r1 = 0x70000000 (положительно) r2 = 0x7fffffff (положительно)

3.7 ARMv5E Расширения 81 ADDS r0, r1, r2 POSTcpsr = NzcVqiFt\_SVC r0 = 0xefffffff (отрицательно) В примере реестры r1 и r2 содержат положительные числа. Реестр r2 равно 0x7fffff, что является максимальным положительным значением, которое можно хранить в 32 битах. В перфектном мире, если добавить эти числа вместе, то получится большое положительное число. Вместо этого значение становится отрицательным и устанавливается флаг переполнения, V. ■ В отличие от этого, используя инструкции ARMv5E, вы можете насытить результат, как только превысит наибольшее число, результаты остаются на максимальном значении 0x7fffffff. Это избегает необходимости использования дополнительного кода для проверки возможных перетоков. В таблице 3.14 приведены все инструкции по насыщению ARMv5E. В таблице 3.14 указания на насыщение. Инструкция Насыщенное расчет QADD Rd=Rn+Rm QDADD Rd=Rn+(Rm2) QSUB Rd=Rn−Rm QDSUB Rd=Rn−(Rm2) Пример 3.32В этом примере показываются те же данные, которые передаются в инструкцию QADD. PREcpsr = nzcvqiFt\_SVC r0 = 0x00000000r1 = 0x70000000 (положительный) r2 = 0x7fffffff (положительный) QADD r0, r1, r2 POSTcpsr = nzcvQiFt\_SVC r0 = 0x7ffffffffffff Вы вернете насыщенный номер в реестр r0. Также установлен Qbit (бит 27 cpsr), что указывает на наличие насыщения. Qflag липкий и останется установленным до тех пор, пока не будет четко очищено. ■ 3.7.3 ARMv5E Умножение инструкций Таблица 3.15 показывает полный список инструкций умножения ARMv5E.

82 Глава 3 Введение в наборы инструкций ARM Таблица 3.15 Подписанные умножить и умножить инструкции накопления. Подписанные Указание с флагом Q с флагом [Акумулировать] результат обновлен Расчет SMLAxy (16-бит \* 16-бит) + 32-битный 32-битный да Rd=(Rm.x \*Rs.y) + Rn SMLALxy (16-битный \* 16-битный) + 64-битный [RdHi, RdLo]+= Rm.x \*Rs.y SMLAWy ((32-битный \*16-битный) / большемуч16) + 32-битный да Rd=ULRm \*Rs.y) / большемуч16) + Rn SMLALxy (16-битный \*16-битный) + 32-битный Rd=Ry\*Ry\*Ry\*Ry\*Ry) \*Rd=32-битный (Rd16-Ry) \*Rd=32-bitный (Rd16-Ry) \*Rd-Ry-Ry) \*Ry-Ry-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y-y Эти поля назначаются буквой T для верхних 16-битных или буквой B для нижних 16-битных. Для умножения накопительных операций с 32-битным результатом Qflag указывает, если накопитель переполняет подписанное 32-битное значение. Пример 3.33Этот пример показывает, как вы используете эти операции. Пример использует подписанную инструкцию умножить накопление, SMLATB. PREr1 = 0x20000001 r2 = 0x20000001r3 = 0x00000004 SMLATB r4, r1, r2, r3 POSTr4 = 0x00002004 Инструкция умножает верхние 16 бит реестра r1 на нижние 16 бит реестра r2. Она добавляет результат для регистрации r3 и записывает его в регистра назначения r4. ■ 3.8 Условное выполнение Большинство инструкций ARM выполняются условновы можете указать, что инструкция выполняется только в том случае, если знаки кода условия проходят определенное условие или испытание. Используя инструкции условного исполнения, вы можете увеличить производительность и плотность кода. Поле условий - это двухбуквенная мнемоника, приложенная к инструкции мнемоника. По умолчанию мнемоника - AL, o ralways execute. Условное исполнение уменьшает количество ветвей, что также уменьшает количество выбросов трубопровода и, таким образом, улучшает производительность выполненного кода. Условное выполнение зависит от двух компонентов: поле условий и знаки условий.

3.8 Условное выполнение 83 Пример 3.34В этом примере показана инструкция ADD с приложением EQcondition. Эта инструкция будет выполняться только тогда, когда нулевой флаг в cpsr установлен на 1. ; r0 = r1 + r2 если нулевой флаг установлен ADDEQ r0, r1, r2 Только инструкции по сравнению и инструкции по обработке данных с приложением Ssuffix к мнемоническому обновлению условий флаг в cpsr. ■ Пример 3.35 Чтобы помочь проиллюстрировать преимущества условного выполнения, мы возьмем простый фрагмент кода C, показанный в этом примере, и сравним выход сборщика с использованием безусловных и условных инструкций. В этом примере используется только условное исполнение на инструкции ветви: ; Greatest Common Divisor Algorithm gcd CMP r1, r2 BEQ complete BLT lessthanSUB r1, r1, r2 B gcd lessthan SUB r2, r2, r1 B gcd complete... Теперь сравнить тот же код с полной условной исполнением. Как видите, это резко уменьшает количество инструкций: gcd CMP r1, r2

84 Глава 3 Введение на наборы инструкций ARM SUBGT r1, r1, r2 SUBLT r2, r2, r1BNE gcd ■ 3.9 Резюме В этой главе мы рассмотрели набор инструкций ARM. Все инструкции ARM имеют длину 32 бит. Инструкции по арифметике, логике, сравнению и движению могут использовать встроенный смещатель барреля, который заранее обрабатывает второй регистр Rm, прежде чем он входит в ALU. Комплект инструкций ARM имеет три типа инструкций по хранению грузов: однорегистрный склад, многорегистрный склад и свап. Стандарт звонков по процедурам ARM-Thumb (ATPCS) определяет стак как полный нисходящий стак. Инструкция по прерыванию программного обеспечения вызывает прерывание программного обеспечения, которое заставляет процессор входить в режим SVC; эта инструкция призывает привилегированные процедуры операционной системы. Инструкции по регистру статуса программы записываются и читаются в cpsr и ssr. Существуют также специальные псевдоинструкции, которые оптимизируют загрузку 32-битных констант. Расширения ARMv5E включают пересчет ведущих нулей, насыщение и улучшенные инструкции умножения. Инструкция пересчета ведущих нулей считает количество бинарных нулей до первого бинарного. Насыщение обрабатывает арифметические расчеты, которые переполняют 32-битное целое число. Улучшенные инструкции умножения обеспечивают лучшую гибкость при умножении 16-битных значений.

Эта страница умышленно осталась пустой

4.1 Использование регистра пальцев 4.2 Взаимодействие ARM-пальца 4.3 Другие инструкции по отрасли 4.4 Инструкции по обработке данных 4.5 Инструкции по единому регистру загрузки-магазин 4.6 Инструкции по многократному регистру загрузки-магазин 4.7 Инструкции по стеку 4.8 Инструкция по прерыванию программного обеспечения 4.9 Резюме

Глава Введение к набору инструкций по пальцу4 В этой главе представлен набор инструкций по пальцу. Поскольку Thumb имеет более высокую производительность, чем ARM на процессоре с 16-битной базой данных, но более низкую производительность, чем ARM на 32-битной базой данных, используйте Thumb для систем с ограниченной памятью. При этом, в среднем, реализация одного и того же кода с помощью Thumb занимает примерно на 30% меньше памяти, чем эквивалентная реализация ARM. Например, на рисунке 4.1 показана такая же рутина деления кода, реализуемая в ARM и Thumb Assembly Code. Несмотря на то, что в реализации Thumb используется больше инструкций, общий запас памяти снижается. Поскольку он также был разработан как целевая компиляторная система, а не для ручной сборной кода, мы рекомендуем вам написать код с целенаправленной палец на языке высокого уровня, например C или C ++. Каждое указание на палец связано с 32-битным указанием ARM. На рисунке 4.2 показано простое указание Thumb ADD, декодируемое в эквивалентное указание ARM ADD. В таблице 4.1 представлен полный список указаний Thumb, доступных в архитектуре THUMBv2, используемой в архитектуре ARMv5TE. Ограниченное пространство в 16 битях делает операции с перемещением баррелей ASR,LSL,LSR иROR отдельными инструкциями в ISA "Тамп". 87

88 Глава 4 Введение к указанию на дюйм-класс Набор кода ARM Дюйм-класс ARMDivide Дюйм-класс; IN: r0(ценность), r1(дивизор) ; IN: r0(ценность), r1(дивизор) ; OUT: r2(модуль), r3(DIVIDE) ; OUT: r2(модуль), r3(DIVide) MOV r3,#0 MOV r3,#0 петля SUBS r0,#0,r1 ADD R3,#1 ADDGE r3,#1 ADDGE r3,r3,#1 SUB r0,r1 BGE loop BGE loop ADD r2,r0,r1 SUB r3, D#1 ADD rumb2,r0,r1 5×4=20 байтов 6×2=12 по фигуре 4.1 DDS. В приложении A для полного перечня инструкций по "Тумплюс". В данной главе рассматриваются использование регистра "Тумплюс", взаимодействие между ARM и "Тумплюс", инструкции по отрасли, инструкции по обработке данных, инструкции по хранению данных, операции по сборке и прерывания программного обеспечения.

4.1 Фундальный регистр Использование 89 Таблица 4.1 Фундальный инструкционный набор. Мнемоника THUMB ISA Описание ADC v1 добавьте два 32-битных значения и переносите ADD v1 добавьте два 32-битных значения И в1 логический битпособно И из двух 32-битных значений ASR v1 арифметический сдвиг правой B v1 ветвь относительно BIC v1 логический битпособно очиститься (И НЕ) из двух 32-битных значений BKPT v2 разрывные указания BL v1 относительный ветвь с ссылкой BLX v2 ветвь с ссылкой и обменом BX v1 ветвь с обменом CMN v1 сравнить отрицательные два 32-битных значения CMP v1 сравнить два 32-битных целых числа EOR v1 логический сдвиг правой OR из двух битных значений LDMI v1 подвиток LDMI v1 подвиток слова с сдвигаются из памяти в ARM реестры LDR v1 одно битное значение Использование в LST v1 LB v1 LB1 LB1 LB1 LB1 LB1 LB1 ЛГ1 ЛГ1 ЛГ1 ЛГ1 ЛГ1 ЛГ1 ЛГ1 ЛГ1 ЛГ1 ЛГ1 НОДРН НОДРНОДДДДДДДДРНОДРНОДРНОДРНОДРНОДРНОДРНОДРНОДРНОДРНОДРНОДРОДРНОДРНОДРЕДРНОДРНОДРОДРЕДРОДРНОДРОДРОДРОДРОДРОДРОДРОДРОДРОДРОДРОДОДОДОДОДОДОДОДОДОДОДОДОДОДОДОДОДОДОДОДОДОДОДОДОДОДОДОДО Только низкие регистры r0tor7 полностью доступны, как показано в таблице 4.2. Высшие регистры r8tor12 доступны только с помощью инструкций MOV,ADD,o rCMP.

90 Глава 4 Введение в комплект указаний по пальцам Таблица 4.2 Резюме использования регистра по пальцам. Регистры Доступ r0r7 полностью доступен r8r12 доступен только с помощью MOV, ADD и CMP r13 sp ограниченная доступность r14 lr ограниченная доступность r15 pc ограниченная доступность cpsr только косвенный доступ spsr нет доступа Вы, возможно, заметили в списке наборов указаний по пальцам и в таблице использования регистра по пальцам, что нет прямого доступа к cpsr или spsr. Другими словами, не существует инструкций для пальца, эквивалентных MSR и MRS. Чтобы изменить cpsr или spsr, вы должны перейти в состояние ARM, чтобы использовать MSR и MRS. Точно так же, в состоянии Thumb нет инструкций для сопроцессора. 4.2 ARM-Thumb Interworking ARM-Thumb interworking - это название метода соединения ARM и Thumb кода вместе для сборки и C / C ++. Он управляет переходом между двумя состояниями. ATPCS определяет стандарты вызова процедуры ARM и Thumb. Для вызова рутины Thumb из рутины ARM, ядро должно изменить состояние. Это изменение состояния показывается в Tbit cpsr. Инструкции BXandBLXbranch вызывают переключение между состоянием ARM и Thumb при пересечении к рутине. В ARMv5T была введена инструкция BLX. На ядрах ARMv4T ссылка использует фанер для переключения состояния на подпрограмму вызова. Вместо того, чтобы называть рутину напрямую, ссылка называет фанер, который переходит на состояние Дюбка с использованием инструкции BX. Существует две версии инструкции BXorBLX: инструкция ARM и эквивалент Дюбка. Инструкция "Тумпл Бкс" делает то же самое. Синтаксис: Бкс Рм Блкс Рм

4.2 ARM-Thumb Interworking 91 BX Thumb version branch exchange p c=R n &0xfffffffe T=R n [0] BLX Thumb version of the branch exchange lr =(инструкционный адрес после BLX) +1 с ссылкой pc = накладка, T =0 p c=R m &0xfffffe, T=R m [0] В отличие от версии ARM, Thumb BXinstruction не может быть выполнена условно. Пример 4.1Этот пример показывает небольшой фрагмент кода, который использует как ARM, так и Thumb версии BXinstruction. Вы можете видеть, что адрес ветки в Thumb имеет самый низкий бит. Это устанавливает Tbit в cpsr на состояние Thumb. Скорее код устанавливает адрес возвращения изреченно с использованием MOV инструкции до ветви: ; ARM код CODE32 ; словосочетание LDR r0, =thumbCode+1 ; +1 для ввода Дюйм-состояния MOV lr, pc ; установить адрес возвращения BX r0 ; ветви к Дюйм-коду & режим ; продолжать здесь ; Дюйм-код CODE16 ; полусловное выравнивание Дюйм-кода ADD r1, #1 BX lr ; возвращение к ARM-коду & состояние Вветвизная инструкция обмена также может быть использована в качестве абсолютной ветви , обеспечивающей бит 0 не используется для принуждения к изменению состояния: ; c(дюйм-код) = 0x00010000 ; rpsr = rcvqIFt\_SVC0000000000 ; r0x000000 = 0x000000 0x000000 LDR0, rx000000 rth = ccvcodec1cvcvcvcvcvcvcvcx0000 r0x000000 r0x0000;

92 Глава 4 Введение набор инструкций по большому пальцу; cpsr = nzcvqIFT\_SVC; r0 = 0x00010001; pc = 0x00010000 Вы можете видеть, что наименее значимый бит реестра r0 используется для установления Tbit cpsr. cpsr изменяется от IFt, до выполнения BX, t oIFT, после выполнения. ■ Пример 4.2Замена инструкции BX с BLX упрощает вызов рутины Thumb, поскольку она устанавливает адрес возвращения в реестре ссылок lr: CODE32LDR r0, =thumbRoutine+1 ; введите состояние ThumbBLX r0 ; перейдите к код Thumb; продолжите здесь CODE16 thumbRoutine ADD r1, #1 BX r14 ; возвращайтесь к коду ARM и состоянию ■ 4.3 Другие инструкции к ветви Первый аналогичен версии ARM и выполняется условно; диапазон ветвей ограничен подписанным 8-битным непосредственным, или -256 - +254 байтами. Вторая версия удаляет условную часть инструкции и расширяет действенный диапазон ветвей до подписанного 11-битного непосредственного, или -2048 до +2046 байтов. Условное ветвейное указание является единственным условным выполненным инструкцией в состоянии Дюйма. Синтаксис: B<вторых> накладка B накладка BL накладка B ветвейная pc = накладка BL ветвей с с ссылкой pc = накладка lr =(адрес инструкции после BL) + 1 TheBLinstruction не выполняется условным образом и имеет приблизительный диапазон +/−4 MB. Этот диапазон возможен, потому что BL(и BLX) инструкции переводятся в пару 16-битных

4.4 Инструкции по обработке данных 93 Инструкции по большому пальцу. Первая инструкция в паре содержит верхнюю часть отсека ветви, а вторая - низкую часть. Эти инструкции должны использоваться в паре. Код здесь показывает различные инструкции, используемые для возвращения из BL-суб-противообразования: MOV pc, lr BX lr POP {pc} Для возвращения мы устанавливаем значение pcto в lr. Инструкция по обработке данных, называемая POP, будет обсуждаться подробно в разделе 4.7. 4.4 Инструкции по обработке данных Инструкции по обработке данных манипулируют данными в реестрах. Инструкции по обработке данных ARM являются подмножеством инструкций по обработке данных. Синтаксис: <ADM в том, что он есть> Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, Rd, R

94 Глава 4 Введение в наборы указаний на большую часть и логические биты и двух 32-битных значениях R d=R d& Rm ASR 32-битных 32-битных наборов Rn + Rm ASR 32-битных наборов сдвига права R d=R m /greatermuchimmediate, C flag =R m [immediate −1] R d=R d/greatermuch Rs, C flag=R d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d d

4.4 Инструкции по обработке данных 95 Эти инструкции следуют тому же стилю, что и эквивалентные инструкции ARM. Большинство инструкций по обработке данных Thumb работают на низких регистрах и обновляют cpsr. Исключениями являются MOV Rd,Rn ADD Rd,RmCMP Rn,Rm ADD sp, #immediate SUB sp, #immediateADD Rd,sp,#immediateADD Rd,pc,#immediate, которые могут работать на более высоких регистрах r8r14 и компьютерах. Эти инструкции, за исключением CMP, не обновляют знаки состояния в cpsr при использовании более высоких реестров. Инструкция CMP, однако, всегда обновляет cpsr. Пример 4.3В этом примере показывается простое указание Thumb ADD. Он берет два низких реестра r1 и r2 и добавляет их вместе. Результат затем помещается в реестр r0, переписывая оригинальное содержание. Cpsr также обновляется. PREcpsr = nzcvIFT\_SVC r1 = 0x80000000 r2 = 0x10000000 ADD r0, r1, r2 POSTr0 = 0x90000000 cpsr = NzcvIFT\_SVC ■ Пример 4.4Thumb отклоняется от стиля ARM, поскольку операции с сдвига ведра (ASR, LSL, LSR и ROR) являются отдельными инструкциями. В этом примере показано инструкция логического левого смещения (LSL) для умножения реестра r2by 2. PREr2 = 0x00000002 r4 = 0x00000001 LSL r2, r4 POSTr2 = 0x00000004 r4 = 0x00000001 ■ Для полного перечня инструкций по обработке данных по пальцам см. Приложение А.

96 Глава 4 Введение набор инструкций по большому пальцу 4.5 Инструкции по единому регистру для грузовых хранилищ Набор инструкций по большому пальцу поддерживает регистрацию грузовых и хранилищ, или LDR и STR. Эти инструкции используют два предварительного режима адресования: офорс по регистру и офорс по немедленному. Синтаксис: <LDR та STR>{<B уйеH>} Rd, [Rn, #immediate] LDR{<HССБШ>} Rd, Rn,Rm] STR{<BH>} Rd, Rn,Rm] LDR Rd, Rpc, #immediate] <LDR STR> Rd, #immediate] LDR загружает слово в регистр Rd<-mem32[адрес] STR загружает слово из регистра загрузки Rd-mem-> 4.address[адрес] LDRB загружает слово в регистр Rd-mem8[address] STRB загружает слово из регистра Rd-mem8[address] Rd[address] LDRD[address] LDRD[address] LDRD[address] LDRD[address] LDRD[dress] LDRD[dress] LDRD[dress] LDRD[dress] LDRD[dress] LDRD[dress] LDRDress] LDRD[dress] LDRDress] LDRD[dress] LDRDressdressdressdressdressdressdressdressdressdressdressdressdressdressdressdressdressdressdressdressdressdressdressdressdressdressdressdressdressdressdressdressdressdressdressdressdressdressdressdressdressdressdressdressdressdressdressdressdressdressdressdressdressdressdressdressdressdressdressdressdressdressdressdressdressdressdressdressdressdressdressdressdressdressdressdressdressdressdressdressdressdressdressdressdressdressdressdressdressdressdressdressdressdressdressdressdressd Оффес по регистру используется базовый регистр Rn плюс реестр оффес Rm. Второй использует тот же базовый регистр Rn плюс 5-битовый непосредственный или значение, зависящее от размера данных. 5-битное оформяние, кодированное в инструкции, умножается на один для байтных доступа, два для 16-битных доступа и четыре для 32-битных доступа. Таблица 4.3 Режим адресования.

4.6 Инструкции по многократному регистрации загрузки-магазина 97 PRE mem32[0x90000] = 0x00000001 mem32[0x90004] = 0x00000002 mem32[0x90008] = 0x00000003 r0 = 0x00000000r1 = 0x00090000r4 = 0x00000004 LDR r0, [r1, r4]; регистрация POST r0 = 0x00000002 r1 = 0x00090000 r4 = 0x00000004 LDR r0, [r1, #0x4]; непосредственная POST r0 = 0x00000002 Обе инструкции выполняют одну и ту же операцию. Единственное отличие заключается в том, что вторая LDR использует фиксированный офсет, тогда как первая зависит от значения в регистре r4. ■ 4.6 Инструкции для многочисленных регистрационных загрузок. Они поддерживают только режим обращения с (IA) увеличением. Синтаксис: <LDMSTM>IA Rn!, {низкий список реестра} LDMIA загружает несколько реестров {Rd}\*N<-mem32[Rn + 4N], Rn=Rn + 4N STMIA сохраняет несколько реестров {Rd}\*N-> mem32[Rn + 4N], Rn=Rn + 4N Здесь ниже количество реестров в списке реестров. Вы можете видеть, что эти инструкции всегда обновляют базовый реестр Rnafter исполнение. Базовый реестр и список реестров ограничены низкими реестрами r0tor7. Пример 4.6В этом примере хранятся реестры r1tor3в память от 0x9000 до 0x900c. Он также обновляет базовый реестр r4. Обратите внимание, что знак обновления !не является опцией, в отличие от набора инструкций ARM. PREr1 = 0x00000001 r2 = 0x00000002

Глава 4 Введение в набор инструкций по большому пальцу r3 = 0x00000003 r4 = 0x9000 STMIA r4!,{r1,r2,r3} POSTmem32[0x9000] = 0x00000001 mem32[0x9004] = 0x00000002 mem32[0x9008] = 0x00000003 r4 = 0x900c ■ 4.7 Инструкции по стеку Операции по большому пальцу отличаются от эквивалентных инструкций ARM, потому что они используют более традиционную концепцию POP и PUSH. Синтаксис: POP {low\_register\_list{, pc}} PUSH {dlow\_register\_list {, lr}} POP-регистры из стеков RdN<, lr}} POP-регистры из стеков RdN<, lr32[sp4p N], sp=sp4p N + PN инструкции на стеках RdN-N + n n n n n n n n n n n n n n n n n n n n n n n n n n n n n n n n n n n n n n n n n n n n n n n n n n n n n n n n n n Это связано с тем, что указатель стека фиксируется как реестр r13 в операции Thumb и spis автоматически обновляется. Список реестра ограничивается низкими реестрами r0tor7. Список реестра PUSH может также включать реестр ссылок lr; аналогично список POP реестра может включать компьютер. Это обеспечивает поддержку для входа и выхода субрутин, как показано в Примере 4.7. Инструкции стека поддерживают только полные операции спускающейся стек. Примером 4.7В этом примере мы используем инструкции POP и PUSH. Подпрограмма ThumbRoutine называется с помощью ветви с инструкцией ссылки (BL). ; Назовите подпрограмма BL ThumbRoutine; продолжайте ThumbRoutine PUSH {r1, lr}; введите подпрограмма MOV r0, #2 POP {r1, pc}; возвращайтесь из подпрограмма

4.8 Инструкция по прерыванию программного обеспечения 99 Инструкция по прерыванию программного обеспечения В качестве аналога ARM инструкция по прерыванию программного обеспечения (SWI) вызывает исключение прерывания программного обеспечения. Если в состоянии Thumb поднимается какой-либо знак прерывания или исключения, процессор автоматически возвращается в состояние ARM, чтобы обрабатывать исключение. Синтаксис: SWI непосредственное SWI программное обеспечение прерывает lr\_svc = адрес инструкции после SWI spsr\_svc = cpsr pc = векторы + 0x8 cpsr режим = SVC cpsr I = 1 (маска IRQ прерывает) cpsr T = 0 (состояние ARM) Инструкция SWI Thumb имеет тот же эффект и почти тот же синтаксис, что и эквивалент ARM. Это отличается тем, что число SWI ограничено диапазоном от 0 до 255 и оно не выполняется условно. Пример 4.8В этом примере показано выполнение инструкции Thumb SWI. Обратите внимание, что процессор переходит от состояния Thumb к состоянию ARM после выполнения. PRE cpsr = nzcVqifT\_USER pc = 0x00008000 lr = 0x003fffff; lr = r14 r0 = 0x12 0x00008000 SWI 0x45 POSTcpsr = nzcVqIft\_SVC spsr = nzcVqifT\_USER pc = 0x00000008 lr = 0x00008002r0 = 0x12 ■

100 Глава 4 Введение набор инструкций по пальцу 4.9 Резюме В этой главе мы рассмотрели набор инструкций по пальцу. Все инструкции по пальцу имеют длину 16 бит. В Интерверкинг используется инструкция по обмену ветвями (BX) и инструкция по обмену ветвями с ссылкой (BLX) для изменения состояния и перехода на определенную рутину. В Thumb выполняются только инструкции по ветвям условно. Операции смещения ветвей (ASR, LSL, LSR и ROR) являются отдельными инструкциями. Набор инструкций "Тумп" включает в себя инструкции POP и PUSH в качестве операций на стеке. Эти инструкции поддерживают только полный нисходящий стек.

Эта страница умышленно осталась пустой

5.1 Обзор C компиляторов и оптимизации 5.2 Основные типы данных C 5.2.1 Местные переменные типы 5.2.2 Типы аргумента функции5.2.3 Подписанный против неподписанного типы 5.3 C петляные структуры 5.3.1 петли с фиксированным числом итераций5.3.2 петли с использованием переменного числа итераций 5.3.3 петля разворачивание 5.4 Регистрационное распределение 5.5 Функция вызовы 5.6 Поискание 5.7 Структурное устройство 5.8 Битовые поля 5.9 Несоответствующие данные и точность 5.10 Раздел 5.10.1 Повторяемое несоответствующее разделение с остатками 5.10.2 Преобразование разделений в множества5.10.3 Несоответствующее разделение постоянным 5.10.4 Подписанное постоянным 5.11 Пункт 5.12 Инлайн-функции и портативность 5.13 Подведение итогов 5.14

Глава Эффективное программирование C5. Цель этой главы состоит в том, чтобы помочь вам написать код C в стиле, который будет эффективно компилировать на архитектуре ARM. Мы посмотрим на многие небольшие примеры, чтобы показать, как компилятор переводит C-источник на сборщик ARM. Как только вы получите ощущение этого процесса перевода, вы можете отличить быстрый код C от медленного кода C. Техники применяются также и к C++, но мы будем придерживаться простого C для этих примеров. Мы начнем с обзора компиляторов C и оптимизации, который даст представление о проблемах, с которыми сталкивается компилятор C при оптимизации вашего кода. В разделах 5.2 и 5.3 рассматривается, как оптимизировать базовую циркулл C. Эти разделы используют сумму проверки пакета данных в качестве простого примера для иллюстрации идей. В разделах 5.4 и 5.5 рассматривается оптимизация всего корпуса функции C, включая то, как компилятор распределяет реестры в рамках функции и как уменьшить общие затраты на вызов функции. В разделах 5.6 - 5.9 рассматриваются проблемы с памятью, включая обработку указателей и эффективное упаковку данных и доступ к памяти. В разделах 5.10 - 5.12 рассматриваются основные операции, которые обычно не поддерживаются непосредственно инструкциями ARM. Вы можете добавить свои собственные базовые операции с использованием встроенных функций и монтажа.

Глава 5 Эффективное программирование C 5.1 Обзор компиляторов и оптимизации C Эта глава предполагает, что вы знакомы с языком C и имеете некоторое знание об программировании сборки. Последнее не является необходимым, но полезно для примера выхода компилятора. См. Глава 3 или Приложение A для подробностей синтаксиса сборки ARM. Обычно, это только стоит оптимизировать функции, которые часто выполняются и важны для производительности. Мы рекомендуем вам использовать инструмент профилирования производительности, найденный в большинстве симуляторов ARM, чтобы найти эти часто выполняемые функции. Документируйте неочевидные оптимизации с помощью комментариев исходного кода для поддержки поддержания. Компиляторы C должны буквально перевести вашу функцию C в сборщик, чтобы она работала для всех возможных входов. Функция memclr очищает Nbytes памяти на адресной информации. пусто memclr(char \*data, int N) { для (; N>0; N--) { \*data=0; data++; } Независимо от того, насколько продвинутый компилятор, он не знает, может ли N быть 0 на входе или нет. Поэтому компилятору необходимо проверить этот случай прямо перед первой итерацией петли. Компилятор не знает, является ли указатель массива данных четверобайтным или нет. Если он четверобайтным, то компилятор может очистить четыре байта одновременно с использованием интстора, а не карты. Если Nis кратный из четырех, то компилятор может повторять цепь тела четыре раза или хранить четыре байта одновременно с использованием хранилища int. Компилятор должен быть консервативным и предполагать все возможные значения для Nand все возможные подровнения для данных. В разделе 5.3 подробно рассматриваются эти конкретные моменты. Для того, чтобы написать эффективный код C, вы должны знать области, где компилятор C должен быть консервативным, границы архитектуры процессоров, к которым компилятор C использует карту, и границы конкретного компилятора C. Большая часть этой главы охватывает первые две точки выше и должна быть применима к любому компилятору ARM C. Третий пункт будет очень зависеть от поставщика компилятора и от пересмотра компилятора.

5.2 Основные типы данных C 105 Чтобы сохранить конкретные примеры, мы проверили их с использованием следующих конкретных компиляторов C: ■armcc из версии 1.1 ARM Developer Suite (ADS1.1). Это ARM-цель для компилятора GNU C, gcc, и доступен бесплатно. Мы использовали armcc из ADS1.1 для генерации выхода примерного сборщика в этой книге. Вы можете использовать это для воспроизведения наших примеров. armcc -Otime -c -o test.o test.c fromelf -text/c test.o > test.txt По умолчанию armcc имеет полную оптимизацию включенной (сменщик командной строки -O2). Если вы используете gcccompiler, то следующий короткий сценарий генерирует аналогичный список выхода сборщика: arm-elf-gcc -O2 -fomit-frame-pointer -c -o test.o test.c arm-elf-objdump -d test.o > test.txt Полное оптимизацию отключают по умолчанию для компилятора GNU. Сменщик указателя -fomit-frame- препятствует компилятору GNU поддерживать реестр указателей кадров. Показы кадров помогают в восстановлении дебгу, указывая на локальные переменные, хранящиеся на стековой раме. Однако они неэффективны для поддержания и не должны использоваться в критических для производительности кодов. 5.2 Основные типы данных C Давайте начнем с того, как компиляторы ARM обрабатывают основные типы данных C. Мы увидим, что некоторые из этих типов более эффективны для использования для локальных переменных, чем другие. Существуют также различия между режимами адресования, доступными при загрузке и хранении данных каждого типа. Другими словами, вы должны загружать значения из памяти в реестры, прежде чем действовать на них.Нет никаких арифметических или логических инструкций, которые могли бы непосредственно манипулировать значениями в памяти.

106 Глава 5 Эффективная C Таблица программирования 5.1 Загрузка и хранение инструкций по архитектуре ARM. Инструкция Действие предварительного ARMv4 LDRB загрузка неподписанного 8-битного значения STRB хранилище подписанного или неподписанного 8-битного значения LDR загрузка подписанного или неподписанного 32-битного значения STR хранилище подписанного или неподписанного 32-битного значения ARMv4 LDRSB загрузка подписанного 8-битного значения LDRH загрузка неподписанного 16-битного значения LDRSH загрузка подписанного 16-битного значения STRH хранилище подписанное или неподписанное 16-битное значение ARMv5 LDRD загрузка подписанного или неподписанного 64-битного значения STRD хранилище подписанного или неподписанного 64-битного значения Эти архитектуры использовались на процессе до ARM7TDMI. В таблице 5.1 показаны классы инструкций нагрузки/устройства, доступные для архитектуры ARM. В таблице 5.1 нагрузки, действующие на 8- или 16-битные значения, расширяют значение до 32 бит, прежде чем записывать в регистр ARM. Неподписанные значения имеют нулевую расширение, а подписанные значения - знак-расширение.Это означает, что загрузка загруженного значения к типу int не стоит дополнительных инструкций.Таким же образом, хранилище с 8- или 16-битовым значением выбирает самые низкие 8 или 16-биты реестра. Выпуск инт на более маленький тип не стоит дополнительных инструкций в магазине. Архитектура ARMv4 и поддержка выше подписывают 8-битные и 16-битные загрузки и магазины напрямую, через новые инструкции. (Смотрите раздел 3.3 для подробности различных режимов адресования.) Мы увидим эффект этого в примере\_checksum\_v3 в раздел 5.2.1. Наконец, ARMv5 добавляет поддержку инструкций для 64-битной нагрузки и хранилищ. Поэтому компиляторы ARM C определяют char как неподписанное 8-битное значение, а не подписанное 8-битное значение, как это типично во многих других компиляторах. Компиляторы ARM и GCC используют картографию типа данных в таблице 5.2 для цели ARM. Обычный пример - использование переменной типа char, как счетчика петли, с условием продолжения петли i≥0. Если для компиляторов ARM петля не подписана, она никогда не закончится. К счастью, armcc в этой ситуации создает предупреждение: не подписанное сравнение с 0 . Компиляторы также предоставляют переключатель перемены для подписания. Например, опция командной строки -fsigned-char сделает char подписанным на gcc. Обязатель командной строки -zc будет иметь тот же эффект, что и с armcc. Для остальной части этой книги мы предполагаем, что вы используете процессор ARMv4 или выше.

5.2 Основные типы данных C 107 Таблица 5.2 C компиляторные картографии типы данных. C Тип данных Implementation char неподписанный 8-битный короткий подписанный 16-битный полуслово int подписанный 32-битный слово длинный подписанный 32-битный слово длинный подписанный 64-битный двойной слово 5.2.1 Местные переменные типы ARMv4 процессоры на основе эффективно загружают и хранят 8-битные и 32-битные данные. Однако большинство операций обработки данных ARM составляют только 32-битные. По этой причине, вы должны использовать 32-битовый тип данных, intorlong, для локальных переменных, где это возможно. Избегайте использования char и short как локальные переменные типы, даже если вы манипулируете 8- или 16-битовым значением. Если вам нужна модульная арифметика формы 255 +1=0, то используйте тип char. Чтобы увидеть эффект локальных типов переменных, давайте рассмотрим простой пример. Большинство протоколов связи (например, TCP/IP) имеют проверку суммы или проверку циклической избыточности (CRC) для проверки ошибок в пакете данных. Он показывает, почему вы должны избегать использования char для местных переменных. int checksum\_v1(int \*data) { char i; int sum=0; for (i=0; i<64; i++) { sum += data[i]; } return sum; } На первый взгляд кажется, что объявление ias achar эффективно. Все ARM-регистры 32-битные, а все записи на стеке - не менее 32-битные. Кроме того, чтобы точно реализовать i++, компилятор должен учесть случай, когда i=255. Любая попытка увеличить 255 должно дать ответ 0.

Глава 5 Эффективное C-программирование Рассмотрим выход компилятора для этой функции. Мы добавили этикетки и комментарии, чтобы сделать сборку ясной. checksum\_v1 MOV r2,r0 ; r2 = данные MOV r0,#0 ; сумма = 0 MOV r1,#0 ; i=0 checksum\_v1\_loop LDR r3,[r2,r1,LSL #2]; r3 = данные[i] ADD r1,r1,#1 ; r1 = i+1 И r1,r1,#0xff ; i = (char) r1 CMP r1,#0x40 ; сравнить i, 64 ADD r0,r3,r0 ; сумма += r3 BCC checksum\_v1\_loop; если (i<64) MOVloop,r14 ; сравнить сумму компилятора с суммой выхода, где мы вместо этого декомпинируем i\_an\_ium. В следующем случае, предположим, что пакет данных содержит 16-битные значения, и нам нужна 16-битная чекс-сумма. Вызывает соблазн написать следующий код C: short checksum\_v3(short \*data) { незаписанный int i; short sum=0; для (i=0; i<64; i++) { sum = (short)(sum + data[i]);

5.2 Basic C Data Types 109 } return sum; } Вы можете задаться вопросом, почему в for loop body не содержится код sum += data[i]; с armcc этот код будет производить предупреждение, если вы включите немыслимые предупреждения с помощью коммутатора компилятора -W+n . Выражение sum+data[i] является целым числом и поэтому может быть присвоено только короткому с помощью (немыслимого или явного) сжатия. Как вы можете видеть в следующем выходе сборки, компилятор должен вставить дополнительные инструкции для реализации узкого каста: checksum\_v3 MOV r2,r0 ; r2 = данные MOV r0,#0 ; сумма = 0 MOV r1,#0 ; i=0 checksum\_v3\_loop ADD r3,r2,r1,LSL #1 ; r3 = &data[i] LDRH r3,[r3,#0] ; r3 = данные [i] ADD r1,r1,#1 ; i++ CMP сравнить r1,#0x40 ; i, 64 ADD r0,r3,r0 ; r0 = + r3 MOV r0,r0,LSL #16MOV r0,r0,R #16 ; sum = (shortum) r0 BCC\_loop checks (v3,v64) ; если сумма циркулы ранее, чем, например, 3 циркулы, возвращаются, если сумма циркулы ранее, чем, например, r14 r2 Существуют две причины для дополнительных инструкций: ■Инструкция LDRH не позволяет смещать адресную смесь, как это делалось с инструкцией LDR inchecksum\_v2. Поэтому первая ADDin цикл рассчитывает адрес пункта i в массиве. (Смотрите таблицу 5.1.) ■Двух инструкций по движению, необходимых для резки, уменьшающей общее количество+арей[i] до сокращения. Компилятор смещается налево на 16 и затем на право на 16 для реализации 16-битного раздела знака. Смена справа - это раздел расширения знака, поэтому он воспроизводит бит знака, чтобы заполнить верхние 16 бит. Мы можем избежать второй проблемы, используя переменную типа int для удержания частичной суммы. Мы только уменьшаем сумму до короткого типа на выходе функции.

110 Глава 5 Эффективное программирование C Однако первая проблема является новой. Мы можем решить ее, получив доступ к массиве путем увеличения показательных данных, а не использования индекса, как в данных[i]. Это эффективно независимо от размера типа массива или размера элемента. Все инструкции по загрузке и хранению ARM имеют режим адресования постинкремента. Пример 5.1Thechecksum\_v4 код исправляет все проблемы, которые мы обсудили в этом разделе. Он использует местные переменные типа int, чтобы избежать ненужных выбросов. Он увеличивает показательные данные вместо использования индекса свертываемых данных[i]. короткий checksum\_v4(short \*data) { незаписанный int i; int sum=0; для (i=0; i<64; i++) { сумма += \*(данные++); }return (short)sum; }\*(данные++) операция переводится в одну инструкцию ARM, которая загружает данные и увеличивает показатель данных. Конечно, вы можете написать сумму += \*данные; данные++; или даже \*++данные вместо этого, если вы хотите. Компилятор производит следующий выход. Из внутренней петли были удалены три инструкции, сберегая три цикла в петле по сравнению с checksum\_v3. checksum\_v4 MOV r2,#0; сумма = 0 MOV r1,#0; i=0 checksum\_v4\_loop LDRSH r3,[r0],#2; r3 = \*(++) ADD r1,r1,#1 ; i++ CMP r1,#0x40 ; сравнивайте i, 64 ADD r2,r3,r2 ; сумма += r3 BCC checksum\_v4\_; если (сумма <64) gotoLS loopMOV r0,r2,l #16MOV r0,r0,ASR #16 r0 = (краткая сумма) MOV, pcr14 return r0 The compiler is still performing one cast to return range, on the function 16-bit. Вы можете удалить это также, вернув результат int, как обсуждается в разделе 5.2.2. ■

5.2 Основные типы данных C 111 5.2.2 Типы аргументов функций Мы видели в разделе 5.2.1 что преобразование локальных переменных из типов char илиshort в тип int повышает производительность и уменьшает размер кода. То же самое относится и к аргументам функций. Рассмотрим следующую простую функцию, которая добавляет два 16-битных значения, уменьшая на половину второе и возвращает 16-битную сумму: короткое add\_v1(short a, короткое b) { return+( b > > 1 ) ; } Эта функция немного искусственная, но это полезный тест-кас для иллюстрации проблем, с которыми сталкивается компилятор. Входное значение a,b, а возвращаемое значение будет передано в 32-битных ARM реестрах. Должен ли компилятор предположить, что эти 32-битовые значения находятся в диапазоне ашортного типа, то есть -32 768 до +32 767? Или должен компилятор заставить значения находятся в этом диапазоне, увеличив наименьшие 16 бит для заполнения 32-битового реестра? Или звонка или звонка должны выполнять выброс на короткий тип. Мы говорим, что аргументы функции передаются широко, если они не сокращены до диапазона типа и узко, если они есть. Вы можете узнать, какое решение сделал компилятор, глядя на выход сборки для add\_v1. Если компилятор передает аргументы узко, то вызовитель должен сократить диапазон. Если компилятор возвращает значения в целом, то вызовитель должен сократить значение возвращения до правильного диапазона. Если компилятор возвращает значения узкого, то calllee должен сократить диапазон перед возвращением значения. Forarmcc в ADS, аргументы функции проходят узким и значения возвращаются узким. Другими словами, звончик бросает значения аргумента и calllee бросает значения возвращения. Компилятор использует прототип ANSI функции для определения типов данных аргументов функции. Выход armcc для add\_v1 показывает, что компилятор выбросает значение возврата на короткий тип, но не выбросает значения ввода. Он предполагает, что звон уже обеспечил, что 32-битные значения r0andr1 в диапазоне короткого типа. Это показывает узкое прохождение аргументов и значение возврата. add\_v1 ADD r0,r0,r1,ASR #1 ; r0 = (int) a + ((int) b>>1) MOV r0,r0,LSL #16 MOV r0,r0,ASR #16 ; r0 = (short) r0MOV pc,r14 ; return r0 Использованный gcccompiler более осторожен и не делает никаких предположений о диапазоне значения аргумента. Эта версия компилятора уменьшает входные аргументы до диапазона

112 Глава 5 Эффективное C Программирование короткого слова как в звонке, так и в звонке. Он также выбросает значение возврата на короткий тип. Вот составленный код для add\_v1: add\_v1\_gcc MOV r0, r0, LSL #16 MOV r1, r1, LSL #16 MOV r1, r1, ASR #17; r1 = (int) b>>1 ADD r1, r1, r0, ASR #16; r1 += (int) aMOV r1, r1, LSL #16MOV r0, r1, ASR #16; r0 = (short) r1 MOV pc, lr ; return r0 Независимо от достоинств различных узких и широких протоколов звонка, вы можете видеть, что аргументы или значения типа возврата функции или возврата вводят дополнительные касты. 5.2.3 Подписанные против неподписанные Типы Предыдущие разделы демонстрируют преимущества использования типа intr вместо char или short для локальных переменных и аргументов функций. В этом разделе сравнивается эффективность подписанного и неподписанного. Если код использует добавление, удаление и умножение, то нет разницы в производительности между подписанными и неподписанными операциями. Рассмотрим следующий короткий пример, который имеет в среднем два целых числа: int average\_v1(int a, int b) { return (a+b) /2; } Это компилирует в среднем\_v1 ADD r0,r0,r1 ; r0=a+b ADD r0,r0,r0,LSR #31; если (r0<0) r0++ MOV r0,r0,ASR #1 ; r0 = r0>>1MOV pc,r14; return r0 Обратите внимание, что компилятор добавляет один к сумме перед перемещением по правую сторону, если сумма отрицательна. Другими словами, он заменяет x/2 утверждением: (x<0) ? (x+1)>>1): (x>>1)

5.3 C Looping Structures 113 Это должно быть сделано потому, что xis подписан. В C на ARM-целе, разделение на два не является правым сдвигом, если xis отрицательный. Например, −3/greatermuch1=−2 но −3/2=−1. Для общих разделений, разделение в библиотеке C быстрее для неподписанных типов.Смотрите раздел 5.10 для обсуждения об избежании разделений полностью.Обзор Эффективное использование типов C ■Для местных переменных, хранящихся в реестрах, не используйте диаграммный или короткий тип, если не требуется 8-битовая или 16-битовая модульная арифметика. Используйте вместо этого подписанные или неподписанные типы. Неподписанные типы быстрее, когда вы используете разделы. ■Для записей массива и глобальных переменных, хранящихся в основной памяти, используйте тип с наименьшим возможным размером для хранения требуемых данных. Это экономит отпечаток памяти. Архитектура ARMv4 эффективна при загрузке и хранении всех ширины данных при условии, что вы пересекаете массивы путем увеличения показателя массива. Избегайте использования оффсетов из основы массива с короткими массивами типа, поскольку LDRH не поддерживает это. ■Используйте эксплюсивные касты при чтении записей массива или глобальных переменных в локальные переменные или при написании локальных переменных в записи массива. Включить в компилятор предупреждения об уменьшении немыслимого в виде кастинга для обнаружения немыслимых кастингов. ■Об избежание немыслимых или немыслимых уменьшения в выражениях, поскольку они обычно стоят дополнительных циклов. Вместо этого используйте тип int, даже если диапазон параметра меньше. Это предотвращает выполнение компилятором ненужных кастов. 5.3 C Looping Structures В этом разделе рассматриваются наиболее эффективные способы кодирования и изменения во время циркуляций на ARM. Мы начинаем с рассмотрения циркуляций с фиксированным числом итераций, а затем переходим к циркуляциям с переменным числом итераций. 5.3.1 Лопушки с фиксированным числом итераций Какой самый эффективный способ написать лопушку на ARM?

114 Глава 5 Эффективное программирование C Вот последняя версия 64-словного пакета-счетного плана, который мы изучали в разделе 5.2. Это показывает, как компилятор обрабатывает петлю с увеличивающимся количеством i++. int checksum\_v5(int \*data) { неподписанный int i; int sum=0; для (i=0; i<64; i++) { сумма += \*(данные++); } сумма возврата; } Это компилирует к checksum\_v5 MOV r2,r0 ; r2 = данные MOV r0,#0 ; сумма = 0 MOV r1,#0 ; i=0 checksum\_v5\_loop LDR r3,[r2],#4 ; r3 = \*data++) ADD r1,r1,#1 ; i++ CMP r1,#0x40 ; сравнить i, 64 ADD r0,r3,r0 ; сумма += r3 BCC checksum\_v5\_\_r; It (<64) gotoVV, return, i14 , если эта сумма для выполнения цикл, если эта структура не менее эффективная, чем для проверки, если i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i На ARM петля должна использовать только две инструкции: ■Вычитать для декремента счетчика петли, который также устанавливает знаки кода условия на результат ■Условное указание ветви

5.3 C Структуры 115 в состоянии флагов. Пример 5.2Этот пример показывает улучшение, если мы переходим на декрементирующую петлю, а не на инкрементирующую петлю. int checksum\_v6(int \*data) { неподписанная int i; int sum=0; для (i=64; i!=0; i--) { сумма += \*(data++); }return sum; } Это компилирует для проверки sum\_v6 MOV r2,r0 ; r2 = data MOV r0,#0 ; sum = 0 MOV r1,0x40 ; i=6 4 checksum\_v6\_loop LDR r3,[r2],#4 r3 \*SUBS = \*data\_loop) RMO1,r1,r1 ,i; i и set rBSd0,rbv0 ,rbv0; RBSd0,rb0 + rb0; bssd0 ,r3 ,rb0; r6 ,rb0; r6 ,rb0; r6 ,rb0; rb0; rb0; r6 ,rb0; rb0; rb0; rb0; rb; rb0; rb; rb; rb; r6 Наше пример чексум теперь имеет минимальное количество четырех инструкций на петлю. Это намного лучше, чем шесть для чексум\_в1 и восемь для чексум\_в3. ■ Для незаписанного циклного счетчика iwe может использовать любое из условий продолжения цикла i!=0 ori>0. Вы можете ожидать, что компилятор будет генерировать следующие две инструкции для реализации петли: SUBS r1,r1,#1; сравнить i с 1, i=i-1 BGT петли; если (i+1>1) goto петли

116 Глава 5 Эффективное программирование C Фактически компилятор будет генерировать SUB r1, r1, # 1; i-- CMP r1, # 0; сравнить i с 0BGT петлю; если (i>0) goto петля Компилятор не является неэффективным. Следует быть осторожным в случае, когда i = -0x80000000, потому что два раздела кода генерируют разные ответы в этом случае. Для первого куска кода инструкция SUBS сравнивает i с 1 и затем с уменьшениями i. Поскольку 0x80000000 < 1, петля заканчивается. Для второго куска кода мы уменьшаем i и затем сравниваем с 0. Модульное арифметическое значение означает, что у inow есть значение +0x7fffffff, которое больше нуля. Таким образом, петля продолжается во многих итерациях. Компилятор не может обычно определить это, особенно если цикл начинается с переменного числа итераций (см. Раздел 5.3.2). 5.3.2 петли с использованием переменного числа итераций Теперь предположим, что мы хотим, чтобы наша схема чекс-сум обрабатывала пакеты произвольного размера. Используя уроки из последнего раздела мы подсчитываем до N=0 и не требуем дополнительного счетчика петли i. Пример checksum\_v7 показывает, как компилятор обрабатывает петлю for с переменным числом итераций N. int checksum\_v7(int \*data, неподписанная int N) { int sum=0; for (; N!=0; N--) { sum += \*(data++); }return sum; } Это компилирует для checksum\_v7 MOV r2,#0; sum = 0 CMP сравнить r1,#0 ; N, 0 BEQ checksum\_v7\_end; если (N==0) goto end

5.3 C Looping Structures 117 checksum\_v7\_loop LDR r3,[r0],#4 ; r3 = \*(data++) SUBS r1,r1,#1 ; N-- и установленные флаги ADD r2,r3,r2 ; сумма += r3 BNE checksum\_v7\_loop ; если (N!=0) goto loop checksum\_v7\_end MOV r0,r2 ; r0 = сумма MOV pc,r14 ; возвращение r0 Обратите внимание, что компилятор проверяет, что Nis не нулевая при входе в функцию. Часто эта проверка не нужна, так как вы знаете, что массив не будет пустым. Пример 5.3В этом примере показано, как использовать цепку до-за, чтобы удалить тест для NBeing zero, который происходит в цепке for. int checksum\_v8(int \*data, неподписанная int N){ int sum=0; do { sum += \*(data++); } в то время как (--N!=0); return sum; } выход компилятора теперь checksum\_v8 MOV r2,#0; sum = 0 checksum\_v8\_loop LDR r3,[r0],#4++; r3 = \*data) ; r1,r1,#1BS Nloop и наборы ADD--2,r3,r2 ; sum = r3 B checksum\_v8\_v8 ! (NV0); если выход компилятора теперь является checksum\_v8 MOV r2,0; если выходной цикл R0 = r0,r0 ,r0; если выходной цикл R0 = r0 ,r0 ,r0 ,r0 ,r0 ,r0 ,r0 ,r0 ,r0 ,r0 ,r0rrrr; ■ 5.3.3 Unrolling Loop Мы увидели в разделе 5.3.1 что каждая итерация петли стоит две инструкции в дополнение к корпусу петли: вычитать для уменьшения количества петли и условное ветви.

118 Глава 5 Эффективное C-программирование Мы называем эти инструкции цепной перегрузкой. На процессорах ARM7 или ARM9 вычитать занимает один цикл и ветвь три цикла, давая перегрузку четыре цикла в цепь. Вы можете сохранить некоторые из этих циклов, разворачивая цепь повторяя цепную структуру несколько раз, и уменьшая количество итераций цепь в том же пропорции. Например, давайте развернем нашу схему счета пакетов примерно четыре раза. Мы предполагаем, что число слов в пакете Nis кратное четыре. int checksum\_v9(int \*data, неподписанная int int N) { int sum=0; do { sum += \*(data++); sum += \*(data++); sum += \*(data++); sum += \*(data++); sum += \*(data++);N- =4 ; } в то время как (N!=0); return sum; } Это составляет checksum\_v9 MOV r2,#0; sum = 0 checksum\_v9\_loop LDR r3,[r0],#4 ; r3 = \*(data) SUBS r1,r1,r1,#4 ; N - 4 & set flags r2,r3,r2 ; sum +=3 LDR sum r3,r3 sum [r4],r3 sum; r3 sum \*r3 \*r3 \*r3 \*r3 r3 r3,r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r3 r

5.3 C Лопучие структуры 119 Мы сократили обходный заряд петли с 4 Ncycles до (4 N)/4=Ncycles. На ARM7TDMI это ускоряет петлю с 8 циклов на аккумулятор до 20/4=5 циклов на аккумулятор, почти удвоив скорость! ■ При развертывании петли необходимо задать два вопроса: ■Сколько раз я должен развертывать петлю? ■Что, если количество итераций петли не является кратным количеством развертывания? В противном случае развертывание увеличит размер кода с небольшим выгодой от производительности. развертывание может даже уменьшить производительность, вытесняя более важный код из кеша. Предположим, что петля важна, например, 30% всего приложения. Предположим, что вы развертываете петлю, пока она не достигнет 0,5 КБ в размере кода (128 инструкций). Стоимость обхода петли составляет 3/128, примерно 3%. Примечательно, что цепочка составляет 30% всего приложения, в целом цепочка обхода составляет всего 1%. Обычно не стоит разворачивать дальше, если выигрыш меньше 1%. Если это невозможно, то вам нужно добавить дополнительный код, чтобы позаботиться о оставшихся случаях.Это немного увеличивает размер кода, но поддерживает высокую производительность. Пример 5.5Этот пример обрабатывает сумму проверки любого размера пакета данных с помощью петли, которая была отсчитана четыре раза. int checksum\_v10(int \*data, неподписанный int N) { неподписанный int i; int sum=0; для (i=N/4; i!=0; i--) { сумма += \*(++data); сумма += \*((\*data++); сумма += \*(\*data++++); сумма += \*((data++); } для (i=N&3; i!=0; i) {

120 Глава 5 Эффективная C программа сумма += \*(данные++); } возвращающая сумма; } Вторая для петли обрабатывает оставшиеся случаи, когда Nis не кратный четырех. Обратите внимание, что как N/4 и N&3 могут быть нулевыми, поэтому мы не можем использовать цепи до-за время. ■ Сокращение записывания цепей Эффективно ■ Используйте цепи, которые считаются до нуля. Тогда компилятору не нужно выделять реестр для хранения значения окончания, и сравнение с нулем является бесплатным. ■По умолчанию используйте неподписанные счетчики петли и условие продолжения i!=0 вместо i>0. Это гарантирует, что надлежащая нагрузка петли составляет только две инструкции. ■Пока используете петли, а не выходное, когда вы знаете, что петля будет повторяться хотя бы один раз. Это позволяет компилятору проверить, является ли количество петли нулевым. ■Отбросить важные петли, чтобы уменьшить общую стоимость петли. Не перебросить. Если общая стоимость петли небольшая в процентном отношении к общему, то отброс увеличит размер кода и повредит производительности кеша. ■Попробуйте организовать, чтобы количество элементов в массивах было кратным четырём или восемью. Затем вы можете легко развернуть петли двумя, четырьмя или восемью раза, не беспокоясь о оставшихся элементах массива. 5.4 Регистрационное распределение Компилятор пытается выделить регистр процессора каждой локальной переменной, которую вы используете в функции C. Он попытается использовать один и тот же регистр для различных локальных переменных, если использование переменных не перекрывается. Когда локальных переменных больше, чем доступных реестров, компилятор хранит избыточные переменные на стаке процессора. Для эффективной реализации функции необходимо минимизировать количество переменных, которые были выброшены, и убедиться, что наиболее важные и часто доступные переменные хранятся в реестрах. В таблице 5.3 представлены стандартные имена и использование регистра при использовании стандарта ARM-Thumb-procedure call standard (ATPCS), который используется в коде, генерируемом компиляторами C.

5.4 Регистр распределение 121 Таблица 5.3 C Использование компилятора регистра. Альтернативные имена регистра номеры Регистра АТПЦС регистра использование r0 r1 r2 r3a1a2a3 a4Регистры аргументов. Функция может испортить эти реестры и использовать их в качестве общих реестра скретча в рамках функции. r4 r5 r6 r7 r8v1 v2 v3v4v5 Общие переменные реестры. Функция должна сохранить значения звонков этих реестра. r9 v6 sb Общий переменный реестр. Функция должна сохранить значение звонков этого реестра, за исключением при составлении для независимости позиции чтения и письма (RWPI). Затем r9 содержит статический базовый адрес. Это адрес данных чтения и письма. r10 v7 sl Общий переменный реестр. Функция должна сохранить значение звонка этого реестра, за исключением при составлении с помощью проверки лимита стека. Только старые версии armcc используют указатель кадров. r12 ip Общий реестр скрещивания, который может испортить функцию. Он полезен в качестве реестра скрещивания для функций фанеров или других требований для вызова внутрипроцедуры. r13 sp Показатель стека, указывающий на полную нисходящую стеку. r14 lr Регистр ссылок. На функционном звонке это сохраняет адрес возврата. r15 pc Программный счетчик. При условии, что компилятор не использует проверку программного обеспечения или указатель кадров, тогда компилятор C может использовать реестры r0tor12 и r14 для хранения переменных. Он должен сохранить значения calllee r4tor11 и r14 на стеке, если использовать эти реестры. На практике некоторые компиляторы используют фиксированный реестр, такой как r12 для работы с промежуточным скрачом, и не присваивают переменные этому реестру.

122 Глава 5 Эффективное C-программирование Если компилятору действительно необходимо обменять переменные, то он выбирает, какие переменные обменять на основе частоты использования. Вы можете указать компилятору, какие переменные важны, убедившись, что эти переменные используются в самом внутреннем петле. Однако разные компиляторы рассматривают это ключевое слово по-разному, а разные архитектуры имеют разное количество доступных реестров (например, Thumb и ARM). Компилятор должен быть в состоянии выделить их в ARM-регистры. ■Вы можете указать компилятору, какие переменные важны, обеспечивая использование этих переменных в самом внутреннем цикле. 5.5 Призывы к функциям Стандарт призывов к процедуре ARM (APCS) определяет, как передавать аргументы функции и возвращать значения в ARM-регистрах. В более позднем стандарте ARM-Thumb Procedure Call Standard (ATPCS) также охватывается взаимодействие ARM и Thumb. Первые четыре целых числа проходят в первых четырех регистрах ARM: r0, r1, r2 и r3. В результате, в случае, если вы не можете получить полный номер, то вы можете получить полный номер, если вы не можете получить полный номер. Компилятор может проходить структуры в реестрах или по ссылке в зависимости от вариантов компилятора командной строки.Первый момент, который следует отметить о стандарте процедурного вызова, - это правило четырех реестра. Для функций с большим количеством аргументов, как зватель, так и зватель должны получить доступ к стеку для некоторых аргументов. Если ваша функция C требует более четырех аргументов, или ваш метод C++ более трех явных аргументов, то практически всегда эффективнее использовать структуры.

5.5 Функция называет 123 Аргумент 8... ... Аргумент 7 Аргумент 6 Аргумент 5 Аргумент 4sp + 16 sp + 12 sp + 8 sp + 4 sp Аргумент 3 Аргумент 2 Аргумент 1 Аргумент 0 Возвращение оценщика3 r2 r1 r0 Рисунок 5.1 ATPCS Аргумент прохождения. Следующий пример иллюстрирует преимущества использования указателя структуры. Мы реализуем очередь с использованием циклического буфера с отправной адресой Q\_start (включая) и конечным адресом Q\_end (исключительно). char \*queue\_bytes\_v1(char \*Q\_start, /\* Адрес отправной адресы \*/ char \*Q\_end, /\* Адрес отправной адресы \*/ char \*Q\_ptr, /\* Текущее положение указателя очереди \*/ char \*data, /\* Данные, которые должны быть вставлены в очередь \*/ незаписанный int) /\* Количество байтов, которые должны быть вставлены \*/ { do{ \*(Q\_ptr++) = \*(++++); если (Q\_ptr == Q\_end) { Q\_ptr = Q\_start; } , пока (--N); Возвратить Q\_ptr; };