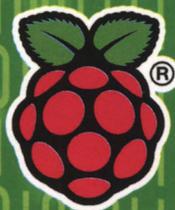


Брюс Смит



АССЕМБЛЕР ДЛЯ RASPBERRY PI

Практическое руководство

4-е издание

Raspberry Pi OS
ASSEMBLY
LANGUAGE

Hands-on-Guide

Fourth Edition

Брюс Смит

АССЕМБЛЕР ДЛЯ RASPBERRY PI

Практическое руководство

4-е издание

Санкт-Петербург

«БХВ-Петербург»

2022

УДК 004.4
ББК 32.973.26-018.2
С50

Смит Б.

С50 Ассемблер для Raspberry Pi. Практическое руководство: Пер. с англ. — 4-е изд.— СПб.: БХВ-Петербург, 2022. — 320 с.: ил.

ISBN 978-5-9775-6801-2

Рассмотрены основы программирования на языке ассемблера для процессоров ARM на примере Raspberry Pi с операционной системой Raspberry Pi OS. Приведены подробные сведения об архитектуре и особенностях ARM, вызовах операционной системы. Подробно описан синтаксис ассемблера для ARM. Рассмотрены компилятор GCC, отладка с GDB, использование функций языка C в ассемблере с помощью библиотеки libc. Описаны функции GPIO, система команд ARM Neop и команды Thumb. Все разделы снабжены практическими примерами. Книга ориентирована на начинающих разработчиков, желающих освоить программирование на языке ассемблера для устройств с архитектурой ARM.

Электронный архив на сайте издательства содержит исходный код программ из книги.

Для начинающих программистов

УДК 004.4
ББК 32.973.26-018.2

Группа подготовки издания:

Руководитель проекта	<i>Павел Шалин</i>
Зав. редакцией	<i>Людмила Гауль</i>
Перевод с английского	<i>Михаила Райтмана</i>
Редактор	<i>Григорий Добин</i>
Компьютерная верстка	<i>Ольги Сергиенко</i>
Оформление обложки	<i>Зои Канторович</i>

Copyright © 2021 by Bruce Smith
Translation Copyright © 2021 by BHV All rights reserved
Перевод © 2021 BHV. Все права защищены.

Подписано в печать 01.12.21
Формат 70×100^{1/16}. Печать офсетная. Усл. печ. л. 35,8
Тираж 1000 экз. Заказ № 2949
"БХВ-Петербург", 191036, Санкт-Петербург, Гончарная ул., 20
Отпечатано с готового оригинал-макета
ООО "Принт-М", 142300, М.О., г. Чехов, ул. Полиграфистов, д. 1

ISBN 978-0-6480987-3-7 (англ.)
ISBN 978-5-9775-6801-2 (рус.)

© Bruce Smith, 2021
© Перевод на русский язык, оформление
ООО "БХВ-Петербург", ООО "БХВ", 2021

Оглавление

Об авторе.....	13
1. Введение	14
Безграничные возможности.....	15
Начинаем экспериментировать	16
Компилятор GNU C.....	16
Учимся на примерах.....	17
Что вы узнаете?.....	17
Совместимость четвертого издания книги	18
ОС Raspberry Pi.....	19
А что насчет 64-разрядной системы?.....	20
Клавиатурные вычисления.....	20
Значимость ARM	20
Raspberry Pi сквозь века	21
Вычислительные модули	23
Используемые обозначения	24
Центр истории вычислительной техники	24
Веб-сайт и бесплатные книги	25
Благодарности.....	26
2. Начало	27
Числа со смыслом.....	27
Команды ARM	28
Процесс преобразования.....	29
А зачем вообще машинный код?.....	30
Языковые уровни.....	30
На орбиту!	31
RISC и наборы команд	32
Структура ассемблера	32
Ошибки на пути	33
Кросс-компиляторы.....	33
Чипы Raspberry Pi ARM.....	33
3. Проба пера	35
Командная строка3. Проба пера.....	35
Создание исходного файла	36
Написанное — исполнить!.....	40
Ошибки ассемблера.....	40
Компоненты	41
А если нет метки <code>_start</code> ?	44
Связывание файлов.....	44
Приблиаемся.....	46

Пара слов о комментариях.....	47
Редактор Geany Programmer's Editor.....	48
4. О битах в RISC-машинах.....	49
Преобразование двоичных чисел в десятичные.....	50
Преобразование десятичных чисел в двоичные.....	51
Преобразование двоичного числа в шестнадцатеричное.....	51
Преобразуем шестнадцатеричные числа в десятичные и обратно.....	53
Двоичное сложение.....	53
Вычитание.....	54
Дополнительный код.....	56
Когда двоичные числа не складываются.....	57
Стандартный калькулятор.....	58
5. Соглашения ARM.....	59
Длина слов.....	59
Доступ к памяти по байтам и словам.....	60
Регистры.....	61
Регистр <i>R15</i> : программный счетчик.....	63
Регистр состояния текущей программы.....	63
Биты и флаги.....	64
Установка флагов.....	64
Суффикс <i>S</i>	65
<i>R14</i> : регистр ссылок.....	66
<i>R13</i> : указатель стека.....	66
6. Обработка данных.....	67
Команды сложения.....	68
Вычитание.....	71
Умножение.....	72
Теперь о делении.....	74
Команды перемещения.....	75
Команды сравнения.....	76
Сортировка чисел.....	77
7. Входы и выходы.....	78
Команды <i>SWI</i> и <i>SVC</i>	78
Вывод на экран.....	79
Чтение с клавиатуры.....	81
Регистр <i>eax</i> и прочие.....	83
Программа Make.....	83
8. Логические операции.....	86
Логическое И (<i>AND</i>).....	86
Логическое ИЛИ (<i>OR</i>).....	87
Исключающее ИЛИ (<i>EOR</i>).....	87
Команды логических операций.....	88
Команда <i>ORR</i> для преобразования регистра символов.....	89
Очистка бита командой <i>BIC</i>	90
Проверка флагов.....	91
Регистры системных вызовов.....	94

9. Условное выполнение	95
Коды состояния с одним флагом.....	97
<i>EQ</i> : равно.....	97
<i>NE</i> : не равно.....	97
<i>VS</i> : переполнение.....	98
<i>VC</i> : нет переполнения.....	98
<i>MI</i> : знак «минус».....	98
<i>PL</i> : знак «плюс».....	98
<i>CS</i> : имеется перенос (<i>HS</i> : беззнаковое больше или равно).....	99
<i>CC</i> : нет переноса (<i>LO</i> : беззнаковое меньше).....	99
<i>AL</i> : безусловное исполнение.....	100
<i>NV</i> : безусловное неисполнение.....	100
Коды, проверяющие несколько флагов.....	100
<i>HI</i> : беззнаковое больше.....	100
<i>LS</i> : беззнаковое меньше или равно.....	101
<i>GE</i> : знаковое больше или равно.....	101
<i>LT</i> : знаковое меньше.....	101
<i>GT</i> : знаковое больше.....	101
<i>LE</i> : знаковое меньше или равно.....	102
Добавление суффикса <i>S</i>	102
10. Ветви и сравнения.....	103
Команды ветвления.....	103
Регистр ссылок.....	104
Использование команд сравнения.....	104
Применяем дальновидное мышление.....	105
Эффективное использование условных операторов.....	106
Обмен ветвей.....	107
11. Сдвиги и вращения.....	108
Логические сдвиги.....	108
Логический сдвиг вправо.....	110
Арифметический сдвиг вправо.....	110
Вращение.....	111
Расширенное вращение.....	112
Использование сдвигов и вращений.....	112
Прямой постоянный диапазон.....	113
Движение вверх.....	115
12. Умные числа	116
Длинное умножение.....	116
Умножение с накоплением.....	118
Деление и остаток.....	119
Умное умножение.....	120
Это только начало.....	121
13. Программный счетчик <i>R15</i>.....	122
Конвейерная обработка.....	123
Расчет ветвей.....	124

14. Отладка с использованием GDB	126
Когда все зависло.....	126
Сборка с GDB	127
Дизассемблер	130
Точки останова	132
Дамп памяти.....	136
Сокращения.....	137
Параметры сборки GDB.....	137
15. Передача данных	139
Директива <i>ADR</i>	139
Косвенная адресация	141
Команды <i>ADR</i> и <i>LDR</i>	143
Предварительно индексированная адресация	143
Доступ к байтам памяти.....	144
Обратная запись адреса.....	146
Постиндексированная адресация	146
Байтовые условия	148
Относительная адресация через регистр <i>PC</i>	148
16. Передача блока	150
Обратная запись.....	152
Процедура копирования блока	153
17. Стеки	155
Тянитолкай ;-)	155
Рост стека	157
Применение стеков.....	159
Работа в фрейме	160
Указатель фрейма	160
18. Директивы и макросы	162
Директивы хранения данных	162
Выравнивание данных.....	164
Макросы	165
Включение макросов	168
19. Работа с файлами	171
Права доступа к файлам	176
20. Использование библиотеки <i>libc</i>	179
Использование функций языка C в ассемблере	179
Структура файла исходного кода	180
Исследование исполняемого файла	182
Ввод чисел с помощью функции <i>scanf</i>	184
Вывод информации	186
21. Пишем функции	188
Стандарты функций.....	188
Использование регистров	190

Больше трех.....	190
Сохранение ссылок и флагов.....	192
Надежные процедуры вывода.....	193
Пузырьковая сортировка.....	195
22. Дизассемблирование программ на C.....	198
GCC — он как швейцарский нож.....	198
Простой фреймворк C.....	199
Создание файла ассемблера.....	200
Строительные блоки.....	202
Пример функции <i>printf</i>	204
Переменные указателя фрейма.....	205
Дизассемблирование системных вызовов.....	206
23. Функции GPIO.....	207
Отображение памяти.....	207
Контроллер GPIO.....	209
Вводы и выходы GPIO.....	211
Сборка кода.....	217
Другие функции GPIO.....	222
Описание контактов GPIO.....	222
24. Числа с плавающей точкой.....	225
Архитектура VFP.....	225
Регистровый файл.....	227
Управление и вывод на экран.....	229
Сборка и отладка на VFP с помощью GDB.....	231
Загрузка, хранение и перемещение.....	233
Преобразование точности.....	235
Векторная арифметика.....	236
25. Регистр управления VFP.....	237
Условное исполнение.....	238
Скалярные и векторные операции.....	240
Какой тип оператора?.....	242
Параметры <i>LEN</i> и <i>STRIDE</i>	243
26. Сопроцессор Neon.....	247
Ассемблер Neon.....	249
Команды и типы данных Neon.....	251
Режимы адресации.....	253
Параметр <i>Stride</i> команд <i>VLD</i> и <i>VST</i>	253
Загрузка в прочих форматах.....	256
Neon Intrinsic.....	256
Массивы Neon.....	257
Правильный порядок.....	261
Матричная математика.....	262
Матричное умножение.....	265
Пример использования макроса.....	270

27. Код Thumb	272
Различия	272
Пишем на Thumb	274
Доступ к старшим регистрам.....	278
Операторы стека	279
Одно- и многорегистровые команды	279
Функции в Thumb	279
Команды ARMv7 Thumb.....	280
28. Единый язык	281
Изменения Thumb	282
Новые команды A32	283
Сравнение по нулю.....	284
Сборка UAL.....	284
29. Обработка исключений.....	287
Режимы работы.....	287
Векторы	288
Настройка регистров	290
Обработка исключений	292
Команды <i>MRS</i> и <i>MSR</i>	293
Что происходит при возникновении прерывания?	294
Решения о прерываниях	295
Возврат из прерываний	295
Пишем процедуры прерывания	296
30. System-on-Chip	297
Микросхема и набор команд ARM	298
Сопроцессоры	299
Конвейер.....	299
Память и кэши.....	300
GPU	301
Обзор ARMv8.....	301
64-разрядная ОС Raspberry Pi.....	302
А что в итоге?.....	302
Принцип Архимеда.....	302
Приложение 1. Коды символов ASCII	304
Приложение 2. Набор команд ARM.....	306
Команды сравнения и проверки	307
Команды ветвления	307
Арифметические команды	308
Логические команды	308
Команды перемещения данных	309
Приложение 3. Системные вызовы ROS.....	310
Приложение 4. Описание электронного архива.....	316
Предметный указатель	317

СПИСОК ПРОГРАММ

Программа 3.1. Простой исходный файл	39
Программа 3.2. Часть 1 исходного файла	44
Программа 3.3. Часть 2 исходного файла	45
Программа 6.1. Простое 32-битное сложение	69
Программа 6.2. 64-битное сложение	70
Программа 6.3. 32-битное умножение	73
Программа 6.4. Использование умножения с накоплением MLA	73
Программа 6.5. Деление командой SDIV	74
Программа 7.1. Системный вызов 4 для вывода строки на экран	80
Программа 7.2. Системный вызов 3 и чтение с клавиатуры	81
Программа 7.3. Автоматизация сборки и привязки с помощью Make	84
Программа 8.1. Преобразование регистра символов	89
Программа 8.2. Вывод двоичного числа	91
Программа 10.1. Условное выполнение позволяет сократить программу	107
Программа 12.1. Долгое и нудное умножение	117
Программа 12.2. Долгое нудное деление	119
Программа 14.1. Гибкий make-файл для отладки кода	138
Программа 15.1. Использование директивы <code>ADR</code>	140
Программа 15.2. Использование предварительно индексированной косвенной адресации	145
Программа 15.3. Использование постиндексированной адресации	147
Программа 16.1. Перемещение блоков памяти	153
Программа 18.1. Использование директив <code>.byte</code> и <code>.equ</code>	163
Программа 18.2. Реализация простого макроса	166
Программа 18.3. Многократный вызов макроса	167
Программа 18.4. Файл макроса <code>AddMult</code>	168
Программа 18.5. Проверка подключения макроса	169
Программа 19.1. Создание файлов и доступ к ним	172
Программа 20.1. Структура исходного файла GCC	181
Программа 20.2. Передача параметров в <code>printf</code>	183
Программа 20.3. Чтение и преобразование числа с помощью функции <code>scanf</code>	185
Программа 20.4. Объединение <code>scanf</code> и <code>printf</code>	187
Программа 21.1. Передача значений функции через стек	191
Программа 21.2. Вывод вектора слов	193
Программа 21.3. Проверка <code>printw</code>	194
Программа 21.4. Процедура пузырьковой сортировки	195
Программа 21.5. Тест пузырьковой сортировки	196
Программа 22.1. Простая программа на языке C для вывода символа	200
Программа 22.2. Окончательный вариант кода	203
Программа 22.3. Код программы <code>hello world</code> на C	204
Программа 22.4. Код для работы с функцией <code>scanf</code>	205
Программа 23.1. Файловый доступ к памяти GPIO	213
Программа 24.1. Вывод значения с плавающей точкой с помощью функции <code>printf</code>	229
Программа 24.2. Вывод двух или более значений с плавающей точкой	230
Программа 25.1. Условное выполнение в VFP	240
Программа 25.2. Использование битов <code>LEN</code> и <code>STRIDE</code> для сложения векторов	245
Программа 26.1. Простая проверка Neon	249
Программа 26.2. Поворот 2D-матрицы на 90 градусов (по часовой стрелке)	258
Программа 26.3. Сложение двух матриц размером 4×4	263
Программа 26.4. Матричное умножение чисел с одинарной точностью	266
Программа 26.5. Умножение матриц, но с макросами	270
Программа 27.1. Вызов режима <code>Thumb</code> и запуск кода <code>Thumb</code>	275
Программа 27.2. Использование внешних функций путем взаимодействия кода	277
Программа 28.1. Единый язык ассемблера	285

*Посвящается всем медицинским работникам:
медсестрам, врачам и другому персоналу,
осуществляющим уход за пациентами во всем мире.*

*Тем, кто выложился по полной в эти годы пандемии.
Память о вас останется в наших сердцах навсегда.*

Спасибо!

Об авторе



Брюс Смит приобрел свой первый компьютер Acorn Atom в 1980 году. Увлечение компьютерами мгновенно затянуло его, и он стал постоянным автором крупных тематических изданий, включая Computing Today и Personal Computer World. С появлением компьютера BBC Micro его работа перешла от журналов к книгам, и его книга 1982 года «Interfacing Projects for BBC Micro» стала классикой своего времени. В ней он рассказал простым пользователям домашних компьютеров, как подключиться к внешнему миру. Он был одним из первых, кто писал о чипе ARM, когда они были пред-

ставлены на Acorn Archimedes в 1987 году.

Брюс рассказывал обо всех аспектах использования компьютера. Его дружелюбный и понятный стиль письма вдохновил одного рецензента на вот такой отзыв: «Это первая книга о компьютерах, которую я читал для удовольствия, лежа в постели, а не ради того, чтобы поскорее заснуть!» Книги Брюса переведены на многие языки и продаются по всему миру. Его издавали в BBC Books, Collins, Virgin Books и Rough Guides.

Брюс родился в лондонском Ист-Энде, а сейчас живет в Сиднее, Новый Южный Уэльс.

Его сайт в Интернете: www.brucesmith.info.

Из отзывов с пятью звездами на AMAZON НА ПРЕДЫДУЩИЕ ИЗДАНИЯ ЭТОЙ КНИГИ:

«Отличная книга. Настоятельно рекомендую ее всем, кто хочет работать с ARM-асемблером».

«Эта книга подойдет не только начинающим, но будет также полезна программистам среднего уровня, и в ней затронуты некоторые более сложные темы. На мой взгляд, эта книга заслуживает оценки 10/10 и обязательна к прочтению всем, кто планирует начать или возвращается к программированию на асемблере на процессоре ARM v6 (Raspberry Pi). Книга сэкономила мне много времени, поскольку вся необходимая информация теперь собрана в одном месте. Спасибо Брюсу за такое сокровище!»

«Это отличная книга для начинающих программистов, которые хотят глубже изучить Raspberry Pi и по-настоящему понять, как работает компьютер. Приведены отличные и 00подробные примеры».

«Смит не только излагает все нужные материалы и инструкции на языке асемблера для процессора ARM Raspberry Pi, но также дает четкие практические указания о том, как заставить все это работать... Не важно, хотите ли вы просто побаловаться с Raspberry Pi, или рассматриваете Pi как ступеньку к чему-то большему — книга Брюса Смита определенно должна стоять на вашей полке».

1. Введение

Я ничуть не удивился, когда в начале 2013 года появилась новость о том, что продажи контроллеров Raspberry Pi (рис. 1.1) достигли 1 млн штук. А само это число заставило меня вернуться на несколько лет (даже десятилетий) назад и вспомнить аналогичную новость о платах BBC Micro. Обе платформы производятся на предприятии, расположенном в Кембридже в Великобритании, и общего у них гораздо больше, чем вы можете себе представить.

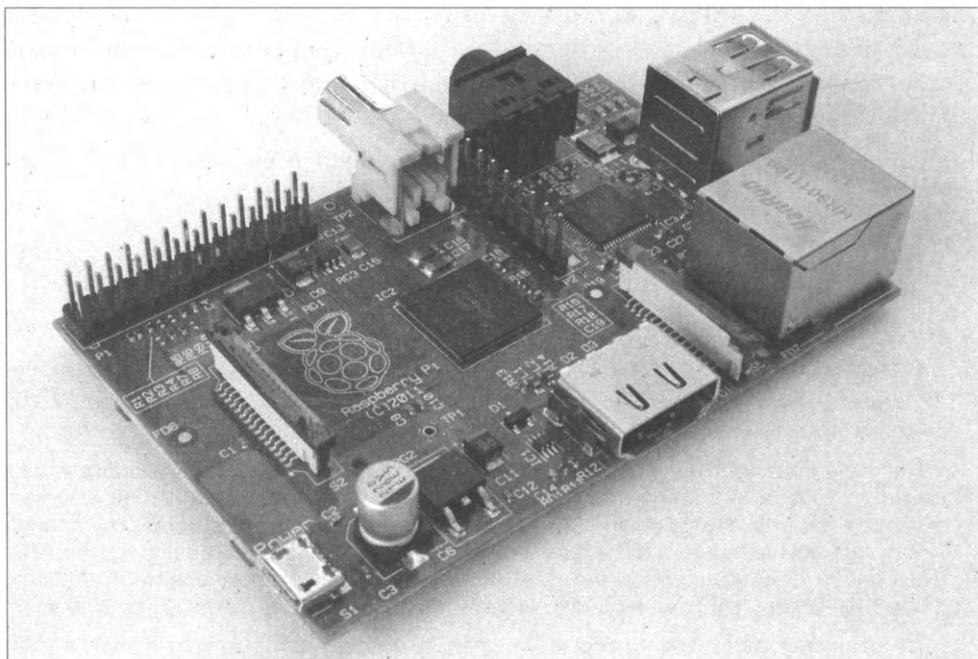


Рис. 1.1. Плата Raspberry Pi Model B

Обе системы покорили мое сердце и сердца миллионов других людей. На первую я в свое время спускал все свои сбережения, а вот Raspberry Pi совершенно не ударила по моему кошельку. Но огромная разница в цене была почти единственным различием, а прочие черты оказались весьма схожи. Одна из самых важных черт —

возможность запускать на них широкий спектр программного обеспечения и образовательных инструментов, а также легкость в подключении внешних устройств и управлении внешним миром. Я подозреваю, что у большинства владельцев Raspberry Pi лежит не один, а как минимум парочка контроллеров.

Впервые после BBC Micro на рынке появилась дешевая и невероятно богатая возможностями система, которую может использовать практически каждый. В то же время на рынке доминировали платформы PC и Mac и появились другие устройства, более ориентированные на игры. Но ничего из этого не сказалось на тех, чьим домашним хобби была работа с техникой. У профессионалов имелось множество платформ для разработки, с которыми можно было экспериментировать, но у домашних любителей компьютеров оставались проблемы с доступностью таких платформ. А вот Raspberry Pi со своей ценой, соизмеримой с парочкой DVD, изменила все.

Запуск Raspberry Pi Zero в ноябре 2015 года вывел экономические преимущества платформы на новый уровень. Полноценный компьютер на базе ARM можно получить всего за 5 долларов. Это ж с ума сойти! И, чтобы окончательно добить фанатов, к каждому экземпляру MagPi (официального журнала Raspberry Pi) за декабрь 2015 года прилагался контроллер в подарок. Нельзя ли теперь считать Raspberry Pi первым в мире полноценным компактным компьютером?

На момент подготовки этой книги (декабрь 2020 г.) продажи плат Raspberry превысили 30 млн штук. Эта отметка, как сообщается, была достигнута к началу 2020 года, когда многие компании начали использовать Raspberry Pi в своей инфраструктуре и разработках. Компания Oracle назвала кластер из 1060 плат Raspberry Pi самым большим в мире суперкомпьютером Pi с 4240 ядрами. Jet Propulsion Laboratory (JPL) используют эти платы в своей марсианской миссии.

Безграничные возможности

Одна из причин появления Raspberry Pi заключалась в том, что у людей имелась необходимость в дешевой системе программирования, пределы возможностей которой ограничивались бы только воображением пользователя. Я надеюсь, что эта книга поможет многим из читателей осуществить свою мечту. А это вполне реально. Весьма вероятно, что чтение этих страниц станет вашим первым шагом к полезному и познавательному времяпрепровождению. И, кто знает, может быть, вы станете частью нового поколения компьютерных программистов, творящих на пределе возможного.

Цель этой книги — дать читателю лучшее понимание того, как устроена работа компьютера на уровне ниже высокоуровневых языков программирования, таких как C или Python. Глубже разобравшись с работой компьютера, вы сможете гораздо более продуктивно писать программное обеспечение на языках более высокого уровня. Именно обучение программированию на ассемблере поможет вам достичь этой цели.

Эта книга представляет собой начальное руководство по написанию кода на ассемблере для Raspberry Pi и, в частности, с использованием ОС Raspberry Pi

(операционной системы, ранее называвшейся Raspbian). Язык ассемблера генерирует машинный код, который можно запускать прямо на вашем компьютере.

Сам я сначала учился программировать на ассемблере на машинах, созданных еще на ранних этапах разработки Acorn, а позднее стал свидетелем развития чипов ARM в системах Archimedes и BBC Micro Second-Processor. В конечном итоге именно тот ассемблер стал предшественником того, что сегодня используется в Raspberry Pi. Этим я хочу лишь подчеркнуть, что все, чему вы здесь научитесь, будет полезно и актуально в течение долгих лет. Однозначно, стоящая инвестиция.

Начинаем экспериментировать

Эта книга адресована не совсем уж полным новичкам, однако вам не потребуется иметь опыт работы с языком ассемблера или машинным кодом, чтобы понять ее и начать экспериментировать. Тем не менее вам пригодится любой опыт в программировании, и любой известный вам структурированный язык поможет понять базовые концепции ассемблера.

Поскольку это в первую очередь практическое руководство, мы с вами напишем множество программ. Чтобы научиться программировать, нужно экспериментировать, совершать ошибки, а затем учиться на них. Вне всяческих сомнений, лучший способ понять происходящее — экспериментировать со значениями и данными, и такое экспериментирование следует поощрять. Все изложенные в книге программы можно будет скачать с сайта книги. Книга одинаково применима ко всем версиям Raspberry Pi. Но подробнее об этом чуть позже.

Компилятор GNU C

Существует несколько операционных систем (сред программирования), которые вы можете скачать для работы с Raspberry Pi. Как следует из ее названия, в этой книге мы будем работать с ОС Raspberry Pi (ранее называвшейся Raspbian). В ней есть все, что вам потребуется для написания и запуска ваших программ.

Также для работы нам потребуется GCC — коллекция компиляторов GNU. Первоначальным автором компилятора GNU C (GCC) был Ричард Столлман, основатель проекта GNU. Проект GNU был запущен в 1984 году с целью создания полноценной свободно доступной операционной системы, которая дала бы свободу сотрудничеству между пользователями компьютеров и программистами. Любой операционной системе нужен компилятор C, и, поскольку в те времена свободно доступных компиляторов не существовало, проекту GNU пришлось писать свои с нуля.

Работа финансировалась за счет пожертвований физических лиц и компаний в Фонд свободного программного обеспечения — некоммерческую организацию, созданную для поддержки проекта GNU. Первая версия GCC вышла в 1987 году. С тех пор GCC стал одним из важнейших инструментов в разработке бесплатного программного обеспечения и работает практически на всех существующих ОС.

GCC — бесплатное программное обеспечение, распространяемое под Стандартной публичной лицензией GNU (GNU General Public License). Это означает, что вы

можете использовать и изменять GCC, как и все программное обеспечение GNU. Если вам нужна поддержка нового ЦП, нового языка или новой функции, вы можете добавить ее самостоятельно или попросить кого-нибудь сделать это за вас.

Вы, наверное, знаете, что C — чрезвычайно популярный язык программирования, а еще он очень тесно связан с микропроцессором Advanced RISC Machine (ARM), который используется в ядре Raspberry Pi. Но вам не обязательно знать C, чтобы писать программы на ассемблере, поэтому этим не озабочивайтесь.

GCC — очень умная программа, и использовать ее можно по-разному. Одним из ее ключевых компонентов является ассемблер, который нас, собственно говоря, и интересует в первую очередь.

Учимся на примерах

Программы, приведенные в этой книге, призваны проиллюстрировать вам концепции, которые я буду объяснять простым языком и, где это возможно, с практической точки зрения. Я не буду грузить вас длинными и сложными листингами программ, да и нет в этом необходимости. Я буду давать простые примеры и информацию и объединять их в группы по два, три, четыре и более, чтобы программа в итоге получала полезный результат, а вы научились бы многому в процессе ее создания.

Иногда возникают вопросы вроде «курицы и яйца», но я старался свести их к минимуму. Мы будем вводить новые понятия в порядке, соответствующем полученным знаниям. Однако это не всегда возможно, и на этих моментах я буду останавливаться отдельно. В таких случаях вам придется просто поверить, что это работает, и однажды станет ясно, почему.

Программирование — это и впрямь весело. Я написал множество книг на эту тему, и значительная часть из них была посвящена домашним компьютерам и тому, как их программировать и использовать. Меня никто не учил работе с компьютерами. А если я могу сделать что-то, значит, сможет и любой другой человек. Но в этом есть и доля огорчения! Нет на свете ни одного программиста, будь то новичок или эксперт, который не тратил бы кучу времени на решение некоторой задачи программирования с тем, чтобы позже понять, что решение было прямо у него под носом. Настоящее удовлетворение приходит тогда, когда вы сами решаете задачу.

Дам совет: если вы не можете решить проблему, отвлекитесь и займитесь чем-нибудь еще. Очень часто решение приходит к вам именно тогда, когда вы занимаетесь чем-то другим. Это работает не только в программировании, но и в жизни!

Что вы узнаете?

Если говорить коротко, вы станете опытным программистом на ассемблере. К концу этой книги, если вы проработаете и примените на практике приведенные в ней примеры программ, вы сможете разрабатывать, писать и создавать программы с машинным кодом для выполнения огромного множества задач. Вы также получи-

те основу, позволяющую углубиться в другие темы, касающиеся микросхем ARM и системного программирования.

Вы также познакомитесь с различными способами использования компилятора GCC, включая написание кода для операционной системы Raspberry Pi и объединение собранных программ с библиотеками автономных функций.

Вы узнаете, как интерпретировать и управлять возможностями платы Raspberry Pi на фундаментальном уровне. Вы будете писать инструкции непосредственно для чипа ARM.

Вам также необходимо научиться решать проблемы. Когда программа с машинным кодом начинает работать неправильно, причиной этого часто является простая логическая ошибка. У GCC есть встроенные инструменты отладки, и мы увидим, как их эффективно использовать. Я также дам несколько полезных советов о том, как сузить область поиска и в конечном итоге найти источник проблемы.

Совместимость четвертого издания книги

В принципе, эта книга совместима со всей линейкой компьютеров Raspberry Pi. Один из основных принципов разработчиков аппаратного и программного обеспечения этой системы — обеспечить ее обратную совместимость (насколько это возможно). По сути, это означает, что программа, которая работает на одной модели Raspberry Pi, будет работать и на другой, если она написана правильно (с точки зрения программного обеспечения). Стороннее оборудование тоже должно работать, за исключением имеющего некоторые аппаратные изменения. Например, порт HDMI работать будет, но подключаться к нему придется через другой кабель (к примеру, не стандартный, а с микроразъемом).

Физически разница между моделями и версиями достаточно очевидна (рис. 1.2). Для целей этой книги я буду исходить из предположения, что у вас есть плата

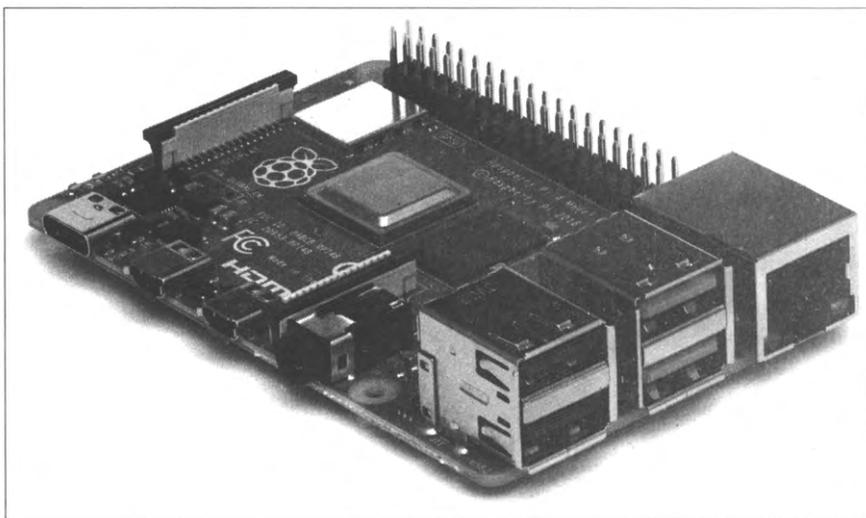


Рис. 1.2. Raspberry Pi 4 Model B

с портом GPIO в стандартной комплектации, а именно здесь с точки зрения этой книги проявляется значительная часть изменений. Вам станет понятно, что я имею в виду, когда вы станете рассматривать некоторые из примеров, в которых показано, как использовать код для подключения к устройствам. О необходимых изменениях я также буду предупреждать.

К сегодняшнему моменту плата Raspberry Pi сменила несколько различных версий чипа ARM, и это может продолжаться и в будущем. В настоящее время просто имейте в виду, что различия могут быть, но они станут видны вам по мере роста ваших знаний. Поскольку книга эта, по сути, адресована новичкам, имеющиеся различия не повлияют на основные понятия программирования. Эта книга — введение в язык ассемблера ARM, независимо от модели Raspberry Pi, которую вы используете.

ОС Raspberry Pi

Когда появилась новость о выпуске Raspberry Pi 4 с объемом памяти 8 Гбайт (май 2020 г.), компания Raspberry Pi Foundation сообщила, что меняет название своей официальной операционной системы с Raspbian на Raspberry Pi OS. Целью такого изменения явилось создание аналогичной и стабильной операционной системы и среды в стиле Windows для всех платформ, а также по максимуму, насколько возможно, обеспечение постоянной обратной совместимости с программным обеспечением и приложениями.

Имеются две основные категории процессоров: 32-разрядные и 64-разрядные. Платформа поддерживает как 32-разрядные, так и 64-разрядные операционные системы, и внешний вид ОС не меняется, т. к. сборка этих версий происходит отдельно. Попросту говоря, 64-разрядный процессор более эффективен, чем 32-разрядный, т. к. он может обрабатывать больше данных одновременно. 64-битный процессор способен хранить больше значений, включая адреса памяти, и имеет в распоряжении в 4 млрд раз больше адресов, чем физическая память 32-разрядного процессора. Это очень круто!

С точки зрения пользователя, использующего стандартную 32-разрядную операционную систему, функциональной разницы в том, что описано в этой книге, не будет. Читайте Raspberry Pi OS вместо Raspbian и наоборот. Что еще более важно, ОС остается обратно совместимой, так что ОС Raspberry Pi будет работать на всех версиях Raspberry Pi. Вы можете обновить текущую операционную систему до последней версии Raspberry Pi OS в любой момент.

Проект Raspbian не сворачивается и, несомненно, ляжет в основу при создании 32-разрядной версии ОС Raspberry Pi. Учитывая, что несколько моделей Raspberry Pi, включая очень популярную Raspberry Pi Zero, никогда не будут работать с 64-разрядной ОС, 32-разрядная платформа еще несколько лет останется востребованной, если не доминирующей.

А что насчет 64-разрядной системы?

Raspberry Pi 2B (v1.2) была первой платой с 64-разрядным процессором ARM, но для него не было выпущено официальной 64-разрядной ОС. Дело в том, что Raspberry Pi Foundation сосредоточилась в основном на том, чтобы 32-разрядная ОС Raspbian работала на всех поколениях RPi. Начиная с упомянутой Raspberry Pi 2B (v1.2), все версии RPi работают на 64-разрядных процессорах, которые также могут работать в 32-разрядном режиме.

32-разрядный режим микросхемы ARM называется AArch32 (или A32), а 64-разрядный — AArch64 (или A64). Это означает, что 32-разрядное программное обеспечение может работать на 64-разрядном чипе ARM, а вот наоборот — нет.

На момент подготовки книги уже вышла бета-версия 64-разрядной ОС Raspberry Pi, и вы можете скачать и попробовать ее. После отработки бета-версии она будет выпущена для всех 64-разрядных плат (см. главу 30). Обратите внимание, что 32-разрядная версия (A32) языка ассемблера ARM отличается от 64-разрядной версии (A64), поэтому код, написанный на A32, не будет просто так работать на A64 без определенных изменений.

Клавиатурные вычисления

В ноябре 2020 года произошло неизбежное: наконец-то появились настольные компьютеры на Raspberry Pi, в которых плата была модернизирована и упакована в красно-белую клавиатуру для запуска Raspberry Pi 400 (рис. 1.3).



Рис. 1.3. Raspberry Pi 400 — идеальный компьютер для клавиатурных вычислений

Значимость ARM

Все микропроцессоры основаны на определенной архитектуре набора команд (ISA), и наиболее значимой из них в последние годы была архитектура x86, которая доминировала на рынке настольных компьютеров и ноутбуков (ПК и Apple). В последнее время также используется 64-разрядная версия архитектуры под названием x86-64, или AMD64.

Однако большинство планшетов и смартфонов Apple и Android работают на процессорах ARM, несовместимых с архитектурой x86. Чип ARM на этих устройствах

используется из-за его низкого энергопотребления и, как следствие, увеличения времени автономной работы. Но эта несовместимость означает, что программное обеспечение, скомпилированное для настольных компьютеров и ноутбуков, нельзя напрямую запустить на мобильных устройствах под управлением ARM.

Вычислительные процессоры с сокращенным набором команд (или процессоры RISC), применяемые в устройствах ARM, используют для выполнения задачи множество простых инструкций. Процессоры x86 работают на CISC (сложном наборе команд) и для выполнения аналогичной задачи задействуют большее число сложных инструкций.

За счет этого в чипах ARM и достигается более низкое энергопотребление, поскольку в их структуре ядра меньше транзисторов, чем у чипов на основе CISC. Это также означает, что, с точки зрения программиста, вам, как программисту ARM, придется учить меньше инструкций!

Другое очень важное отличие состоит в том, что компания ARM Holdings не является производителем микросхем. Она разрабатывает микросхемы, а затем продает лицензии на их производство, чтобы другие производители могли включить их в свои собственные разработки.

В середине 2020 года компания Apple Computers объявила, что полностью перейдет на собственные чипы ARM, которые будут устанавливаться во все новые компьютеры, что обеспечит работоспособность ее приложений на всех выпускаемых ею устройствах. Microsoft Office 365, Adobe Creative Cloud и прочие программы, вероятно, тоже станут запускаться на новых компьютерах Mac ARM. Не требуется большого воображения, чтобы представить, как это будет работать на любой машине на базе ARM.

Ноутбуки и настольные компьютеры на базе x86 еще какое-то время продолжат доминировать на рынке, по крайней мере с точки зрения количества выпускаемых компьютеров, но машины Apple Mac на базе ARM должны вывести на основной рынок ноутбуки/настольные компьютеры на базе ARM и могут соблазнить этим других игроков.

А это — еще один повод изучать программирование на ассемблере ARM.

Raspberry Pi сквозь века

Платформа Raspberry Pi развивается с момента первого выпуска в 2012 году. В каждом новом поколении часто внедряются значительные улучшения типа центрального процессора, емкости памяти, поддержки сети и периферийных устройств.

Некоторые из основных спецификаций для каждой из моделей Raspberry Pi приведены в табл. 1.1. Как правило, Raspberry Pi бывают трех разных моделей:

- ◆ Model B — это «полноразмерные» платы с портами Ethernet и USB. В версиях B+ часто добавляются улучшения или особые различия;
- ◆ Model A — это платы квадратной формы. Их можно считать «более легкой» версией Raspberry Pi, обычно с более низкими характеристиками, чем у флаг-

манской модели B, с меньшим количеством портов USB и часто без Ethernet. Поэтому они дешевле;

- ◆ Zero — самый маленький доступный вариант Raspberry Pi. У этих плат меньше вычислительной мощности, чем у модели B, но они также потребляют меньше энергии. Ни USB, ни Ethernet – все простенько и со вкусом!

Объем памяти, доступный на каждой Raspberry Pi, для этой книги значения не имеет, но имейте в виду, что адреса в памяти, которые будут отображаться в вашей программе, могут отличаться от приведенных в книге примеров. Однако это не должно мешать работе программы.

Таблица 1.1. Модели Raspberry Pi и их основные характеристики

Модель RPI	Дата выпуска	SOC	ЦП	FPU	GPU	ARM IS
400	Ноябрь 2020 г.	BCM 2711C0	4 × Cortex-A72 с частотой 1,8 ГГц	VFPv4 + NEON	Broadcom VideoCore VI	ARMv8-A (64/32-бит)
4 B	Июнь 2019 г.; май 2020 г. (8 Гбайт)	BCM2711B0	4 × Cortex-A72 с частотой 1,5 ГГц	VFPv4 + NEON	Broadcom VideoCore VI	ARMv8-A (64/32-бит)
3 B +	Май 2018 г.	BCM2837B0	4 × Cortex-A53 с частотой 1,4 ГГц	VFPv4 + NEON	Broadcom VideoCore IV	ARMv8-A (64/32-бит)
3 B	Февраль 2016 г.	BC2837	4 × Cortex-A53 с частотой 1,2 ГГц	VFPv4 + NEON	Broadcom VideoCore IV	ARMv8-A (64/32-бит)
3 A +	Ноябрь 2018 г.	BCM2837B0	4 × Cortex-A53 с частотой 1,4 ГГц	VFPv4 + NEON	Broadcom VideoCore IV	ARMv8-A (64/32-бит)
2 B v1.2	Октябрь 2016 г.	BCM2837	4 × Cortex-A53 с частотой 900 МГц	VFPv4 + NEON	Broadcom VideoCore IV	ARMv8-A (64/32-бит)
2 B	Февраль 2015 г.	BCM2836	4 × Cortex-A7 с частотой 900 МГц	VFPv3 + NEON	Broadcom VideoCore IV	ARMv7-A (32-бит)
1 B +	Июль 2014 г.	BCM2835	1 × ARM 1176JZF-S с частотой 700 МГц	VFPv2	Broadcom VideoCore IV	ARMv6Z (32-бит)
1B	Май 2012 г.	BCM2835	1 × ARM 1176JZF-S с частотой 700 МГц	VFPv2	Broadcom VideoCore IV	ARMv6Z (32-бит)
1 A +	Ноябрь 2014 г.	BCM2835	1 × ARM 1176JZF-S с частотой 700 МГц	VFPv2	Broadcom VideoCore IV	ARMv6Z (32-бит)
1 A	Февраль 2013 г.	BCM2835	1 × ARM1176JZF-S с частотой 700 МГц	VFPv2	Broadcom VideoCore IV	ARMv6Z (32-бит)
Zero WWW	Февраль 2017 г.	BCM2835	1 × ARM 1176JZF-S с частотой 1 ГГц	VFPv2	Broadcom VideoCore IV	ARMv6Z (32-бит)
Zero v1.3	Май 2016 г.	BCM2835	1 × ARM1176JZF-S с частотой 1 ГГц	VFPv2	Broadcom VideoCore IV	ARMv6Z (32-бит)
Zero v.1.2	Ноябрь 2015 г.	BCM2835	1 × ARM1176JZF-S с частотой 1 ГГц	VFPv2	Broadcom VideoCore IV	ARMv6Z (32-бит)

В табл. 1.1 используется много сокращений, которые могут быть вам знакомы, а могут быть и новыми. Мы кратко рассмотрим каждое из них и при необходимости повторно вернемся к ним позже в книге.

- ◆ **SOC, System-on-Chip (Система на кристалле)** — системный чип. Один из ключевых элементов дизайна Raspberry Pi и, как правило, самый большой чип на плате, который содержит все важные компоненты в одном корпусе. В него входят процессор ARM и графические процессоры. Обратите внимание, что каждый чип SoC сопровождается числом — например, на плате Raspberry Pi 4 B установлен чип BCM2771.
- ◆ **ЦП** — это процессор ARM или используемые процессоры, интегрированные в SoC. В таблице указан конкретный чип ARM, а также количество его ядер и скорость работы. К примеру, на оригинальном Raspberry Pi B установлен процессор ARM 1176JZF-S, работающий на частоте 700 МГц. В новой Raspberry Pi 4 B используются четыре ядра ARM Cortex-A72, работающие на частоте 1,5 ГГц.
- ◆ **FPU** — модуль с плавающей точкой, или математический сопроцессор, который реализует сложные математические функции. Он тоже входит в SoC, а в более поздних версиях SoC установлен сопроцессор Neon, позволяющий FPU выполнять несколько операций параллельно. **VFP** — это сокращение от **Vector floating-point**, вектор с плавающей точкой.
- ◆ **GPU** — графический процессор. Он входит в SoC Broadcom, производится компанией Broadcom и обычно представляет собой VideoCore IV, а в Raspberry Pi 4 B — версию VideoCore VI. По сути, это мультимедийный процессор, который может декодировать различные графические и звуковые форматы (кодеки) для обеспечения отличного качества изображения на мониторе при сохранении низкого энергопотребления.
- ◆ В последнем столбце таблицы указана реализованная архитектура набора команд и отмечено, является она 32-разрядной или 64-разрядной. Как мы уже говорили ранее, 64-битный набор инструкций доступен на Raspberry Pi, начиная с RPi 2 B v1.2, т. е. фактически с октября 2016 года.

Вычислительные модули

Вычислительный модуль (Computer Module) — это упрощенная версия Raspberry Pi, предназначенная для промышленных или коммерческих приложений. Суть в том, что их может использовать в своих проектах кто угодно (вы приобретаете плату и дорабатываете ее самостоятельно под свою задачу). Таким образом, вычислительные модули не имеют возможностей подключения, которые вы можете найти на эквивалентной плате Raspberry Pi, но при этом они не столь надежны. Поэтому, хоть они и работают на процессорах ARM и на Raspberry Pi, они здесь не рассматриваются.

Но затем появился модуль Computer Module 4. В отличие от предыдущих вычислительных модулей, у него имеется обновленная плата ввода/вывода, простой доступ

ко всем стандартным разъемам интерфейсов, и, стало быть, это уже готовая платформа разработки и отправная точка для ваших собственных проектов.

Используемые обозначения

В этой книге применяются стандартные обозначения. Числовые типы и определенные операции с числами часто используются в книгах по программированию, подобных этой, и важно различать их. В табл. 1.2 приведен их краткий список. Точное значение обозначений будет ясно, когда мы встретим их в тексте книги.

Таблица 1.2. Стандартные обозначения

Обозначение	Пояснение
% или 0b	Обозначает, что идущее за этим обозначением число записано в двоичной системе счисления (с основанием 2). Например: %11110000 или 0b11110000
0x	Обозначает, что следующее за ним число является шестнадцатеричным. Например: 0xCAFE
< >	Угловые скобки часто используются для обозначения слова, которое не следует понимать буквально, а надо расценивать как объект, используемый с командой. Например, запись <Регистр> означает, что в угловых скобках следует использовать имя регистра — например: R0, а не само слово «Регистр»
Dest	Сокращение от destination
операнд1	Комментарии в тексте часто будут ссылаться на операнд1 и его использование. Вы же мысленно подставляете вместо него те или иные значения. Обычно операнд1 — это регистр
операнд2	Комментарии в тексте часто будут ссылаться на операнд2 и его использование. Вы же мысленно подставляете вместо него те или иные значения. Обычно операнд1 — это регистр
оп1	Сокращение от операнд1
оп2	Сокращение от операнд2
{ }	Скобки показывают, что указанный в них элемент является необязательным и может быть опущен, если он не нужен. Например, запись ADD(S) означает, что значения S может и не быть

Центр истории вычислительной техники

Центр истории вычислительной техники (ССН, Center of Computing History) — новаторская образовательная благотворительная организация, открывшаяся на своем нынешнем месте в Кембридже в августе 2013 года. ССН (www.computinghistory.org.uk) раскрывает историю информационной эпохи на основе изучения исторического, социального и культурного воздействия разработок в области персональных компьютеров на человеческую цивилизацию. В этом центре имеется огромная кол-

лекция иллюстрирующих эту историю экспонатов, которые используются в образовательных программах и программах мероприятий.

Цель ССН — вдохновить и дать возможность обучения всем: от дошкольников до людей старше 70 лет, чтобы они стали уверенными и творческими пользователями информации и цифровых технологий. Центр предоставляет образовательные услуги, включая семинары по программированию и электронике, а также другое интерактивное обучение с использованием BBC Micros и Raspberry Pi 1980-х годов.

Веб-сайт и бесплатные книги

Зайдите на сайт www.brucesmith.info (рис. 1.4), щелкните на интересующей вас книге и следуйте появляющимся там указаниям. С этого сайта вы можете скачать все приведенные в книге программы, получить доступ к обновлениям книги, а также к дополнительной информации и функциям. Кроме того, здесь же вы найдете ссылки на другие полезные сайты и полезные материалы, а также подробную информацию о предстоящих публикациях Bruce Smith Books, посвященных Raspberry Pi.

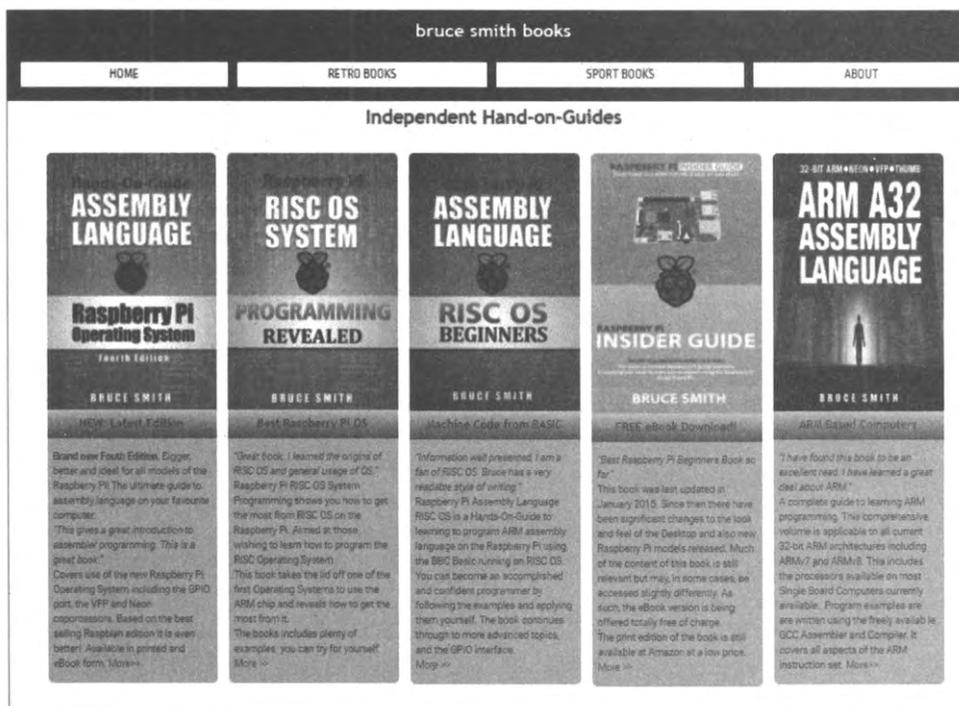


Рис. 1.4. Сайт автора. Нажмите на обложку книги для получения дополнительной информации

ПРИМЕЧАНИЕ К РУССКОМУ ИЗДАНИЮ

Электронный архив с файлами приведенных в книге программ можно загрузить с FTP-сервера издательства «БХВ» по ссылке: <ftp://ftp.bhv.ru/9785977568012.zip> или со страницы книги на сайте <https://bhv.ru/> (см. приложение 4). Обратите при этом внима-

ние, что автор использует для нумерации программ в книге и в электронном архиве цифробуквенную запись: 3a, 6a, 6b и т. д., мы же нумеруем программы в книге более привычной нашему читателю цифровой записью: 3.1, 6.1, 6.2 и т. д.

Бесплатно скачать с этого сайта можно и PDF-файлы некоторых из моих книг. Это книги, которые были опубликованы до 2004 года, т. е. до эры электронных книг и самопубликации. И хотя я написал многие из них (фактически все, кроме двух первых), используя текстовый процессор, исходный их текст давно утрачен (первые два были вообще написаны от руки — синими чернилами на листах формата А4 в линейку!)

Благодарности

Спасибо Ричарду Хури за его помощь с кодом C в этой книге и его мастерство работы с GCC и GDB. Спасибо Майку Гиннсу за идеи некоторых приведенных в книге программ. Некоторые листинги взяты из его книги Archimedes Assembly Language, которая впервые была опубликована Dabs Press в 1988 году (представьте, сколько лет уже существует ARM!) Кроме того, я благодарен Брайану Скалану, Стиву Сирелли и Тони Палмеру за их отзывы. Я также признателен многим читателям, которые присылали мне предложения по улучшению этой книги.

2. Начало

Ассемблер как язык позволяет вам обращаться к родному языку Raspberry Pi — *машинному коду*. Это собственный язык ARM-чипа, который по сути есть разум и душа вашей компьютерной системы. ARM расшифровывается как Advanced RISC Machine (продвинутая машина с сокращенным набором команд), и именно этот чип заведует всем, что происходит с вашей Raspberry Pi.

Микропроцессоры, такие как ARM, управляют использованием и передачей данных. Процессор также часто называют ЦП — *центральным процессором*, и данные, которые обрабатывает ЦП, текут сквозь него в виде непрерывного, почти бесконечного потока единиц и нулей. Порядок этих единиц и нулей не случайный, и определенная их последовательность превращается в набор определенных действий. Это похоже на азбуку Морзе, где правильная последовательность точек и тире превращается в осмысленные буквы и слова.

-. - (здесь закодировано слово CLEVER, умный)

Числа со смыслом

Сама программа, написанная машинным кодом, представляет собой бессчетное множество строк из единиц и нулей. Вроде вот этого:

```
11010111011011100101010100001011
01010001011100100110100011111010
01010100011001111111001010010100
10011000011101010100011001010001
```

Понять, что именно означают эти числа, почти невозможно (или возможно, но потребовало бы очень много времени). Ассемблер помогает преодолеть эти проблемы.

Язык ассемблера — это своего рода сокращенная форма, которая позволяет писать программы машинного кода с помощью английской лексики. А *ассемблер* — это программа, которая переводит программу на языке ассемблера в машинный код, избавляя нас от трудоемкой и рутинной расшифровки цифр. Программа на языке ассемблера часто представляет собой обычный текстовый файл, который читается

компилятором и преобразуется в двоичный эквивалент (из единиц и нулей). Программа на языке ассемблера называется *входным*, или *исходным*, файлом, а машинный код — *объектным файлом*. Ассемблер переводит (или компилирует) исходный файл в объектный файл.

Язык ассемблера написан с использованием мнемоники. *Мнемоника* — это механизм быстрого обучения и запоминания, который основывается на ассоциациях между легко запоминающимися последовательностями букв и информацией, которую нужно запомнить. В естественном языке ту же функцию выполняют аббревиатуры. Например, чтобы запомнить цвета радуги, вы можете использовать фразу: «Каждый Охотник Желает Знать, Где Сидит Фазан», и взять первую букву каждого слова.

В среде SMS-сообщений и интернет-чатов тоже возник свой мнемонический язык. Благодаря ему текстовые сообщения становятся короче и компактнее. Например, «мп» может означать «мои поздравления», «спс» — «спасибо», а «дв» — «до встречи».

Команды ARM

У микросхемы ARM есть определенный набор команд машинного кода, которые она понимает. Именно таким операциям (или «кодам операций») и их использованию посвящена эта книга. ARM — это всего лишь один из типов микропроцессоров. Существуют и другие типы, и у каждого из них свой уникальный набор команд.

Нельзя просто так взять программу машинного кода, написанную для ARM, и успешно запустить ее на другом микропроцессоре. Она либо не сработает так, как ожидалось, либо вообще не запустится. Но понятия и концепции, о которых мы поговорим, будут применимы в работе с большинством других видов микропроцессоров и в аналогичных задачах. Если вы научитесь программировать на одном ассемблере, с программированием на других будет уже проще. По сути, в новом языке нужно будет просто выучить новый набор команд, причем многие из них будут похожи на уже известные вам.

Микропроцессоры в ходе своей работы перемещают и обрабатывают данные, поэтому неудивительно, что многие команды машинного кода посвящены именно этим операциям и у большинства *наборов команд* (собирательный термин для этих мнемоник) есть команды для сложения и вычитания чисел. Мнемоника языка ассемблера, используемая для этих операций, обычно выглядит так:

ADD
SUB

Это простой пример, и многие другие мнемоники ARM тоже просты, если рассматривать их в отдельности. Но вот последовательность строк, объединенная в цельную программу, выглядит уже сложнее. Разбив программу на составные части, можно без труда понять, как все работает.

Мнемоника языка ассемблера обычно состоит из трех символов, но иногда их бывает и больше. Как и в любом новом деле, вам нужно будет немного привыкнуть, но, если вы проработаете примеры, приведенные в этой книге, и примените их в своих собственных экспериментах, у вас не должно возникнуть особых проблем.

Так, `MOV` — мнемоника команды `MOVE` (переместить). Эта команда берет информацию из одного места и перемещает ее в другое место. Проще простого, да?

Процесс преобразования

Когда вы разработали свою программу на языке ассемблера, ее нужно преобразовать в машинный код с помощью *компилятора ассемблера*. Например, когда ассемблер встречает мнемонику `MOV`, он подставит вместо нее число, которое для процессора представляет собой инструкцию. Ассемблер сохраняет собранный машинный код в виде файла в памяти, после чего этот код можно будет запускать или выполнять. В процессе сборки программы ассемблер также проверяет ее синтаксис, чтобы убедиться в том, что все написано правильно. Если он обнаружит ошибку, то сообщит вам об этом и скажет ее исправить. Исправив проблему, вы можете еще раз попробовать собрать программу. Обратите внимание, что проверка синтаксиса гарантирует лишь правильность написания самих команд ассемблера. Но никто не проверит за вас логику программы, и поэтому, если вы написали сами команды правильно, но суть изложили неверно, программа соберется, но сработает неверно. Например, вам нужно сложение, а вы вместо этого вы запрограммировали вычитание!

Написать программу на ассемблере можно разными способами. Первые чипы ARM были разработаны `Acorn` и, как и стоило ожидать, они появились в ряде компьютеров на базе `Acorn`, работающих под управлением ОС `RISC`. К ним относились `Archimedes` и `RISC PC`. На этих машинах использовался компилятор `BBC BASIC`, который был чем-то новым в том смысле, что позволял писать программы на языке ассемблера как расширение `BBC BASIC`. Вы и сейчас можете писать этим методом, установив `RISC OS` на свою `Raspberry Pi`.

Как вы уже поняли, в этой книге мы предполагаем, что вы используете операционную систему `Raspberry Pi` и программное обеспечение `GNU GCC`. Есть и другое ПО для ассемблера, причем большая часть его бесплатна, и вы без труда найдете его в Интернете. Основным преимуществом программирования на `GCC` является то, что этот компилятор также может собирать программы, написанные на языке `C`.

Эта книга не посвящена программированию на `C`, но по ряду причин нам будет полезно познакомиться с некоторыми его моментами, и даже эта малость может помочь вам в написании программ. Подробнее мы поговорим об этом в книге на примере позже.

В целом вам ничто не мешает попробовать любой другой или вообще все ассемблеры, и знания, которые вы получите в этой книге, помогут вам в этом.

А зачем вообще машинный код?

На этот вопрос ответить нетрудно. По сути, все, что делает ваш Raspberry Pi, выполняется с использованием машинного кода. Программируя напрямую в машинном коде, вы работаете на самом фундаментальном уровне работы Raspberry Pi.

Когда вы используете язык высокого уровня, такой как BBC BASIC или Python, все его операции все равно превращаются в машинный код всякий раз, когда вы запускаете программу. Это занимает некоторое время, хотя для нас эти доли секунды и незаметны. Но тем не менее процесс преобразования, или интерпретации, замедляет работу программного обеспечения. Фактически даже самые эффективные языки могут быть более чем в 30 раз медленнее, чем их эквивалент машинного кода, и это еще если повезет!

Если же вы программируете на машинном коде, программа будет работать намного быстрее, поскольку здесь преобразование не выполняется. Когда вы запускаете ассемблер, процесс преобразования происходит один раз, но однажды преобразованный машинный код можно многократно запускать напрямую. Необязательно запускать ассемблер каждый раз. Если вы довольны работой своей программы, то можете сохранить машинный код и использовать именно его. Вы также можете сохранить исходную программу на языке ассемблера и работать с ней, особенно если хотите что-то поменять в ее коде позже.

Языковые уровни

Языки вроде C или Python называются *языками высокого уровня*. На языках высокого уровня часто легче писать, поскольку их синтаксис похож на английский язык, а еще в них есть команды, которые выполняют большую и сложную последовательность действий, для подробного расписывания которой потребовался бы длинный список команд машинного кода. Машинный код — это *язык низкого уровня*, поскольку он работает в самых внутренностях компьютера, описывая каждое действие, и из-за этого его труднее понимать.

Это, конечно, преимущество языков высокого уровня по сравнению с языками низкого уровня. Но пока вы набираетесь опыта в языке ассемблера, ничто не мешает вам создавать библиотеки подпрограмм для выполнения конкретной задачи и просто добавлять их в свои программы по мере их написания. Впрочем, если вы углубитесь в мир ARM, то выяснится, что нужные библиотеки уже существуют.

Написанные на ассемблере программы легко переносить на другие компьютеры или системы, работающие на чипе ARM. Нужно лишь загрузить файл ассемблера в компилятор на новой машине, собрать код в программу машинного кода и запустить ее.

Компилятор GNU GCC есть практически на всех разновидностях микропроцессоров, поэтому, познакомившись с GCC, вы сможете перенести свои новые навыки на другие системы, если захотите.

Если раскрыть возможности чипа ARM на полную катушку, вы даже можете напрямую передавать и запускать машинный код. Эти возможности невероятно кру-

ты, особенно учитывая тот факт, что почти каждый существующий сегодня смартфон или планшет работает на чипах ARM!

На орбиту!

Чтобы еще раз подчеркнуть возможности чипа ARM и смартфонов в целом, вспомним, что на орбиту Земли было выведено целое новое поколение спутников под названием CubeSats (рис. 2.1). Они небольшие (порядка 10 см) и выполняют определенные задачи. Космический центр Суррея на юге Великобритании разработал несколько спутников CubeSat, которые работают на смартфонах Android. Эти спутники стоят около 100 тыс. долларов каждый, что гораздо меньше, чем стоимость аппаратов прошлого поколения. Опять же, вычислительная мощность одного современного смартфона, возможно, в десятки тысяч раз больше, чем у компьютеров всех лунных миссий Аполлона, вместе взятых! И вся эта мощь — вот она, прямо в Raspberry Pi.

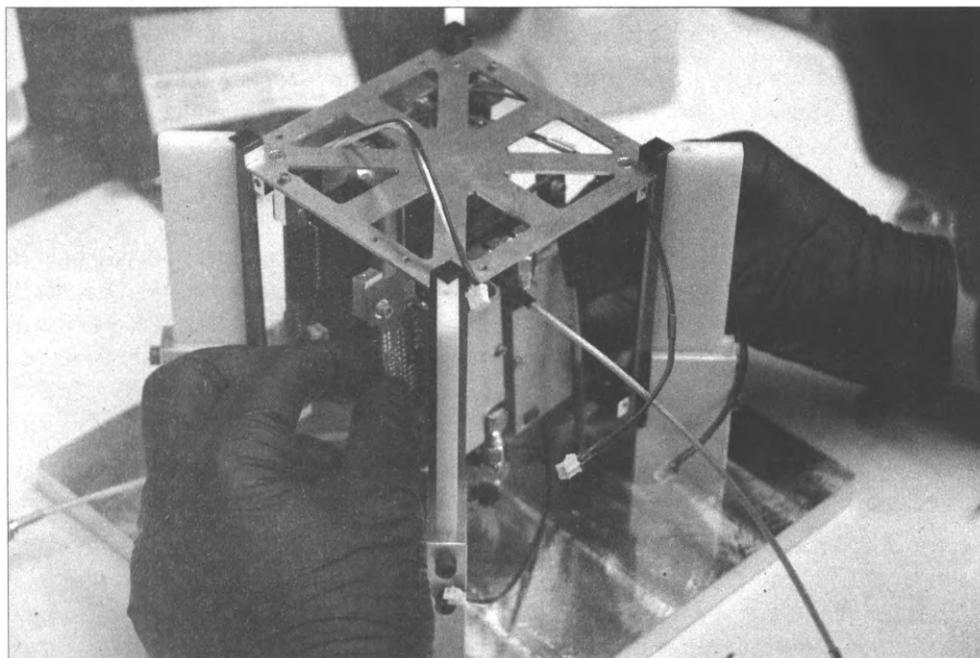


Рис. 2.1. Изготовление CubeSat

Но не все коту масленица! У плат есть различия в версиях ЦП. Как и в случае с программным обеспечением, чип ARM постоянно совершенствовался и возникали проблемы с новыми версиями. Однако базовый набор команд остается прежним, поэтому «перенос» будет не так сложен, как может показаться. Реальной проблемой он становится только в том случае, если вы используете более продвинутые функции микропроцессора. Для нашего вводного руководства эти изменения не актуальны, так что в этой книге все подойдет к вашей Raspberry Pi.

RISC и наборы команд

Буква R в аббревиатуре ARM означает RISC. Это тоже сокращение от Reduced Instruction Set Computing (вычисления с сокращенным набором команд). Все процессоры работают на машинном коде, и каждая из команд машинного кода или код операции выполняет свою определенную задачу. Объединенные вместе, команды образуют набор команд.

Идея RISC заключалась в создании небольшого оптимизированного набора команд. У этого подхода есть несколько преимуществ — меньше придется запоминать. Но при этом образуеться большее разнообразие в их использовании.

Структура ассемблера

Программирование на любом языке, как и общение на любом языке, подчиняется некоторому ряду правил. Эти правила определяются структурой и синтаксисом языка, который мы используем. Чтобы писать программы, нам нужно знать синтаксис языка и правила, которые определяют структуру программы.

Самый простой способ разработать программу — просто написать список задач, который она должна выполнить. Программа начинается с начала и выполняется команда за командой, пока не дойдет до конца. Другими словами, все команды выполняются по очереди, пока не будет достигнут конец программы. Это работает, но весьма неэффективно.

Современные языки более структурированы и позволяют писать программы в виде множества независимо выполняемых процедур, или подпрограмм. Эти подпрограммы вызываются из основной программы по мере необходимости. Таким образом, основная программа управляет потоком выполнения и запускает операции, которые подпрограммам недоступны.

Программы, написанные с помощью подпрограмм, получаются меньше и лучше поддаются редактированию. В линейной программе нам, вероятно, пришлось бы много раз повторять большие участки кода, чтобы выполнить поставленную задачу.

В листинге 2.1 показан некоторый псевдокод, на примере которого я иллюстрирую, как может выглядеть структурированная программа. В этом примере команды программы написаны прописными буквами. Разделам кода подпрограммы даются имена, которые пишутся в нижнем регистре и, как это ни парадоксально, с точкой в начале записи. Весь поток программы содержится в шести строках, начинающихся с раздела `.main` и заканчивающихся словом `END`. Программа получается короткой, но читается ясно, и с первого взгляда становится понятно, что в ней происходит. Подпрограммам специально даются осмысленные имена.

В этом примере основная программа просто вызывает некоторые подпрограммы. В идеальном мире мы бы писали именно такие программы, поскольку такой подход также упрощает тестирование этих подпрограмм по отдельности, прежде чем мы включим их в основную программу. Это позволяет гарантировать, что наша программа будет работать правильно.

Листинг 2.1. Псевдокод, иллюстрирующий структурированный подход к программированию

```
.main
    DO getkeyboardinput
    DO displayresult
    DO getkeyboardinput
    DO displayresult
END
.getkeyboardinput
    ; Instructions to read input from keyboard
RETURN
.displayresult
    ; print the result on the screen
RETURN
```

Ошибки на пути

Есть довольно серьезная проблема, с которой вы обязательно столкнетесь при изучении любой новой программы, — поиск ошибок. Этот процесс еще называют *отладкой*. Я гарантирую (и многократно убеждался в этом), что первый вариант вашей программы всегда будет работать неправильно. Вы будете медитировать над кодом до вечерней зорьки, но ошибки так и не найдете. Затем вы будете настаивать на своей правоте и валить все на компьютер. И лишь потом на вас снизойдет озарение, и вы увидите, что ошибка все это время была на самом видном месте.

Создав подпрограмму, а затем протестировав ее отдельно и убедившись, что она работает, вы будете знать, что и в основной программе все будет работать правильно и что сединой вы раньше времени не обзаведетесь.

Кросс-компиляторы

С этим термином вы, вероятно, будете сталкиваться часто. Компилятор GCC встречается на множестве компьютеров, и даже на тех, которые не работают на чипе ARM. Вы можете писать и скомпилировать ассемблер ARM вообще на другом компьютере! Но запустить собранный машинный код уже не сможете. Придется сперва перенести его с главной машины на целевую (например, Raspberry Pi). GCC — не единственный компилятор для Raspberry Pi и не единственный кросс-компилятор. Есть и много других. Загляните на форумы на сайте Raspberry Pi для получения подобной информации, и еще можете поискать в Интернете, если интересно.

Чипы Raspberry Pi ARM

Чип ARM, используемый в Raspberry Pi Zero, A, B, A+, B+ — это (приведем полное название) мультимедийный процессор Broadcom BCM2835 System-on-Chip. Термин «System-on-Chip» (система на кристалле, SoC) означает, что чип содержит почти

все необходимое для запуска Raspberry Pi в единой структуре (и именно поэтому у Raspberry Pi столь малые габариты). В BCM2835 используется дизайн ARM11, который построен на наборе команд ARMv6.

В Raspberry Pi 2 используется чип SoC BCM2836. В нем есть все функции BCM2835, но один ARM11 с тактовой частотой 700 МГц заменен на четырехъядерный ARM Cortex-A7 с тактовой частотой 900 МГц, а все остальное остается прежним. Этот чип быстрее, в нем больше памяти, и он может запускать достаточно серьезное программное обеспечение, такое как Windows 10 и весь спектр дистрибутивов ARM GNU/Linux.

В основе Raspberry Pi 3 лежит чип ARM v8, опять же с использованием структуры SoC и с еще большей частотой 1,2 ГГц. В его основе четыре высокопроизводительных процессора ARM Cortex-A53, работающих в тандеме. А еще это 64-битный процессор, который может работать как в состоянии AArch32, так и в AArch64.

В Raspberry Pi 4 используется Broadcom 2711, который стал еще быстрее — 1,5 ГГц. ARMv8 — это четырехъядерный процессор A72, 64-битный процессор, который может работать как в состоянии AArch32, так и в AArch64. Версия 8 Гбайт позволяет использовать память полной 64-разрядной версии для эффективной работы и обработки приложений — ничуть не хуже обычного ПК!

Что-то я увлекся терминами. Пока вы еще новичок, не стоит слишком забивать голову. По мере того как вы начнете узнавать больше о Raspberry Pi, все эти вещи станут вашей второй натурой. Мы вернемся к SoC в конце книги и объясним эту концепцию более подробно.

Кстати, частота в один мегагерц (1 МГц) соответствует миллиону циклов в секунду. Скорость микропроцессоров, называемая *тактовой частотой*, часто измеряется в МГц. Например, микропроцессор, работающий на частоте 700 МГц, выполняет 700 млн циклов в секунду. 1,2 ГГц (гигагерц) — это 1,2 млрд циклов в секунду. Позже мы увидим, как эта скорость влияет на выполнение команд.

Еще прозвучал термин «четырехъядерный». Четырехъядерный процессор имеет четыре независимых блока, называемых «ядрами», которые одновременно читают и выполняют команды. В целом четырехъядерный процессор будет работать быстрее, чем двухъядерный или одноядерный. Каждая открытая программа может работать на собственном ядре, поэтому, если задачи разделены между ядрами, скорость будет лучше. Эта так называемая параллельная обработка является основной особенностью ARM.

3. Проба пера

В этой главе мы пошагово рассмотрим процедуру создания и запуска программы машинного кода, начиная с момента включения Raspberry Pi и заканчивая внесением корректив в уже готовую программу. Первая программа не будет делать ничего фантастического. На самом деле не произойдет вообще ничего, кроме возврата символа приглашения, но даже в написание этой программы вовлечен каждый шаг, который вам нужно знать и использовать при создании и запуске других программ из этой книги.

Прямо сейчас я предполагаю, что у вас есть копия образа Raspberry Pi OS (Raspbian) на SD-карте, вставленной в ваш Raspberry Pi, и что вы использовали ее хотя бы один раз, чтобы выполнить начальную настройку таких важных моментов, как клавиатура и Интернет. Если вы еще этого не сделали, сделайте это сейчас, чтобы можно было продолжить.

Командная строка 3. Проба пера

При первой загрузке вы автоматически войдете в систему и попадете на рабочий стол. В правом верхнем углу выполните двойной щелчок мышью на значке монитора. Перед вами откроется окно **Terminal**, и вы попадете в командную строку, где увидите примерно такое приглашение ввода:

```
pi@raspberrypi $
```

Командная строка — это консоль, в которую вы вводите команды, которые затем выполнит ваша ОС. Командная строка начинается там, где мигает курсор. Консоль ожидает, что вы будете вводить команды, которые затем можно выполнить нажатием клавиши <Return> (также называемой клавишей <Enter>). Попробуйте ввести вот такую команду:

```
dir
```

Ввести ее нужно в точности так, как показано здесь. При нажатии клавиши <Return> перед вами появится список всех каталогов или файлов, которые находятся в текущем каталоге. (С этого момента я не буду каждый раз говорить о нажатии

клавиши <Return>, но вы имейте в виду, что если нужно ввести что-то с клавиатуры, особенно в командной строке, нужно будет нажать клавишу <Return>.)

Теперь введите вот это:

```
Dir
```

Вы получите такой ответ:

```
bash: Dir: command not found
```

Это *сообщение об ошибке*. В командной строке регистр имеет значение, поэтому команды:

```
dir
```

и

```
Dir
```

ОС считает разными, если учитывает регистр. То же самое и с именами файлов:

```
program1
```

и

```
Program1
```

— это разные файлы.

По соглашению о написании команд в командной строке всегда используется нижний регистр. Команды чувствительны к регистру. А в именах файлов могут быть и строчные, и прописные буквы, если вы понимаете разницу. Лучше всегда использовать строчные символы, поэтому не забываем поглядывать на индикатор клавиши <CapsLock>.

Создание исходного файла

Чтобы создать программу с машинным кодом, нужно выполнить процесс из трех действий «написать — собрать — связать», в результате которого мы получим файл, который можно будет запустить. Первый шаг — написать программу на ассемблере. Поскольку именно этот код является источником программы, файл называется *исходным* файлом. У такого файла будет расширение `s` после имени. Например:

```
program1.s
```

Исходные файлы можно писать в любом удобном текстовом редакторе. Существует много отличных и бесплатных редакторов, поэтому стоит уделить время изучению их обзоров и проверить пару вариантов самостоятельно. Возможно, у вас уже есть любимый редактор, и с этим этапом вы разобрались.

В Raspberry Pi OS уже есть набор редакторов, установленных в разделе **Recommended Software**, и вы можете найти их в главном меню приложения (пункт **Raspberry** в меню на рабочем столе). Скорее всего, нужные вам редакторы находятся в списке **Accessories**. Сюда входят (на момент подготовки книги) редакторы Vim, gVim, а также редактор Geany Programmer's Editor (советую попробовать каждый из них и выбрать тот, который вам больше нравится).

Если ни Vim, ни gVim в системе не установлены, вы можете исправить эту проблему с помощью параметра **Recommended Software** или из командной строки, набрав команду:

```
sudo apt-get install vim
```

После этого надо будет ответить на пару запросов: вас могут спросить о необходимости дополнительных функций. Ответ *y* вполне подойдет. Установка займет несколько минут, а на сайте Vim тем временем можно найти много полезных советов.

Если вам больше нравится работать в приложении, возьмите версию Vim с графическим интерфейсом и при необходимости установите ее командой:

```
sudo apt-get install vim-gtk
```

Поскольку вам предстоит провести много времени за программированием на ассемблере, имеет смысл уделить внимание изучению тонкостей Vim. Многие действия и операции, используемые в Vim, выполняются с помощью удобных комбинаций клавиш (особенно в консольной версии). В табл. 3.1 приведены некоторые команды, которые вам необходимо знать. Эта таблица ни в коем случае не является исчерпывающей, а полный набор комбинаций можно найти на сайте Vim. Но для начала хватит и этого.

Таблица 3.1. Важные команды редактора Vim

Клавиша	Действие
Команды перемещения курсора	
←	Переместить влево
↓	Переместить вниз
↑	Переместить вверх
→	Переместить вправо
w	Переместить до следующего слова
W	Переместить до следующего слова, отделенного пробелом
e	Переместить в конец слова
E	Переместить в конец слова (без учета пунктуации)
b	Переместить до предыдущего слова
B	Переместить до предыдущего слова (без учета пунктуации)
0 (ноль)	Новая строка
^	Первый непустой символ
\$	Конец строки
G	Перейти к команде (например, 5G -- перейти к строке 5) <i>Примечание:</i> в команде перехода к команде слева приписывается количество повторений. Например, команда 4j перемещает курсор на 4 строки

Таблица 3.1 (окончание)

Клавиша	Действие
Режим вставки — вставка/дополнение текста	
i	Переход в режим вставки с позиции курсора
I	Вставить в начале строки
A	Вставить после курсора
a	Вставить в конце строки
o	Добавить пустую строку после текущей (без нажатия клавиши <Return>)
O	Добавить пустую строку перед текущей
Esc	Выход из режима вставки
Режим команд	
:w	Запись (сохранение) файла без выхода
:wq	Запись файла (сохранение) и выход из Vim
:Q	Выход (не срабатывает при наличии изменений)
:Q!	Выход без сохранения изменений
:set	Установка нумерации строк

Запустив Vim, вы также можете указать имя файла, который хотите создать. Если файл уже существует, он загрузится в окно редактора, и вы сможете работать с ним, как вам нужно. Если такого файла не существует, Vim создаст для вас новый пустой файл с указанным именем. Откройте новое окно терминала и в командной строке введите:

```
vim prog3a.s
```

Обратите внимание, что между vim и prog3.s есть пробел, а также заметьте, что в конце имени prog3 есть расширение s исходного файла. Соглашение гласит, что расширение s обозначает исходный файл на языке ассемблера.

После этого окно станет в основном пустым, за исключением столбца из символов тильды ~, бегущего по левому краю, и имени файла внизу, а также обозначения того, что это новый файл.

Нажмите клавишу <i>. Обратите внимание что в нижнем левом углу экрана появился текст:

```
-INSERT --
```

Это означает, что мы находимся в режиме вставки. Нажмите клавишу <Esc>. Надпись исчезла. Теперь мы находимся в командном режиме Vim.

Нажатие клавиш <i> и <Esc> станет для вас второй натурой. Когда включен режим вставки, вы можете вводить программу на ассемблере и редактировать ее, пока не

надоест. В командном режиме нажатия клавиш интерпретируются как прямые команды для Vim.

Имена файлов программ — в нашем случае `prog3a.s` — станут для вас привычными. Все программы в этой книге будут именоваться по номерам глав. Таким образом, `prog3a.s`¹ означает, что исходный файл программы взят из главы 3 книги. Буква `a`¹ предполагает, что это 1-я программа в этой главе. Файл с именем `prog4b.s`¹ будет означать, что это 2-я программа из главы 4. Мы будем делать так просто для удобства. Но вы можете использовать любое имя, какое захотите.

Вернитесь в режим вставки (клавишей `<i>`) и обратите внимание на мигающий курсор в верхнем левом углу экрана. Все, что вы начнете вводить, появится там, где находится курсор. Перепишите туда код из программы 3.1. Вам нужно ввести только тот текст, который в этом листинге расположен между двумя пунктирными линиями.

ПРИМЕЧАНИЕ К РУССКОМУ ИЗДАНИЮ

Напомним, что электронный архив с файлами приведенных в книге программ можно загрузить с FTP-сервера издательства «БХВ» по ссылке: <ftp://ftp.bhv.ru/9785977568012.zip> или со страницы книги на сайте <https://bhv.ru/> (см. приложение 4).

ПРОГРАММА 3.1. Простой исходный файл

```
.global _start
start:
MOV R0, #65
MOV R7, #1
SWI 0
```

Обратите внимание, что из этих пяти строк программы первая, третья, четвертая и пятая строки написаны с отступом. Только вторая строка написана без отступа. Величина отступа, который вы добавляете, и даже место их размещения, на самом деле не имеют значения. Но зато они просто упрощают чтение программы и позволяют выделить различные уровни программы.

Чтобы создать отступ, нажмите клавишу `<Tab>`. Другие клавиши работают так, как и должны. Например, клавиши со стрелками используются для перемещения, а клавиши `<Delete>` и `<Backspace>` — для перемещения и редактирования текста. Но между словами `global` и `_start` имеется пробел, и этот пробел важен. Вскоре мы рассмотрим, что конкретно делает этот код.

А пока нажмите клавишу `<Esc>` и введите:

```
:wq
```

Эта команда сохранит ваш файл и закроет Vim. Теперь вы вернулись в командную строку. Исходный файл готов!

¹ Здесь имеется в виду авторская нумерация программ в книге и в электронном архиве (3a, 3b, 4a, 4b). Мы же в книге нумеруем программы более привычной нашему читателю цифровой записью: 3.1, 3.2, 4.1, 4.2 и т. д. (в электронном архиве нумерация остается авторской).

Написанное — исполнить!

Следующим шагом является преобразование исходного файла в исполняемый файл машинного кода. Это делается с помощью двух команд командной строки. В командной строке последовательно введите две команды:

```
as -o prog3a.o prog3a.s
ld -o prog3a prog3a.o
```

Эти две команды сначала собирают, а затем *привязывают* программу на языке ассемблера (о привязывании позже). После этого машинный код готов к выполнению:

```
./<имя файла>
```

Сочетание символов `./` означает «запустить», а имя файла, который нужно запустить, пишется сразу после команды без пробелов. Таким образом, для запуска напишем:

```
./prog3a
```

Когда приглашение появится снова, программа машинного кода будет выполнена. Проще простого!

Выходит, мы только что написали, скомпилировали (собрали и связали) и выполнили программу машинного кода, сделав все основные шаги этого стандартного процесса. Конечно, со временем наши программы будут становиться все сложнее, мы будем шире использовать доступные инструменты, и этот процесс, как мы увидим, станет более сложным.

Ошибки ассемблера

Если в какой-либо момент во время процесса вы получите сообщение об ошибке, или вообще какое-либо сообщение, — тщательно проверьте то, что вы ввели. Сначала внимательно посмотрите на программу на языке ассемблера, затем на команды для сборки и связывания и, наконец, запустите программу. Если произошла ошибка, и вы ее нашли, поздравляем, вы только что отладили свою первую программу на ассемблере.

Если возникает сообщение об ошибке от ассемблера (оно появляется после того, как вы нажали `<Return>` в конце первой строки), обычно в этом сообщении указывается номер строки, в которой что-то неладно. Даже если вы не знаете, что именно означает полученное сообщение, запишите себе номер строки, а затем откройте исходный файл в Vim. Например, сообщение:

```
prog3a.s:5: Error bad expression
```

говорит об ошибке в строке 5 исходного файла.

Пока ваши файлы еще крошечные, вы можете отсчитать количество строк и найти ту, в которую закралась ошибка. В самом редакторе Vim также есть возможность нумерации строк. В командном режиме Vim введите:

```
:set number
```

Обратите внимание, что в левой части окна появились номера строк. Номера строк не являются частью файла с кодом и отображаются только для справки. На рис. 3.1 показано, как это все это выглядит в редакторе gVim: видны номера строк и то, что Vim работает в режиме вставки. Если вы используете gVim, то в нем различные элементы синтаксиса выделяются разными цветами. Это позволяет легко идентифицировать различные компоненты программы.

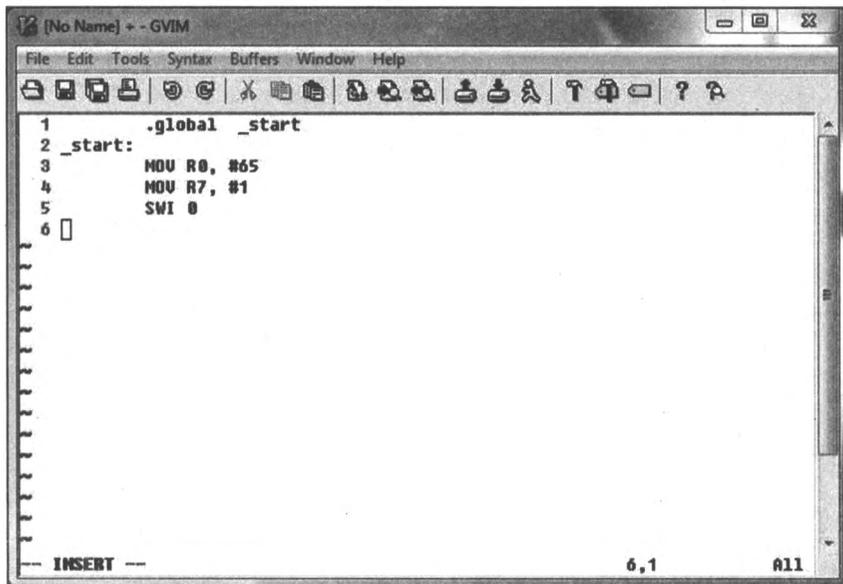


Рис. 3.1. Отображение номеров строк в gVim

Вы можете запустить редактор gVim из командной строки с помощью следующей команды:

```
gvim <имяфайла>
```

Тогда для создания или редактирования файла `prog3a.s` можно выполнить команду:

```
gvim prog3a.s
```

Компоненты

Давайте еще раз взглянем на описанный здесь процесс и попробуем вникнуть в анатомию исходного файла и понять, как все это собралось воедино. Посмотрим еще раз на файл `prog3a.s`. Он состоит всего из пяти строк.

У любого файла с кодом ассемблера должна быть точка начала, и по умолчанию в ассемблере GCC она выглядит вот так:

```
_start:
```

Первая строка этой программы определяет `_start` как глобальное имя, доступное для всей программы. Позже мы увидим, почему важно сделать это имя глобальным.

Вторая строка определяет, где находится `_start`: в программе. Обратите внимание на использование символа `:` в конце, которое определяет это имя как «метку». Мы определили `_start` как глобальное имя и теперь отметили, где находится сама метка `_start`.

Следующие три строки написаны на мнемонике ассемблера, причем две из них одинаковые: команда `MOV`. Когда в языке ассемблера используется символ решетки, он обозначает непосредственное значение. Другими словами, число после решетки — это именно то значение, которое и будет использоваться. Первая команда перемещает значение `65` в регистр `R0`. Буква `R` обозначает *регистр* — специальное место в микросхеме ARM, о котором мы поговорим чуть позже. Во второй строке значение `1` перемещается в `R7`, или в регистр `7`.

Последняя команда: `SWI 0`. Это специальная команда, которая служит для вызова самой операционной системы Raspberry Pi. В нашем случае она задействуется для выхода из программы машинного кода и передачи управления обратно в командную строку (программа в этот момент должна быть запущена).

Также стоит обратить внимание на регистр символов команд языка ассемблера в наших исходных файлах. Я использую прописные буквы для мнемоник и регистров, но подошли бы и строчные буквы, поскольку внутри исходных файлов регистр символов не имеет значения (в отличие от командной строки в консоли, которая чувствительна к регистру), поэтому

```
MOV R0, #65
```

а также

```
mov r0, #65
```

делают одно и то же. В этой книге я буду использовать прописные буквы. Это упрощает чтение команд в тексте книги и позволяет отличить команды от меток, которые будут написаны в нижнем регистре.

Запустите программу еще раз:

```
./prog3a
```

В командной строке введите:

```
echo $?
```

На экран будет выведено следующее:

```
65
```

Вывелось значение, загруженное в `R0`. Попробуйте изменить `65` на другое число — скажем, `49`. Теперь сохраните код, соберите его заново, заново свяжите и запустите. Если вы сейчас наберете:

```
echo $?
```

будет выведено число `49`. У ОС есть определенные способы возврата информации из программ машинного кода, и мы рассмотрим их позже.

Если вы снова посмотрите на код в файле `prog3a.s`, то увидите, что он состоит из двух отдельных частей. Вверху (в начале) приведены некоторые определения,

а в нижней половине — непосредственно команды на языке ассемблера. Исходные файлы на языке ассемблера всегда состоят из последовательности операторов, записанных по одному в каждой строке. Каждый оператор имеет следующий синтаксис (при этом каких-то его частей может и не быть):

```
<обозначение:> <инструкция>           @ комментарий
```

Все три компонента можно вводить в одной строке или в разных строках. Это дело ваше. Но порядок их должен быть именно таким. Например, команда не может располагаться перед меткой (в той же строке).

Компонент «комментарий» для нас в новинку. Когда ассемблер встречает символ @, он игнорирует все написанное после него до конца строки. Такую конструкцию можно использовать для добавления в программу пояснений. Например, вернитесь и отредактируйте код в файле `prog3a.s`, набрав:

```
vim prog3a.s
```

В окне редактора отобразится исходный файл. Курсор будет находиться в верхней части файла. Войдите в режим вставки, создайте новую строку и введите в нее следующее:

```
@ prog3a.s - простенький файл ассемблера
```

Строка комментария с символом @ в начале полностью игнорируется компилятором. Из-за этого размер исходного файла, конечно, увеличится, но это никак не повлияет на работу исполняемого файла.

Комментарии также можно добавлять с помощью символов /* и */, обозначая ими начало и конец комментария:

```
/* Этот комментарий ассемблер проигнорирует */
```

Оба метода приемлемы, и вы можете использовать любой на свой вкус.

Чтобы преобразовать исходный файл в исполняемый, нам потребовалось два шага. Первый:

```
as -o prog3a.o prog3a.s
```

Команда `as` в начале вызывает саму программу ассемблера, которая ожидает после этой команды нескольких аргументов, задающих имена файлов, с которыми она будет работать, а также необходимые действия. Первый из них: `o` — сообщает ассемблеру, что мы хотим создать объектный файл по имени `prog3a.o` из исходного файла `prog3a.s`. Вы можете выбрать другое имя, если хотите. Сами имена не обязательно должны совпадать, но, если они одинаковые, проще найти концы. Расширение в любом случае разное!

Второй и последний шаг: «связать» файл объектного файла и преобразовать его в исполняемый файл с помощью команды `ld` следующим образом:

```
ld -o prog3a prog3a.o
```

Это действие — последняя часть процедуры связки, которая заставляет работать машинный код. В ходе нее из объектного файла, созданного в процессе сборки,

создается исполняемый файл (называемый *эльф-файлом*). Именно команда `ld` использует метку `_start:` и благодаря ей понимает, откуда должна запускаться программа (это может звучать безумно, но начальная точка не всегда находится в начале кода, как мы увидим чуть позднее).

А если нет метки `_start`?

Можно многое узнать о работе ассемблера и компоновщика GCC, просто поэкспериментировав с ними. Как вы думаете, что произойдет, если мы опустим метку `_start:` в исходном файле?

Откройте файл `prog3a.s` в Vim и удалите строку `_start:`, т. е. метку. Выйдите из Vim и соберите программу:

```
as -o prog3a.o prog3a.s
```

А теперь свяжите программу:

```
ld -o prog3a prog3a.o
```

Компилятор выдаст следующее сообщение об ошибке (или подобное):

```
ld: warning: cannot find entry symbol _start; defaulting to 00008054
```

Сообщение говорит само за себя. Поскольку компилятор не может найти указатель на то, где начинается программа, компоновщик предполагает, что точка запуска программы находится в самом начале, а в памяти это адрес `00008054`. (Этот адрес может и, вероятно, будет свой у каждой модели Raspberry Pi.)

Это подстраховка, но не гарантия безопасности. Всегда используйте метку `_start:` в своих файлах, чтобы определить входную точку программы!

Связывание файлов

Сочетание «`ld`» обозначает «динамическое связывание». Сама команда связывания позволяет объединять или последовательно соединять несколько файлов в одну длинную исполняемую программу. Для таких случаев в этих файлах должна быть определена только одна метка `_start:`, т. к. у объединенной программы может быть только одна точка входа. Это легко продемонстрировать на примере нашей программы.

Создайте новый файл в Vim и присвойте ему имя:

```
part1.s
```

В этот файл введите код, показанный в *программе 3.2*.

ПРОГРАММА 3.2. Часть 1 исходного файла

```
/* файл part1.s          */  
    .global _start
```

```
_start:
    MOV R0, #65
    BAL _part2
```

Сохраните файл. Теперь создайте новый файл с именем:

```
part2.s
```

и добавьте в него код из программы 3.3.

ПРОГРАММА 3.3. Часть 2 исходного файла

```
/* файл part2.s      */
    .global _part2
_part2:
    MOV R7, #1
    SWI 0
```

Сохраните и закройте файл. Мы написали два исходных файла, которые теперь скомпилируем и свяжем для создания одного исполняемого файла. В конце файла `part1.s` мы добавили новую инструкцию:

```
BAL _part2
```

Команда `BAL` означает безусловный переход — в нашем случае переход к точке программы, отмеченной меткой `part2:`. Во втором файле мы определили глобальную переменную с именем `part2` и отметили точку, где эта часть начинается. Поскольку мы использовали определение глобальных меток, местоположение адреса будет доступно для всех частей программы.

Теперь нужно скомпилировать оба новых исходных файла:

```
as -o part1.o part1.s
as -o part2.o part2.s
```

Затем идентифицировать и связать метки с помощью компоновщика следующим образом:

```
ld -o allparts part1.o part2.o
```

Здесь компоновщик создаст исполняемый файл с именем `allparts` из файлов `part1.o` и `part2.o`. (Вы можете использовать и другое имя.) Попробуйте запустить файл:

```
./allparts
```

Порядок указания файлов `part1.o` и `part2.o` можно поменять местами. Он не имеет значения, поскольку вопросами порядка выполнения занимается компоновщик. Ключевым моментом здесь является то, что каждый исходный файл пишется и создается независимо, но затем все они соединяются механизмом связывания. Если вы попытаетесь связать только один файл, вы получите сообщение об ошибке, потому что при связывании каждая часть ссылается на другую. То есть компоновщик в таком случае занимается безопасностью.

Этот пример показывает, что, если вы заранее продумаете, как будут выглядеть файлы с кодом, вы можете начать разработку библиотеки файлов, которой затем можно будет пользоваться каждый раз, когда вам понадобится конкретная функция. Если вы вспомните предыдущую главу и концепцию псевдопрограммы (см. листинг 2.1), ее можно было бы создать с помощью таких функций. Позже мы рассмотрим создание функции.

Прибираемся...

Если вы выведете корневой каталог командой:

```
dir
```

то увидите, что среди прочего в нем находятся три файла `prog3`, а именно:

- ◆ `prog3a.s` — исходный файл;
- ◆ `prog3a.o` — объектный файл.
- ◆ `prog3a` — запускаемый файл.

В общем-то, вам нужен только исходный файл, т. к. исполняемый файл можно создать из него в любой момент. Кроме того, можно избавиться от объектного файла с помощью команды `rm` (от `remove files`):

```
rm prog3a.o
```

Чтобы файлы хранились в порядке, стоит создать отдельный каталог для всех ваших файлов ассемблера с помощью команды `mkdir`. Чтобы создать каталог с именем `aal`, выполните команду:

```
mkdir aal
```

Теперь вы можете перейти в этот каталог и работать в нем, выполнив команду:

```
cd aal
```

Обратите внимание, что в строке приглашения появилась приписка:

```
/aal $
```

С этого момента все, что вы будете создавать или делать, будет производиться в каталоге `aal`. Чтобы вернуться в корневой каталог Raspberry Pi, введите команду:

```
cd
```

Обратите внимание, что в приглашении командной строки всегда написано, в каком каталоге файловой системы Raspberry Pi вы в текущий момент работаете.

Если при попытке загрузить или собрать файл, вы получите сообщение об ошибке, стоит проверить, находитесь ли вы в правильном каталоге и что имя файла тоже написано правильно (включая использование прописных букв). Зачастую проблема именно в этом!

Пара слов о комментариях

Не все со мной согласятся, но я считаю, что комментировать свои программы на ассемблере нужно обязательно. То, что вы пишете сегодня, в памяти свежо, но если нужно будет переделать или подредактировать код позже, вам почти наверняка будет трудно вспомнить, что именно делает каждый фрагмент кода, а другой программист, которому надо будет поработать с вашим кодом, вообще запутается. На мой взгляд, комментарии — если они по делу — являются неотъемлемой частью программ на языке ассемблера. Все ассемблеры позволяют писать комментарии в исходном файле. Комментарии не входят в конечный машинный код и не замедляют его выполнение. Единственный минус их в том, что они увеличивают размер вашей исходной программы.

Поэтому комментарии — это хорошо, но не следует писать их «лишь бы написать». Если писать комментарий в каждой строке, то код становится некрасивым и лишние комментарии отвлекают внимание от важных комментариев. Например, посмотрите на эту простую строку и комментарий к команде ADD:

```
ADD R0, R1, R2           @ R0=R1+R2
```

Комментарий здесь не имеет смысла с точки зрения программной документации, поскольку мы и так знаем, что делает эта операция. Уместнее было бы пояснить, что за значения (по смыслу) хранятся в ячейках R1 и R2. Лучше было бы написать вот так:

```
ADD R0, R1, R2           @ сложение act1 + act2
```

Если вы разбиваете свою программу на сегменты, используя формат, показанный в предыдущей главе, обычно бывает достаточно одного или двух комментариев в начале раздела. Дам несколько рекомендаций:

- ◆ комментируйте все ключевые моменты вашей программы;
- ◆ используйте простой язык. Не придумывайте секретные шифры, понятные только вам;
- ◆ если уж комментируете, то комментируйте как следует;
- ◆ делайте комментарии аккуратными, удобочитаемыми и последовательными;
- ◆ комментируйте все определения.

Помня об этих ключевых моментах, вы будете делать все правильно, и я не просто так говорю об этом в начале книги, надеюсь, что вы поймете суть и приобретете хорошие привычки, которые сохранятся на протяжении всей вашей карьеры программиста.

Если вы планируете писать много машинного кода, стоит задуматься о внешнем документировании файлов, создании базы данных или, возможно, книги, в которой будут храниться данные.

Кроме того, остерегайтесь синдрома множества файлов. В ходе процесса разработки у вас может возникнуть несколько их версий. И велик шанс запутаться, какая из

них какая. В результате вы наплодите целый зоопарк файлов с одинаковыми названиями. Неплохо бы добавлять комментарий в самом начале, в котором будет написано, почему вы перешли на новую версию. Переместите файлы, которые, по вашему мнению, вам не нужны, в папку для хранения. Найдя нужный файл, добавьте комментарий и назовите его соответствующим образом.

Редактор Geany Programmer's Editor

Ранее в этой главе я упоминал, что есть еще один редактор для создания файлов с кодом в Raspberry Pi. В более поздних версиях рабочего стола Raspberry присутствует подменю **Programming**, где вы, вероятно, найдете редактор Geany Programmer's Editor.

Geany (рис. 3.2) — это IDE, или «интегрированная среда разработки», в которой есть множество вариантов форматирования для различных проектов и языков. В ней также имеются вкладки, которые позволяют открывать несколько файлов с кодом. Автоматически указываются номера строк.

Если вы не забыли сохранить исходный файл как текстовый файл с расширением *s*, можете использовать этот редактор. Однако полезно сперва понять, как работают некоторые традиционные текстовые редакторы.

На рис. 3.2 также видно, что нумерация строк выполняется автоматически, а в окне Geany можно найти много разной дополнительной информации.

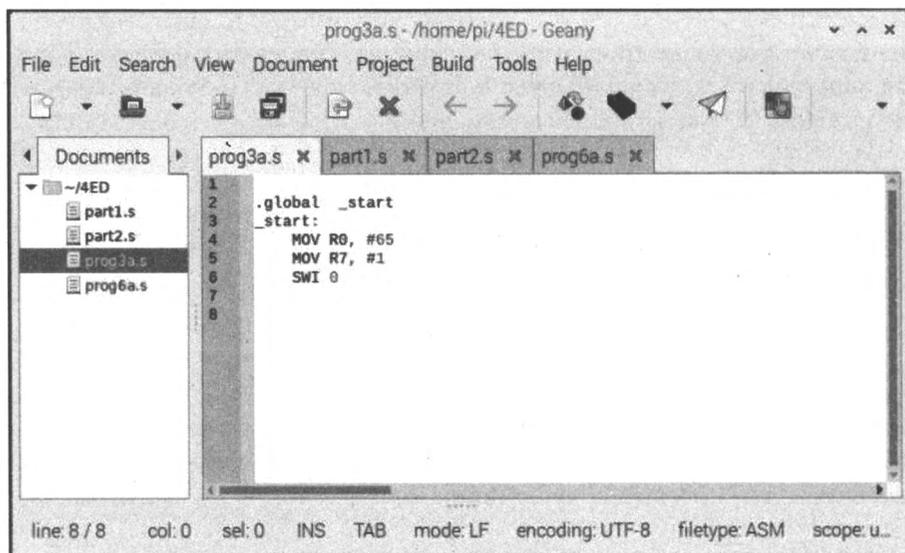


Рис. 3.2. Внешний вид Geany IDE

4. О битах в RISC-машинах

В мире есть 10 типов людей: те, которые понимают двоичную систему, и те, которые ее не понимают.

Если вы не поняли, что это шутка, не волнуйтесь. Прочитав эту главу, вы все поймете. Если же она вызвала у вас улыбку прямо сейчас, вы на правильном пути к изучению этого материала. Начнем мы с объяснения, что вообще такое двоичная система, как ею можно манипулировать и какое значение она имеет для эффективного написания машинного кода.

Создавая программы с машинным кодом, вы работаете на самом базовом уровне компьютера. Проще некуда. В предыдущих главах мы упоминали, что существуют двоичные и шестнадцатеричные числа. Шестнадцатеричные числа — это компактный способ записи чисел, которые в двоичном формате представляют собой длинные последовательности единиц и нулей. Архитектура Raspberry Pi с сокращенным набором команд может делать многое, используя базовый набор команд и извлекая максимум пользы из каждой единицы или нуля. Чтобы полностью понять, как работает RISC-машина, сначала нужно разобраться, как устроены двоичный и шестнадцатеричный файлы и как они используются микросхемой ARM.

В предыдущих главах мы говорили, что команды, с которыми работает процессор ARM, состоят из последовательностей чисел. Каждое число представляет собой инструкцию (код операции) или данные (операнд), некоторой операции машинного кода. В памяти компьютера эти числа представлены в двоичном виде. Двоичное число — это просто число, состоящее из единиц или нулей. Двоичный формат тем удобен, что внутри микропроцессора эти единицы и нули соответствуют состояниям «включено» и «выключено» (с точки зрения электроники это обычно напряжение +5 и 0 В), и нам, как программистам на языке ассемблера, нужно знать состояние отдельных двоичных цифр, или битов.

Коды операций и операнды получаются путем объединения наборов из восьми битов, которые вместе называются *байтами*. Согласно соглашению биты в этих байтах пронумерованы, как показано на рис. 4.1.

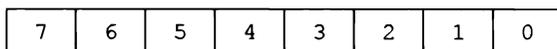


Рис. 4.1. Нумерация битов в байте

Увеличение номера идет справа налево (не слева направо), но это не так странно, как может показаться на первый взгляд.

Рассмотрим десятичное число 2934 — две тысячи девятьсот тридцать четыре. Наибольшее числовое значение, две тысячи, находится слева, а наименьшее, четыре, — справа. То есть положение цифры в числе весьма важно, т. к. оно влияет на «вес» числа.

Во второй строке на рис. 4.2 показано новое представление числа. К основанию системы приписывается суффикс с небольшим числом, или степенью, которая соответствует его положению в числе. Таким образом, 10^3 равняется $10 \times 10 \times 10 = 1000$. Число в нашем примере состоит из двух тысяч плюс девять сотен плюс три десятка плюс четыре единицы.

Значение	1000	100	10	1
Представление	10^3	10^2	10^1	10^0
Цифра	2	9	3	4

Рис. 4.2. Десятичные веса обычных чисел

В двоичном представлении вес каждого бита вычисляется путем возведения основания (2) в степень позиции бита. Например, бит номер 7 (b7) имеет условное представление 2^7 или $2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 = 128$. Вес, или значение, каждого бита показаны на рис. 4.3.

Номер бита	b7	b6	b5	b4	b3	b2	b1	b0
Представление	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Вес	128	64	32	16	8	4	2	1

Рис. 4.3. Двоичные веса чисел

Преобразование двоичных чисел в десятичные

Раз уж мы можем вычислить вес отдельных битов, значит, преобразовать двоичное число в десятичное будет несложно. У этого преобразования всего два правила:

- ◆ если бит равен 1, добавьте его вес к сумме;
- ◆ если бит равен 0, ничего с ним не делайте.

Давайте попробуем для примера преобразовать двоичное число 10101010 в его эквивалентное десятичное.

Глядя на рис. 4.4, складываем значения из третьего столбца и получаем в результате 170. Следовательно, двоичное число 10101010 эквивалентно десятичному 170 ($128 + 0 + 32 + 0 + 8 + 0 + 2 + 0$). Аналогично двоичное значение 11101110 эквивалентно 238 в десятичном виде, как показано на рис. 4.5.

Бит	Вес	Значение
1	128	128
0	64	0
1	32	32
0	16	0
1	8	8
0	4	0
1	2	2
0	1	0

Рис. 4.4. Преобразование двоичного числа 10101010 в десятичное

Бит	Вес	Значение
1	128	128
1	64	64
1	32	32
0	16	0
1	8	8
1	4	4
1	2	2
0	1	0

Рис. 4.5. Преобразование двоичного числа 11101110 в десятичное

Преобразование десятичных чисел в двоичные

Чтобы преобразовать десятичное число в двоичное, нужно выполнить обратную процедуру — вычитать двоичные веса из заданного числа. Если вычитание возможно, в двоичный столбец помещается 1, а остаток переносится в следующую строку. Если вычитание невозможно, в двоичный столбец помещается 0, а число перемещается в следующую строку. Например, на рис. 4.6 показано, как десятичное число 141 преобразуется в двоичное.

Следовательно, десятичное число 141 соответствует двоичному 10001101.

Десятичное	Вес	Остаток	Двоичное
141	128	13	1
13	64	13	0
13	32	13	0
13	16	13	0
13	8	5	1
5	4	1	1
1	2	1	0
1	1	0	1

Рис. 4.6. Преобразование десятичного числа 141 в его двоичный эквивалент

Преобразование двоичного числа в шестнадцатеричное

Двоичная запись, вероятно, лучше всего показывает, как хранятся числа в Raspberry Pi, но работать с такими числами весьма сложно. Во время чтения длинных рядов нулей и единиц глаза попросту разбегаются в разные стороны. Поэтому при работе

с двоичными числами мы чаще используем альтернативную форму их представления — шестнадцатеричную. Шестнадцатеричные числа — это числа с основанием 16. На первый взгляд такой вариант еще менее удобен, но на самом деле у него много преимуществ.

Для представления всех возможных цифр шестнадцатеричного числа нужны шестнадцать различных символов. Поэтому мы сначала используем привычные цифры от 0 до 9, а затем буквы A, B, C, D, E, F — для представления значений от 10 до 15. Двоичные и десятичные значения для каждого шестнадцатеричного числа показаны на рис. 4.7. Если вы внимательно читали о двоичных числах, то на рис. 4.7 можете найти пару интересных закономерностей.

Десятичное	Шестнадцатеричное	Двоичное
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

Рис. 4.7. Десятичные, шестнадцатеричные и двоичные числа

Обратите внимание, что четыре бита двоичного числа представляются одним шестнадцатеричным числом. То есть полный байт (8 двоичных разрядов) можно записать всего двумя шестнадцатеричными символами. В десятичной же системе счисления для байта потребуются три символа. Таким образом, шестнадцатеричная запись — это очень компактный и простой способ представления двоичного кода.

Чтобы преобразовать двоичное число в шестнадцатеричное, байт делится на два набора из четырех битов, называемых *полубайтами*, а затем мы берем соответствующее шестнадцатеричное значение каждого полубайта из таблицы, приведенной на рис. 4.7.

Превратим число 01101001 в шестнадцатеричное:

$$0110 = 6$$

$$1001 = 9$$

Ответ: 69. Поскольку не всегда очевидно, является ли число шестнадцатеричным или десятичным (69 может быть и десятичным), шестнадцатеричным числам обычно предшествует уникальный символ — например 0x (в этой книге мы будем писать именно так):

0x69

Обратным путем можно преобразовывать шестнадцатеричные числа в двоичные.

Преобразуем шестнадцатеричные числа в десятичные и обратно

Чтобы преобразовать шестнадцатеричное число в десятичное, достаточно сложить десятичный вес каждой цифры. Преобразуем число 0x31A в десятичное:

◆ 3 имеет вес $3 \times 16^2 = 3 \times 16^2 = 3 \times 16 \times 16 = 768$

◆ 1 имеет значение $1 \times 16^1 = 1 \times 16 = 16$

◆ A имеет значение $1 \times 16^0 = 10 \times 1 = 10$

В сумме получаем десятичное число 794.

Преобразование десятичного числа в шестнадцатеричное выполняется чуть сложнее: мы многократно делим исходное число на 16, пока не будет получен остаток меньше 16. Этот остаток записывается, а частное делится дальше. Процесс продолжается до тех пор, пока само частное не станет меньше 16.

Например, преобразуем десятичное число 4072 в шестнадцатеричное:

◆ $4072 / 16 / 16 = 15 = F$ Частное: $4072 - (15 \times 16 \times 16) = 232$

◆ $232 / 16 = 14 = E$ Частное: $232 - (14 \times 16) = 8$

◆ Частное = 8 = 8

Следовательно, десятичное число 4072 равно 0xFE8.

Оба преобразования достаточно сложны, поэтому становится ясно, почему проще сразу работать в шестнадцатеричном формате и забыть о десятичных числах. К этому вы привыкнете. Сейчас такой метод может показаться вам чуждым, но позже, когда вы разовьете свои знания ассемблера, он станет вам родным (а еще, если вы хотите преобразовать шестнадцатеричное в десятичное, вы можете использовать число пи!).

Двоичное сложение

Двоичные числа легко складывать и вычитать. На самом деле если вы умеете считать до двух, то все нужное вы уже знаете. В общем-то, не обязательно складывать и вычитать единицы и нули «вручную», но в этой главе мы все же рассмотрим некоторые важные концепции, которые помогут вам в освоении следующих глав и в конечном итоге в программировании на ARM.

В сложении двоичных чисел есть всего четыре простых и понятных правила. Вот они:

- ◆ $0 + 0 = 0$ [ноль плюс ноль равно ноль]
- ◆ $1 + 0 = 1$ [один плюс ноль равно один]
- ◆ $0 + 1 = 1$ [ноль плюс один равно один]
- ◆ $1 + 1 = 0 (1)$ [один плюс один равно ноль, один в уме]

Обратите внимание, что в последнем правиле один плюс один равняется нулю, и один в уме.

Эта самая «единица в уме» называется *битом переноса*, и ее наличие сигнализирует о переполнении разряда с переносом единицы в следующий. Тут следует вспомнить, что двоичное число 10 — это десятичное 2 (теперь уже можно понять шутку в начале главы!). Перенос двоичного разряда подобен переносу, который может произойти при сложении двух десятичных чисел, если результат получается больше 9. Например, сложив вместе $9 + 1$, мы получим результат 10 (десять) — т. е. здесь возникает такое же «переполнение» с переходом в следующий разряд: $9 + 1 = 10$. Точно так же при двоичном сложении, когда результат больше 1, мы берем бит переноса и прибавляем его к следующему разряду (разряду двоек).

Давайте попробуем применить эти правила и сложим два 4-битных двоичных числа: 0101 и 0100:

$$\begin{array}{r} 0101 \quad 0x5 \\ + \quad 0100 \quad 0x4 \\ = \quad 1001 \quad 0x9 \end{array}$$

Двигаясь справа налево, мы получаем:

$$\begin{array}{r} 1 + 0 = 1 \\ 0 + 0 = 0 \\ 1 + 1 = 0 (1) \\ 0 + 0 + (1) = 1 \end{array}$$

В этом примере мы получили бит переноса в третьем разряде — он переносится в четвертый разряд, где добавляется к двум нулям. Сложение 8-битных чисел выполняется аналогичным образом:

$$\begin{array}{r} 01010101 \quad 0x55 \\ + \quad 01110010 \quad 0x72 \\ = \quad 11000111 \quad 0xc7 \end{array}$$

Если в восьмом бите, также называемом старшим *битом*, возникает бит переноса, эта единица переносится во второй байт. Однако в ЦП большинства микросхем есть другой вариант обработки такой ситуации — *флаг переполнения*.

Вычитание

Пока что мы работали исключительно с положительными числами, однако при вычитании двоичных чисел нужен способ как-то представлять и отрицательные числа.

В двоичном вычитании используется техника, немного отличающаяся от привычного вычитания. На самом деле процессор вообще не выполняет вычитание, а вместо этого прибавляет отрицательное значение числа, которое нужно вычесть. Например, вместо выполнения операции $4 - 3$ (четыре минус три) мы выполняем: $4 + (-3)$ (четыре плюс минус три).

Взгляните на шкалу, показанную на рис. 4.8, и решите пример: $4 + (-3)$. Начальная точка — ноль. Сначала перейдите к точке 4, обозначенной символом $>$ (сделайте четыре шага в положительном направлении), и прибавьте к ней 3 (сделайте три шага в отрицательном направлении). Теперь мы находимся в точке 1, которая обозначена символом $<<$. Попробуйте использовать этот метод, чтобы вычесть 8 из 12.

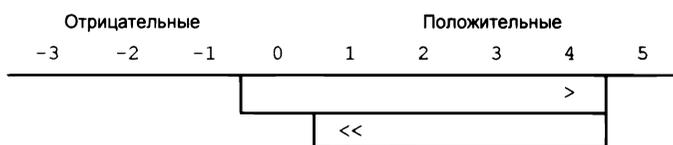


Рис. 4.8. Вычитание чисел

Чтобы сделать то же самое в двоичном формате, нужно сперва придумать способ представления отрицательного числа. Мы используем систему, известную как «двоичный код со знаком». В двоичном коде со знаком бит 7 используется для хранения знака числа. Традиционно 0 в бите 7 обозначает положительное число, а 1 — отрицательное.

На рис. 4.9 показано, как строится двоичное число со знаком. Здесь биты от 0 до 6 содержат само значение — в нашем случае «1». Знаковый разряд здесь указывает на отрицательное значение, поэтому все вместе это дает число -1 .

Бит знака	Значения битов от нулевого до шестого						
1	0	0	0	0	0	0	1

Рис. 4.9. Знаковое двоичное представление числа -1

На рис. 4.10 показано знаковое двоичное представление числа 127. Значение 01111111 в двоичном формате со знаком соответствует числу 127. Биты 0–6 дают собственно число 127, и бит знака очищен.

Бит знака	Значения битов от нулевого до шестого						
0	1	1	1	1	1	1	1

Рис. 4.10. Знаковое двоичное представление 127

Дополнительный код

Просто выделив седьмой разряд под знак, мы не сможем представить отрицательные числа. Сложение $-1 + 1$ должно давать результат 0, но обычное сложение дает результат 2 или -2 в зависимости от того, смотрим ли мы на знаковый разряд. Представление числа в дополнительном коде позволяет кодировать отрицательные числа в обычном двоичном формате так, что сложение будет работать правильно без выделения знаковых разрядов.

Чтобы преобразовать число в его отрицательный аналог, мы должны получить его значение в дополнительном коде. Это делается путем инвертирования каждого бита и прибавления единицы. Чтобы представить число -3 в двоичном формате, сначала запишем двоичное значение числа 3:

```
00000011
```

Теперь инвертируем каждый его бит, заменив 0 на 1 и 1 на 0. Мы получим его *дополнительное значение*:

```
11111100
```

Прибавим 1:

```
11111100
+ 00000001
= 11111101
```

Таким образом, значение -3 в дополнительном коде выглядит так: 1111101. Теперь решим наш знакомый пример: $4 + (-3)$:

```
00000100      4
11111101      -3
= (1) 00000001      1
```

Мы получили число 1, как и следовало ожидать, и, кроме того, возникло переполнение в разряде 7 (значение, показанное в решении в скобках). В настоящее время это переполнение можно проигнорировать, но оно будет нам важно позже — при выполнении вычитания на языке ассемблера — мы это увидим.

Решим еще один пример: $32 - 16$, или $32 + (-16)$:

◆ число 32 в двоичном формате:

```
00100000
```

◆ число 16 в двоичном формате:

```
00010000
```

◆ дополнительный код числа 16:

```
11110000
```

◆ теперь сложим 32 и -16 вместе:

```
00100000  32
+ 11110000 -16
= (1) 00010000  16
```

Если не обращать внимание на перенос, получаем результат 16.

Мы продемонстрировали, что, используя правила двоичного сложения, можно складывать или вычитать числа со знаком. Если при этом не смотреть на переполнение, мы получаем правильный результат, включая знак. Соответственно так же можно сложить два отрицательных числа и получить правильный (отрицательный) результат. Проверим — сложим числа $-2 + -2$:

◆ число 2 в двоичном формате:

```
00000010
```

◆ число 2 в дополнительном коде:

```
11111110
```

◆ возьмем два таких числа и выполним:

```

11111110  -2
+ 11111110  -2
= (1) 11111100  -4

```

Без учета переноса мы получили число -4 . Проверьте правильность ответа, выполнив ту же операцию привычным вам способом.

Понимание дополнительного кода не будет критически необходимо для большинства задач, но может и пригодиться, поскольку вы можете узнать значение каждого бита в числе, будь оно положительное или отрицательное.

Когда двоичные числа не складываются

Существует несколько случаев, когда дополнительный код работает по-особенному, и, что интересно, они связаны с числами 0 и -0 .

Первый — при работе с 0 (нулем). Дополнительный код числа 00000000 выглядит так: 10000000. Отбросив самый старший бит, вы получаете 00000000, что, собственно, и должно быть, ведь числа 0 и -0 , по сути, одинаковы.

Вторая ситуация возникает с числом 10000000, поскольку оно не может иметь отрицательного значения. Инвертировав число 10000000, получаем 01111111, а добавив единицу, чтобы получить его дополнительный код, получим исходное 10000000, с чего вы начали.

Поскольку старший бит в 10000000 равен 1, значение числа отрицательное. Когда вы инвертируете его и добавляете 1, то получаете 10000000, что является двоичным представлением 128, поэтому исходное значение числа должно было быть -128 .

Эта аномалия является причиной асимметричности целочисленных переменных во многих формах BASIC. В 8-битном формате значения лежат в диапазоне от -128 до $+127$, а в 32-битном (четырёхбайтовом) — от -2147483648 до $+2147483647$.

Стандартный калькулятор

В зависимости от настроек, которые вы задали при установке операционной системы Raspberry Pi на плату, вы, возможно, найдете в меню **Accessories** программу-калькулятор. Она запускается на рабочем столе и полезна для выполнения многих числовых задач, которые мы обсуждали здесь, а также других операций, которые мы рассмотрим в ходе чтения книги, если выбрать в ней научный режим. Предлагаю вам самим исследовать ее возможности. Перевод чисел из одной системы в другую — например, в двоичную, шестнадцатеричную и десятичную, там тоже предусмотрен. Есть там даже и восьмеричная система (с основанием 8).

5. Соглашения ARM

У процессоров ARM особый дизайн. Он называется *архитектурой*, поскольку определяет, как устроен и как выглядит процессор с точки зрения пользователя. Понимание этой архитектуры — важный аспект обучения программированию микросхем. Вы должны понимать, как те или иные ее компоненты сочетаются друг с другом и как взаимодействуют, поскольку большая часть машинного кода, который мы будем писать, обращается к различным компонентам процессора ARM и управляет ими.

Благодаря архитектуре с сокращенным набором команд процессор имеет возможность делать много всего, используя небольшой набор команд. Способ его работы определяется режимом. Это означает, что после того, как вы разберетесь с базовыми компонентами чипа ARM, вам нужно будет понять, в каких режимах он может работать. С другой стороны, во время обучения программированию ARM вы почти всегда будет работать в режиме пользователя (User Mode).

Длина слов

В предыдущих главах в примерах с двоичными числами мы использовали однобайтовые значения. И не случайно: на первых популярных компьютерах машинный код работал именно с такими числами. Поэтому и дизайн печатных плат этих компьютеров обеспечивал наличие на них всего восьми линий передачи данных. Каждая линия напрямую была связана с битами в байтах процессора. Таким образом, ЦП мог перемещать данные по плате, переключая логический уровень в каждой линии, устанавливая его значение в 1 или в 0. Он делал это, изменяя напряжение на линии между 5 и 0 В.

Микросхема ARM устроена сложнее и может работать намного быстрее, т. к. перемещает за один раз более крупные единицы информации. Процессоры ARM созданы как в 32-битном, так и в 64-битном вариантах. 32-битный вариант соответствует четырем байтам информации. То есть в нем вместо восьми линий связи используются 32. Вместе эти четыре байта называются *словом*. Таким образом, длина слова в процессоре ARM составляет четыре байта. Однако он может столь же эффективно работать и с однобайтовыми словами. Имейте в виду, что в других компьютер-

ных системах длина слова может быть другой, но на Raspberry Pi длина слова составляет четыре байта, или 32 бита.

Самый старший бит (most significant bit, msb) в слове ARM расположен на позиции 31 (b31), и, если возникает переполнение, из бита 31 поднимается бит переноса. Если перенос произошел из бита 7, он будет перенесен в бит 8, т. е. во второй байт.

Доступ к памяти по байтам и словам

Будучи обладателем компьютера или смартфона, вы знаете, зачем и для чего в компьютере нужна память. Чем больше ее имеется, тем больше данных можно хранить. Каждый адрес в памяти имеет уникальное расположение. Как правило, с увеличением разрядности процессора увеличивается и объем памяти, к которой можно обращаться напрямую. В ранних чипах ARM для адресации памяти использовались только 26 битов из 32. Это накладывало определенные ограничения на процессор и, конечно же, на объем памяти, к которому можно было обращаться напрямую, но в более поздних чипах ARM появилась полная 32-битная адресация. Доступ к наименьшему адресу в этом диапазоне осуществляется путем размещения во всех строках нулей, а к максимальному — путем размещения во всех строках единиц. Первый адрес: 0x0000, а самый высокий: 0xFFFFFFFF (или 0x3FFFFFFF на старых 26-битных шинах ARM до Raspberry Pi).

Устройства управления памятью в процессорах ARM допускают 32-битную адресацию. На рис. 5.1 схематично показано, как устроена память в виде блоков слов длиной в четыре байта. За счет этого диапазон адресов увеличен до 0x00000000–0xFFFFFFFF.

	Бит 31		Бит 00	
(Слово 0)	b03	b02	b01	b00
(Слово 1)	b07	b06	b05	b04
(Слово 2)	b0B	b0A	b09	b08
(Слово 3)	b0F	b0E	b0D	b0C

Рис. 5.1. Блоки слов памяти на ARM

ARM «видит» память в виде таких вот блоков слов, но способен обращаться и к отдельным байтам в каждом слове. С точки зрения работы вся память организована в виде блоков, разделенных по словам. На рис. 5.1 эти блоки называются Слово 0, Слово 1, Слово 2, Слово 3. Заметим, что в Слове 0 содержатся байты b00, b01, b02 и b03, в Слове 1 — байты b04, b05, b06 и b07 и т. д. Это расположение изменить нельзя. То есть мы не можем получить блок из байтов b02, b03, b04 и b05. (Обратите внимание, что буква b здесь означает «байт», а не «бит», как это было в предыдущих примерах.)

Позже в ARM увидели потребность в 64-битных процессорах и начали разрабатывать новые проекты. Началось это задолго до того, как команда анонсировала но-

вую архитектуру ARMv8 — первую архитектуру ARM с 64-битным набором команд. Кроме того, ARM учли ошибки и успехи других разработчиков микросхем, которые перешли на 64-битную архитектуру. Новая 64-битная архитектура ARM полностью совместима с созданной ранее 32-битной архитектурой. Это означает, что если процессор работает в 64-разрядной операционной системе, он может запускать и 32-разрядный код ARMv7 (или двоичные файлы).

Плата Raspberry Pi 2B v1.2 была первой в серии плат с архитектурой ARMv8, и затем такими же стали Raspberry Pi 3 и 4. Эти платы работают в 32-битном режиме, но могут работать и в 64-битном. Мы немного забежали вперед и вернемся к этому моменту позже. Поначалу разница будет незаметна.

Ячейки в памяти адресуются уникальным шестнадцатеричным числом. Адрес памяти, соответствующий началу слова, называется *границей слова* и «выровнен по словам». Адрес памяти выравнивается по словам, если его значение делится на четыре. Следующие адреса выровнены по словам:

```
0x00009030
0x00009034
0x00009038
0x0000903C
```

Выравнивание адресов по словам важно для работы ARM, поскольку именно от этого зависит, как чип ARM выбирает и выполняет машинный код. Например, адрес 0x00009032 не выровнен по словам. Сохранить инструкцию машинного кода ARM на таком адресе нельзя.

В ассемблере GCC есть несколько инструментов, которые помогают правильно управлять границами слов. Поэтому, если вы соберете код с неправильной адресацией, появится сообщение об ошибке. Но мы пока в начале нашего пути, и подобные проблемы возникнут не скоро.

Регистры

У процессора ARM есть несколько внутренних областей памяти, в которых он хранит, отслеживает и обрабатывает информацию. Такая память ускоряет работу и выполнение операций, поскольку для них не требуется доступ к внешней памяти. Эти внутренние области называются *регистрами*. В пользовательском режиме (стандартный режим) доступно 16 регистров, каждый из которых может содержать слово (четыре байта) информации. Каждый из регистров — отдельное слово ARM. На рис. 5.2 показано, как устроена эта память, плюс добавлен дополнительный регистр — регистр состояния (статуса).

Как можно здесь видеть, регистры R0–R12 доступны для использования в любое время. У регистров R13–R15 свое определенное назначение, при этом R13 и R14 используются лишь изредка и управляются всего несколькими командами. Вы, как программист, можете тоже пользоваться этими регистрами для своих целей. А вот регистр R15 лучше не трогать. И не потому, что мы не можем это сделать, а потому,

что нужно четко понимать, что вы с ним делаете и что неверный шаг может добавить вам проблем. Команды ARM могут обращаться к R0–R14 напрямую, и большинство команд могут обращаться к R15.

Банк регистров	
R0	Доступен
R1	Доступен
R2	Доступен
R3	Доступен
R4	Доступен
R5	Доступен
R6	Доступен
R7	Доступен
R8	Доступен
R9	Доступен
R10	Доступен
R11	Доступен
R12	Доступен
R13	Указатель стека
R14	Регистр связи
R15	Программный счетчик
Регистр состояния текущей программы	

Рис. 5.2. Банк регистров ARM в режиме пользователя

Поскольку каждый регистр имеет ширину в одно слово, в регистре может храниться один адрес. Другими словами, регистр может содержать число, которое указывает на любое место на карте памяти Raspberry Pi. Основная функция регистров — хранить такие адреса.

Ассемблер GCC позволяет нам использовать указанные здесь короткие обозначения (такие, например, как R0 и R10) для обращения к ним.

Команды LDR и STR позволяют различными способами загружать регистр из памяти или сохранять его в память. Вот пара примеров команд:

```
LDR R1, [R5]      @ загрузка в R1 данных из адреса R5
STR R1, [R6]      @ сохранение данных из R1 по адресу, указанному в R6
```

В обоих примерах ожидается, что в одном из регистров будет содержаться адрес памяти. Эти регистры заключены в квадратные скобки, давая тем самым ассемблеру понять, что они содержат адреса. Такой тип обозначения называется *режимом адресации*, и у ARM есть несколько режимов адресации. Мы рассмотрим их в следующих главах.

Регистр R15: программный счетчик

Регистр R15, или программный счетчик (Program Counter, PC), — весьма нужный элемент. Если не проявлять должную осторожность в работе с ним, вся ваша программа может «сломаться». Его задача проста — отслеживать, где именно ваша программа выполняет машинный код. По сути, в этом счетчике хранится адрес следующей команды, которую нужно выполнить. Чуть позднее мы рассмотрим этот регистр более подробно — ему посвящена *глава 13*.

Ассемблер GCC позволяет обращаться к программному счетчику как по имени R15, так и через обозначение PC. Например:

```
MOV PC, R0      @ перемещение R0 в R15, программный счетчик
```

Обозначение PC работает так же, как и R15.

Регистр состояния текущей программы

Регистр состояния текущей программы (CPSR, Current Program Status Register) — или просто регистр состояния — хранит важную информацию о текущей программе и результатах операций, которые она выполняет сейчас или уже выполнила. Отдельные биты в этом регистре обозначают некоторые заранее заданные события и позволяют понять, произошли они или нет. Как именно информация попадает в этот регистр? За счет управления значениями отдельных битов в регистре. На рис. 5.3 показано, как это работает.

31	30	29	28	27...8	7	6	5	4	3	2	1	0
N	Z	C	V		I	F	T	Режим				

Рис. 5.3. Конфигурация регистра состояния

Четыре старших бита содержат так называемые *флаги*, которые обозначают наличие или отсутствие некоторого события. Вот эти флаги:

- ◆ N — отрицательный флаг (Negative);
- ◆ Z — нулевой флаг (Zero);
- ◆ C — флаг переноса (Carry);
- ◆ V — флаг переполнения (Overflow).

После выполнения команды процессор ARM, если нужно, обновляет регистр состояния. При возникновении одного из проверяемых условий в нужный флаг помещается значение 1, т. е. флаг поднимается (устанавливается). Если условие не возникло, флаг сбрасывается — в него помещается 0.

Биты и флаги

Если вы читали разделы о двоичной арифметике, то некоторые из приведенных далее соображений будут вам понятны. У нас есть отрицательные числа, и флаг `Negative` используется для обозначения отрицательного числа. Флаг `Carry` — это бит переноса. Мы обсуждали перенос на 8-битных операциях, но и у 32-битных чисел все работает точно так же. Флаг `Zero` поднимается, если результат оказывается равен нулю. Флаг `Overflow` нам не знаком, но с ним тоже все просто — он поднимается, если операция вызвала перенос из бита 30 в верхний бит в бите 31. Если так произошло после операции с числами со знаком, это может указывать на отрицательный результат, даже если отрицательное число на самом деле не получилось (вспомните, что биты нумеруются с нуля, поэтому 32-й бит фактически имеет номер 31, или `b31`).

Например, если результат операции дал 0, будет установлен флаг `Zero`. Это бит `z` на рис. 5.3. Если команда сложения генерирует бит переноса, тогда поднимается флаг `c`. Если перенос не возник, флаг переноса будет снят (`c = 0`).

В языке ассемблера есть мнемоника, которая позволяет проверять флаги регистра состояния и выбирать действия в зависимости от их состояния. Вот пара примеров:

```
BEQ zeroset      @ переход к метке zeroset, если Z = 1
BNE zeroclear   @ переход к метке zeroclear, если Z = 0
```

Здесь `BEQ` (`Branch if Equal`) — это «переход» к указанной метке, который выполняется в случае, если установлен флаг `Zero`. Команда `BNE` (`Branch if Not Equal`) выполняет переход к указанной метке, если флаг `Zero` сброшен.

Аналогичные команды есть и для других флагов. Команда `BNE` часто используется для повтора некоторого участка кода в цикле заданное количество раз, пока счетчик не уменьшится до 0, после чего будет установлен флаг `Zero`.

Биты `i` и `f`, также показанные на рис. 5.3, называются *битами отключения прерывания*, и о них мы поговорим в *главе 29*. Бит `t` касается состояния процессора. На этом этапе мы предположим, что он всегда равен 0 и обозначает некоторое состояние процессора ARM (мы вернемся к этому в *главе 27*). Последние пять битов используются для обозначения режима процессора — мы будем в основном использовать *режим пользователя* (и об этом тоже в *главе 29*).

Интересно, что получить доступ ко всему регистру состояния одной инструкцией нельзя. Вы можете управлять его содержимым только на битовом уровне, выполняя соответствующие действия.

Установка флагов

Существуют две команды, которые позволяют напрямую управлять флагами регистра состояния: `CMP` (`CoMPare`) и `CMN` (`CoMPare Negative`). Первая используется часто и записывается вот так:

```
CMP <Операнд1> <Операнд2>
```

Команда `CMP` выполняет условное вычитание, определяя разницу между *операндом2* и *операндом1*. Физический результат вычитания игнорируется, но флаги регистра состояния обновляются в соответствии с результатом вычитания, который может быть положительным, нулевым или отрицательным (переноса тут быть не может). Если бы результат вычитания был равен 0, был бы поднят флаг `Zero`.

Операнд1 всегда является регистром, а *операнд2* может быть как регистром, так и непосредственно значением. Например:

```
CMP R0, R1      @ сравнение R0 с R1. R0 минус R1
CMP R0, #1      @ сравнение R0 с 1. R0 минус 1
```

Команда `CMP` часто используется в сочетании с инструкцией `BEQ` для создания перехода в другую часть программы:

```
CMP R0, R1
BEQ zeroflagset
```

Этот код передает управление той части программы, которая начинается с метки `zeroflagset`, если сравнение `R0` и `R1` оказывается равным нулю. Если переход не происходит, это будет означать, что результат операции `CMP` не был нулевым и выполнять команду не было бы необходимости. Далее мы напишем код, который все исправит.

`CMP` и `CMN` — единственные команды, которые напрямую позволяют влиять на регистр состояния. По умолчанию остальная часть набора команд ARM не обновляет регистр состояния. Например, если в `R0` и `R1` лежат значения 1 и мы выполним команду:

```
SUB R0, R0, R1
```

в результате получится 0. Но ни один из флагов в регистре состояния не изменится. В регистре сохранится состояние, которое было до выполнения команды.

Суффикс S

Однако в ARM есть метод, позволяющий операциям вроде `SUB` обновлять регистр состояния. Это делается с помощью суффикса `Set` — достаточно добавить символ `s` в конец мнемоники команды, которую мы хотим использовать для изменения флагов:

```
SUBS R0, R0, R1
```

Команда выполнит вычитание содержимого `R1` из `R0`, положит результат в `R0` и обновит флаги в регистре состояния.

Суффикс `s` позволяет вам как программисту использовать на один набор команд меньше. Без него надо было бы написать так:

```
SUB R0, R0, R1
CMP R0, #0
BEQ iszero
```

Но, используя этот суффикс, мы можем удалить строку CMP:

```
SUBS R0, R0, R1
BEQ iszero
```

Ассемблер GCC распознает использование суффикса *s*. Можно также использовать пробелы между инструкцией и буквой *s*, поэтому оба эти примера сработают отлично:

```
SUBS R0, R0, R1
SUB S R0, R0, R1
```

В этом примере показан один из многих способов применения сокращенного набора команд. Суффикс *Set* — лишь один из них, а другие мы рассмотрим в *главе 9*.

R14: регистр ссылок

Рассмотренные нами команды *BEQ* и *BNE* относятся к командам условного перехода. С ними все просто и понятно — они точно выполняют изменение направления кода, т. к. вы задаете ветвь на случай равенства (*BEQ*) и на случай отрицательного результата (*BNE*). Существует второй стиль команд ветвления, известный как *Branch* (или *Branch with Link*, переход со ссылкой) — *BL*. Команда *BL* реализует работу подпрограммы, а именно переходит в другое место в программе, а затем позволяет вернуться к следующей после *BL* команде.

Когда выполняется команда *BL*, адрес возврата (адрес следующей команды) попадает в регистр *R14*, называемый *регистром ссылок* (*LR*). После завершения подпрограммы регистр ссылок копируется в счетчик программы *R15* и программа продолжает работу с того места, откуда прервалась ранее.

Вот один из способов скопировать регистр ссылок в программный счетчик:

```
MOV R15, R14
```

Ассемблер также примет и такой вариант:

```
MOV PC, LR
```

Если вы знакомы с языком *BASIC*, то команды *BEQ* и *BNE* похожи на команду *GOTO*, а *BL* — на *GOSUB*.

R13: указатель стека

Указатель стека (*Stack Pointer*, *SP*) содержит адрес, указывающий на область памяти, которую можно использовать для хранения информации. Эта область памяти называется *стек*, и у нее есть парочка особенностей, которые мы рассмотрим в *главе 17*. Пока стоит лишь отметить, что в *ARM* по умолчанию реализован один стек, но вы можете создать столько стеков, сколько захотите.

6. Обработка данных

В этой главе мы рассмотрим некоторые команды, позволяющие выполнять обработку данных. Это самая большая группа из 18 команд, предназначенных для манипулирования информацией. Их можно разделить на следующие подгруппы:

- ◆ ADD, ADC, SUB, SBC, RSB, RSC;
- ◆ MOV, MVN, CMP, CMN;
- ◆ AND, ORR, EOR;
- ◆ BIC, TST, TEQ;
- ◆ MUL, MLA.

Команды AND, ORR, EOR, BIC, TST и TEQ мы рассмотрим в *главе 8*.

Остальным командам нужно передать данные по следующему шаблону:

<Команда> <Место назначения>, <Операнд1>, <Операнд2>

Давайте рассмотрим каждое поле этого шаблона более подробно.

- ◆ *<Команда>*

Это мнемоника команды языка ассемблера, которую нужно выполнить. Ее можно использовать в исходной форме, которая указана ранее, или с добавлением суффиксов — например, *s*.

- ◆ *<Место назначения>*

Это место, в котором нужно будет сохранить результат. Таким местом всегда является регистр ARM в диапазоне R0–R15.

- ◆ *<Операнд1>*

Это первый элемент информации, с которым вы работаете, и опять же это всегда регистр ARM в диапазоне R0–R15. *Операнд1* может быть таким же, как регистр назначения.

- ◆ *<Операнд2>*

Операнд2 более гибок, чем *операнд1*, поскольку его можно указать тремя разными способами. Это может быть регистр ARM в диапазоне R0–R15. Это может быть

конкретное значение или константа — например, число. В случае с константами в коде указывается само число с хештегом `#`. *Операнд2* также может быть так называемым *смещенным операндом*, и к этому мы вернемся в *главе 11*, когда будем рассматривать арифметический сдвиг чисел.

Далее приведено несколько примеров использования команд по обработке данных:

```
ADD R0, R1, R2      @ R0 = R1 + R2
ADDS R2, R3, #1     @ R2 = R3 + 1 and set flags
MOV R7, #128        @ R7=128
```

Для некоторых команд не обязательно указывать оба операнда. Например, команде `MOV` требуется только *операнд2*. Почему *операнд2*, а не *операнд1*? Потому что это может быть и регистр или постоянное значение (или смещенное, как мы увидим позже).

Команды сложения

В этом разделе мы более подробно рассмотрим команды `ADD` и `SUB`, а также подробнее поговорим о том, что происходит в регистрах, включая регистр состояния и его флаги.

Сложение выполняется двумя командами: `ADD` и `ADC`. Вторая — это та же `ADD`, но с добавлением `Carry`. Синтаксис одинаковый:

```
ADD (<Суффикс>) <Место назначения>, <Операнд1>, <Операнд2>
ADC (<Суффикс>) <Место назначения>, <Операнд1>, <Операнд2>
```

Далее приведен код, в котором используется команда `ADDS`. Эта программа очищает регистр `R0`, помещает `1` в регистр `R1` и устанавливает все 32 бита `R2`. Это самое большое число, которое мы можем сохранить в четырехбайтовом регистре. Что будет, если мы запустим эту программу?

```
MOV R0, #0
MOV R1, #1
MOV R2, #0xFFFFFFFF
ADDS R0, R1, R2
```

Регистры в результате получатся такими:

```
R1: 0x00000001
R2: 0xFFFFFFFF
R0: 0x00000000
```

На вид ничего не изменилось! Все значения попали в нужные регистры, но сложения, похоже, не произошло, поскольку в `R0` все еще лежит `0`. На самом деле да, но, добавив `1`, мы создали бит переноса (помните сложение двоичных чисел, которое мы выполняли в предыдущих главах?). Регистр состояния в результате получился вот таким:

```
NZCV
0110
```

Если бы мы выполнили эту программу, используя команду `ADD`, а не `ADDS`, то флаг переноса не обновился бы и показывал бы результат ранее выполненной соответствующей команды. Возможно, флаг переноса был установлен предыдущей инструкцией, поэтому результат мог бы получиться таким же, но это случайность. А уповать на случайность в программировании на любом языке нельзя. Семь раз отмерь, как говорится.

В программе 6.1 показано, как сложить два числа в машинном коде. Введите этот код в Vim или Geany, а файл назовите `prog6a.s`.

ПРОГРАММА 6.1. Простое 32-битное сложение

```
/* Вычисление R0 = R1 + R2          */
.global _start
_start:
    MOV R1, #50          @ помещение 50 в R1
    MOV R2, #60          @ помещение 60 в R2
    ADDS R0, R1, R2      @ сложение и помещение результата в R0

    MOV R7, #1           @ выход через системный вызов
    SWI 0
```

Можно собрать, связать и запустить этот код следующим образом:

```
as -o prog6a.o prog6a.s
ld -o prog6a prog6a.o
./prog6a
```

Теперь распечатайте результат:

```
echo $?
```

Получится 110.

ПРИМЕЧАНИЕ

Не забудьте: чтобы иметь возможность распечатать результат машинного кода с помощью `bash`, нужно убедиться, что результат хранится в `R0` и используется выход из операционной системы.

Если мы складываем два значения и не уверены на 100%, что переноса не возникло, или если факт его наличия нам не важен, стоит проверить флаг.

Приведенная далее программа 6.2 складывает два 64-битных числа. Такой размер — это уже два слова, поэтому для хранения числа необходимы два регистра, один из которых будет содержать младшие четыре байта, а другой — старшие четыре байта. Поскольку теперь возможен перенос из нижнего слова в верхнее слово, при сложении обязательно надо будет проверить флаг переноса. Для этого следует использовать инструкцию `ADC`.

Код предполагает, что первое число находится в регистрах `R2` и `R3`, а второе — в `R4` и `R5`. Результат помещается в `R0` и `R1`. По соглашению нижний регистр всегда содержит нижнюю половину числа.

ПРОГРАММА 6.2. 64-битное сложение

```

/* Сложение двух 64-битных чисел          */
.global _start
_start:
    MOV R2, #0xFFFFFFFF    @ младшая часть числа 1
    MOV R3, #0x1           @ старшая часть числа 1
    MOV R4, #0xFFFFFFFF    @ младшая часть числа 2
    MOV R5, #0xFF          @ старшая часть числа 2
    ADDS R0, R2, R4        @ сложение младших частей и установка флага
    ADCS R1, R3, R5       @ сложение старших частей с переносом

    MOV R7, #1            @ выход через системный вызов
    SWI 0

```

Вы можете собрать, связать и запустить этот код следующим образом:

```

as -o prog6b.o prog6b.s
ld -o prog6b prog6b.o
./prog6b

```

Теперь распечатайте результат:

```
echo $?
```

В результате получим **254**. Почему?

После выполнения программы в регистрах R0 и R1 окажутся значения 0xFFFFFFFF и 0x101 соответственно. Результат получается таким:

```
0x101FFFFFFFFE
```

В первой команде ADDS добавление вызывало установку флага переноса, который подхватывается командой ADCS. Если мы заменим ADCS другой командой — ADDS, результат будет другим:

```
0x100FFFFFFFFE
```

В десятичном выражении результат отличается аж на 4294967296!

Вы можете спросить, почему команда ADCS не использовалась в обеих частях сложения? Как правило, при сложении нужно быть уверенным, что флаг переноса снят перед началом сложения. В противном случае при использовании ADCS вы подхватите лишний флаг переноса и получите ошибочный результат. Использование ADDS гарантирует, что перенос не будет использоваться, но обновится после сложения.

Вернемся к вопросу о том, почему команда echo вернула **254**, т. е. 0xFE в шестнадцатеричном формате — младший значащий байт значения, хранящегося в R0. Дело в том, что команда echo возвращает не то, что хранится в R0, а то, что находится в младшем байте R0.

Как переписать код, чтобы можно было сложить два значения из трех слов? Может возникнуть соблазн повторить последовательность ADDS и ADCS. Но это неправильно.

Нужно и дальше использовать инструкцию ADCS, пока вы не сложите все слова. Команда ADDS задействуется только в первом слове для определения условия переноса в первом экземпляре. А дальше только ADCS.

Если вы пишете программу, которая считает все неправильно, и значения, которые она возвращает, сильно различаются, всегда стоит проверить, используется ли правильная последовательность операций сложения. Скорее всего, именно здесь и кроется ошибка.

Если бы нужные нам трехсловные числа лежали в регистрах R4, R5, R6 и R7, R8, R9, то сложить их и положить результат в R1, R2, R3 можно было бы следующим образом:

ADDS R1, R4, R7	@ сложение младших слов и проверка переноса
ADCS R2, R5, R8	@ сложение средних слов с переносом
ADCS R3, R6, R9	@ сложение старших слов с переносом

Разумеется, в конце вам нужно проверить флаг переноса, поскольку он станет самым значимым битом в конечном результате.

Вычитание

Сложение выполняют две команды, а вот вычитание — четыре:

SUB (<Суффикс>) <Место назначения>, <Операнд1>, <Операнд2>
SBC (<Суффикс>) <Место назначения>, <Операнд1>, <Операнд2>
RSB (<Суффикс>) <Место назначения>, <Операнд1>, <Операнд2>
RSC (<Суффикс>) <Место назначения>, <Операнд1>, <Операнд2>

Здесь варианты следующие: простое вычитание, которое игнорирует флаги, и еще одно, которое учитывает флаг переноса (SBC). Второй набор команд вычитания работает аналогичным образом, но операнды используются в обратном порядке. Например, команда:

```
SUB R0, R1, R2
```

вычитает содержимое R2 из R1 и помещает результат в R0. А вот эта:

```
RSB R0, R1, R2
```

вычитает содержимое R1 из R2 и помещает результат в R0. Как и в предыдущих примерах, можно использовать суффикс S:

```
SUBS R0, R1, R2
```

Если R0 = 0, R1 = 0xFF и R2 = 0xFE, команда SUBS выполнит вот такое действие:

```
0xFF - 0xFE
```

или

```
255-254
```

В результате получим 1, и это правильно. Но если мы посмотрим на регистр состояния, то увидим флаг переноса. Почему?

Возьмем теперь не SUBS, а RSB:

```
RSBS R0, R1, R2
```

Загрузив те же значения в регистры, в регистре R0 получим 0xFFFFFFFF, а флаг переноса окажется сброшен! При вычитании флаг переноса используется «неправильно», поэтому, если требуется заимствование, флаг не установлен или сброшен. То есть он работает уже не для переноса! Это полезно при работе с числами, превышающими 32 бита, и результат получается правильным. В последнем случае также будет установлен отрицательный флаг, поскольку число 0xFFFFFFFF в числах со знаком окажется отрицательным. Приведенный пример прекрасно иллюстрирует это благодаря использованию чисел в дополнительном коде.

Если написанный код не дает ожидаемого результата, всегда имеет смысл проверить состояние флагов регистра состояния. Они могут вести себя не так, как вы ожидаете.

Здесь следует помнить два правила:

- ◆ если требуется заимствование, флаг переноса сбрасывается, C = 0;
- ◆ если заимствование не генерируется, флаг переноса поднимается, C = 1.

Если мы выполняем вычитание сразу нескольких слов, заимствование из одного слова означает, что нам нужно вычесть лишнее из следующего слова. Но мы уже знаем, что заимствование приводит к тому, что флаг переноса сбрасывается, а нам это не подходит. Именно поэтому процессор ARM инвертирует флаг переноса перед его использованием в операции SVC. Используя эту особенность, можно выполнять вычитание операндов, для представления которых требуется любое количество слов, — просто повторите команду SVC столько раз, сколько требуется.

Вы можете спросить, зачем вообще нужны команды обратного вычитания? Опять же, причины заключаются в общей идее скорости. Имея возможность указать, какой из операндов вычитается, мы устраняем необходимость лишних обменов данных.

Умножение

У ARM есть пара команд, выполняющих 32-битное умножение. Первая из них — MUL — выполняет прямое умножение и использует следующий синтаксис:

```
MUL (<Суффикс>) <Место назначения>, <Операнд1>, <Операнд2>
```

Команда MUL немного отличается от команд вроде ADD и SUB, т. к. у нее есть определенные ограничения на возможные операнды. Правила здесь следующие:

- ◆ *Место назначения* — это должен быть регистр, причем не такой, как *операнд1*. R15 тоже нельзя использовать в качестве места назначения;
- ◆ *Операнд1* — должен быть регистром и не может быть регистром назначения;
- ◆ *Операнд2* — должен быть регистром и не может быть константой или сдвиговой операцией.

Таким образом, с командой `MUL` можно использовать только регистры, нельзя использовать `R15` в качестве места назначения, а регистр назначения не может использоваться в качестве операнда. Пример:

```
MUL S R0, R4, R5    @ R0 = R4 * R5 and set status
```

В программе 6.3 показано, как работает команда `MUL`: два числа мы помещаем в `R1` и `R2`, а результат умножения — в `R0`.

ПРОГРАММА 6.3. 32-битное умножение

```
/* умножение двух чисел R0 = R1 * R2 */
.global _start
_start:
    MOV R1, #20    @ R1 = 20
    MOV R2, #5     @ R2 = 5
    MUL R0, R1, R2 @ R0 = R1 * R2

    MOV R7, #1     @ выход через системный вызов
    SWI 0
```

Чтобы собрать, связать и запустить код, выполните следующие команды:

```
as -o prog6c.o prog6c.s
ld -o prog6c prog6c.o
./prog6c
```

Теперь распечатайте результат:

```
echo $?
```

Вторая команда — `MLA` — это умножение с накоплением. Оно отличается от `MUL` тем, что позволяет складывать результаты умножения в одну «корзинку». То бишь, накапливать значения. Синтаксис команды:

```
MLA (<Суффикс>) <Место назначения>, <Op1>, <Op2>, <Сумма>
```

Правила, изложенные в начале этого раздела, работают и здесь. Появился также дополнительный операнд: *Сумма*, который должен быть регистром. Например:

```
MLA R0, R1, R2, R3    @ R0 = (R1 * R2) + R3
```

Регистр, указанный в *сумме*, может быть таким же, как регистр места назначения, и в этом случае результат умножения будет накапливаться в регистре назначения:

```
MLA R0, R1, R2, R0    @ R0 = (R1 * R2) + R0
```

Давайте доработаем программу 6.3 и получим программу 6.4.

ПРОГРАММА 6.4. Использование умножения с накоплением `MLA`

```
/* Умножение двух чисел с накоплением R0 = (R1 * R2) + R3 */
.global _start
```

```

_start:
    MOV R1, #20          @ R1 = 20
    MOV R2, #5           @ R2 = 5
    MOV R3, #10          @ R3 = 10
    MLA R0, R1, R2, R3  @ R0 = (R1 * R2) + R3

    MOV R7, #1           @ выход через системный вызов
    SWI 0

```

Чтобы собрать, связать и запустить код, выполните следующие команды:

```

as -o prog6d.o prog6d.s
ld -o prog6d prog6d.o
./prog6d

```

Теперь распечатайте результат:

```
echo $?
```

Возвращаемый результат всегда будет на 10 больше, чем произведение двух значений в R1 и R2. И все потому, что в R3 лежит 10.

Обратите внимание, что регистр *места назначения* не может использоваться как *Op1* или *Op2*. Так что этот код:

```
MLA R0, R0, R2, R3      @ R0 = (R0 * R2) + R3
```

выдает сообщение об ошибке.

Теперь о делении

У процессора ARM, используемого в Raspberry Pi 1 и Raspberry Pi Zero, не было команды деления, поэтому деление двух чисел либо требовало некоторой изобретательности, либо выполнялось с использованием методологии подсчета/вычитания.

У Raspberry Pi 2 и в более поздних версиях команды деления SDIV и UDIV стали доступны в вариантах со знаком и без знака соответственно. Команды, которые не влияют на флаги регистров состояния, работают непосредственно с регистрами и записываются так:

```

SDIV <Место назначения>, <Числитель>, <Знаменатель>
UDIV <Место назначения>, <Числитель>, <Знаменатель>

```

В регистр назначения по завершении деления попадет частное. Остаток необходимо рассчитать отдельно. Если *место назначения* опущено, результат запишется в *числитель*. В программе 6.5 приведен пример использования команды SDIV.

ПРОГРАММА 6.5. Деление командой SDIV

```

/* Деление командой SDIV со знаком в модели RPi 2 и более поздней */
.global _start

```

```

_start:
    MOV R3, #20      @ числитель
    MOV R4, #5       @ знаменатель
    SDIV R0, R3, R4  @ R0 = R3/R4
                    @ не использовать SP или PC,
                    @ флаги регистра состояния не меняются
                    @ деление на 0 возвращает 0
    MOV R7, #1       @ выход через системный вызов
    SWI 0

```

Результат можно распечатать, набрав:

```
echo $?
```

В результате получим 4.

В *главе 12* показано, как можно выполнить деление без этих команд. Там же мы увидим, как вычислить остаток, поскольку это тоже часто бывает нужно.

В Raspberry Pi 2 добавили новый SoC, в котором использовался процессор BCM2836 и связанный с ним FPU с реализованной инфраструктурой подразделения. В более поздних версиях это сохранилось.

Команды перемещения

Перемещением данных занимаются две команды: `MOV` и `MVN`. Они используются для загрузки данных в регистр из другого регистра или для загрузки определенного значения в регистр. *Операнд1* здесь не используется, и синтаксис будет таким:

```

MOV (<Суффикс>) <Место назначения>, <Операнд2>
MVN (<Суффикс>) <Место назначения>, <Операнд2>

```

А теперь пара примеров:

```

MOV R0, R1          @ копирование содержимого R1 в R0
MOV R5, #0xFF       @ помещение 255 в R5

```

Обратите внимание на комментарий в этом примере: хотя команда и называется `MOV`, важно понимать, что содержимое исходного регистра не изменилось. Данные лишь копируются. Кроме того, если не используется суффикс `s`, регистр состояния также не изменяется. А вот такая команда:

```
MOVS R0, #0
```

поместит ноль в `R0` и одновременно поднимет флаг нуля.

Команда `MVN` — это «отрицательное перемещение». Перемещаемое значение в процессе инвертируется. Это означает, что единицы превращаются в нули, а нули — в единицы:

```

MVN R0, #9          @ перемещение -9 в R0
9:  0x00000009      00000000 00000000 00000000 00001001
MNV  0xFFFFFFFF6   11111111 11111111 11111111 11110110

```

Вот пара примеров:

```
MVN R0, #0      @ помещаем в R0 значение -1
MVN R1, #1      @ помещаем в R1 значение -2
```

В *главе 11* вы узнаете, что в регистры можно загружать не любые значения. Проще говоря, есть некоторые числа, которые вы просто не можете использовать напрямую, и это не потому, что они, например, слишком большие. Та же проблема возникает при работе с адресами в памяти. В свое время мы рассмотрим эти проблемы подробнее.

Команды сравнения

С этими командами мы уже работали, когда рассматривали регистр состояния и флаги в предыдущей главе. Команды сравнения имеют следующий синтаксис:

```
CMP <Операнд1>, <Операнд2>      @ подъем флага <Оп1> - <Оп2>
CMN <Операнд1>, <Операнд2>      @ подъем флага <Оп1> + <Оп2>
```

Эти команды не перемещают информацию и не изменяют содержимое каких-либо регистров. Вместо этого они обновляют флаги регистра состояния. Поскольку команды `CMP` и `CMN` непосредственное воздействуют на флаги регистра состояния, нет причин использовать суффикс `S`. Команда `CMP` вычитает *операнд2* из *операнда1*, но не сохраняет результат:

```
CMP R3, #0
```

В приведенном примере флаг `Zero` будет установлен только в том случае, если сам `R3` содержит `0`, в противном случае флаг `Zero` будет снят.

```
CMN R3, #128
```

Здесь будет поднят флаг нуля, если в `R3` имеется значение `128`. Если в `R3` окажется значение меньше `128`, будет установлен отрицательный флаг. А как может появиться флаг переполнения? Вообще, когда вы сомневаетесь в результате, просто проверьте его от руки с помощью правил двоичной алгебры!

Команда `CMN` — отрицательная версия сравнения. Эта команда полезна, если вы хотите контролировать уменьшение цикла после нуля. Тогда вам подойдет вот такой вариант:

```
CMN R0, #1      @ сравнение R0 с -1
```

Тут идея та же, что и у команды `MVN`. Это позволяет проводить сравнения с небольшими отрицательными константами, которые невозможно представить иначе.

Важно понимать, что в команде `MVN` используется логическое отрицание от *операнда2*. А в `CMN` используется отрицательное значение операнда. Таким образом, чтобы сравнить `R0` с `-3`, мы бы написали:

```
CMN R0, #3
```

ARM автоматически получит отрицательное значение *операнда2* и затем выполнит сравнение.

Как и в случае с `CMR`, целью `CMN` является воздействие на флаги регистра состояния, а суффикс `S` не применяется.

Сортировка чисел

Порядок хранения данных в памяти весьма важен. Рассмотрим шестнадцатеричное число:

```
0xFF00AA99
```

Если это число хранится в четырех последовательных байтах памяти, как они упорядочены? Так?

```
0xFF, 0x00, 0xAA, 0x99
```

Или так?

```
0x99, 0xAA, 0x00, 0xFF
```

Эти два метода называются методами с *прямым* и *обратным* порядком байтов. Как видите, байты хранятся в разном порядке в зависимости от используемого метода.

Микросхемы ARM работают *в обе стороны* и поддерживают любой метод. Реально используемый метод определяется конкретным битом в `CPSR`, который определяет, какой «порядок следования байтов» задействовать. По умолчанию в ОС Raspberry Pi используется методология с прямым порядком байтов. По большей части порядок байтов будет прозрачным, поскольку ассемблер в этом случае думает за нас.

7. Входы и выходы

Работая с компьютерами и операционными системами вроде Raspberry Pi OS, мы очень многое воспринимаем как должное. Мы набираем команды на клавиатуре, они «запускаются» и чаще всего выдают вывод в виде информации, или что-то происходит на экране. За этим кроется многое.

Рассмотрим пару, казалось бы, простых задач: ввод команды с клавиатуры и вывод ответа на экране. Именно это мы и делаем во время работы с командной строкой Raspberry Pi OS. Тогда возникает вопрос: как мы получаем данные с клавиатуры и выводим что-то на экран в программах, которые пишем?

Строго говоря, мы сами это и делаем. Но для этого требуется много знаний о различных аппаратных компонентах Raspberry Pi, потому что, например, для вывода сообщения на экран, нужно точно знать, что за устройство управляет экраном, где оно находится в памяти компьютера и как записать в него информацию. Аналогично, чтобы считать ввод с клавиатуры, нам нужно понимать, как компьютер воспринимает клавиатуру и какая у нее матрица кодирования (иначе нельзя будет понять, какие клавиши нажаты).

Операции ввода/вывода с аппаратными средствами часто называют программированием на «голое железо», потому что вы «общаетесь» напрямую с аппаратным обеспечением компьютера. Это очень интересная, но достаточно сложная тема, которую не стоит рассматривать в книге для начинающих. И если вы не собираетесь просто поупражняться в программировании на «голом железе», уделять внимание этому сейчас не стоит. Вместо этого мы поговорим о работе со встроенными процедурами операционной системы.

Команды SWI и SVC

Команда SWI позволяет получить доступ к встроенным процедурам, или библиотекам функций операционной системы. Аббревиатура SWI расшифровывается как SoftWare Interrupt (программное прерывание), и неспроста, потому что при обнаружении этого прерывания выполнение вашей программы приостанавливается и передается соответствующей процедуре. После выполнения команды SWI управление возвращается вызывающей программе, и она продолжает работу. Команду SWI так-

же часто называют `svc` (`SuperVisor Call`, вызов супервизора) поскольку именно этот режим работы активируется в чипе ARM.

Вы, возможно, помните, что мы уже использовали команду `SWI` во всех наших программах на ассемблере. А обращались мы к ней для выхода из кода обратно в командную строку. Вот как это было:

```
MOV R7, #1
SWI 0
```

Все вызовы команды `SWI` выполняются с аргументом `SWI 0` (можно также использовать `svc 0`). Это называется вызовом операционной системы, или сокращенно системным вызовом. Выполняемая командой `SWI` функция определяется числом, находящимся в регистре `R7`. Поскольку другие регистры могут содержать постороннюю информацию, для вызова `SWI 0` часто требуются некоторые подготовительные действия. Например, для вывода строки символов на экран нужно положить три других элемента информации в определенные регистры.

Чтобы эффективно пользоваться командой `SWI`, нам нужно понимать, что делают вызываемые ею процедуры, какую информацию необходимо передать и в какие регистры ее положить. Иногда после вызова команды `SWI` информация передается обратно в программу, и в таких случаях нам нужно знать, какая это информация и в каких регистрах она окажется.

В *приложении 3* приведен список системных вызовов, доступных в ОС Raspberry Pi. Подробное описание всех вызовов `SWI` здесь не приводится, но наиболее распространенные и полезные из них описаны в различных местах этой книги. Официального списка системных вызовов не существует, но есть различные источники на независимых сайтах, где вы сможете почитать о них.

Давайте рассмотрим два, вероятно, наиболее важных сейчас для нас системных вызова: вывод данных на экран и чтение с клавиатуры. Мы будем часто использовать их в примерах программ на протяжении всей книги, поэтому имеет смысл прямо сейчас с ними познакомиться. В процессе изучения материала книги мы рассмотрим еще несколько функций ассемблера GCC и попробуем в деле пару методов языка ассемблера, о которых мы узнаем подробнее позже.

Вывод на экран

Чтобы вывести последовательность или строку символов ASCII на экран, нужна функция записи. Это процедура `Syscall 4`. Для процедуры `Syscall 4` требуются следующие параметры:

- ◆ `R0` — определяет выходной поток, значение 1 соответствует монитору;
- ◆ `R1` — адрес строки символов;
- ◆ `R2` — количество выводимых символов;
- ◆ `R7` — номер процедуры `Syscall`, поэтому `R7 = 4`.

ПРИМЕЧАНИЕ

ASCII (American Standard Code for Information Interchange) расшифровывается как Американский стандартный код для обмена информацией, а ASCII-код — это простое число, используемое для представления символа. В *приложении 1* приведена универсальная таблица символов ASCII, принятая в качестве стандарта.

Ассемблер GCC позволяет хранить строку символов ASCII в теле файла машинного кода. В *программе 7.1* показано, как это сделать. Ключевым моментом здесь является указание директивы ассемблера GCC `.ascii` в последней строке. Эта директива сообщает ассемблеру, что за строкой, заключенной в кавычки, следует строка символов ASCII. А еще следует обратить внимание на пару символов `\n` в конце символьной строки внутри кавычек. Символ обратной косой черты (обратный слеш) означает, что следующий за ним символ является «управляющим символом» и делает что-то особенное. Соответственно набор `\n` означает переход на новую строку. Для обозначения начала местоположения используется метка `string`.

ПРОГРАММА 7.1. Системный вызов 4 для вывода строки на экран

```
/* Использование Syscall 4 для записи строки */
.global _start
_start:

    MOV R7, #4           @ номер системного вызова
    MOV R0, #1           @ Stdout - это монитор
    MOV R2, #19          @ строка состоит из 19 символов
    LDR R1,=string       @ строка находится на метке string:
    SWI 0

_exit:
                                @ выход из системного вызова

    MOV R7, #1
    SWI 0

.data
string:
.ascii "Hello World String\n"
```

Создайте, соберите, свяжите и опробуйте эту программу. Заметим, что команду

```
LDR R1,=string
```

можно читать как

```
Load Register R1 with the address of the label string:
```

Когда выполняется вызов `Syscall 4`, он идентифицирует выходной поток (значение 1 в регистре `R0`) и определяет стандартное устройство вывода — монитор. Затем он извлекает длину строки из `R2` и выводит это количество символов, начиная с адреса, хранящегося в `R1`. Указанное в `R2` количество символов включает и пробелы.

лы, и любые знаки препинания. Последняя пара символов `\n` рассматривается как один символ. Попробуйте изменить значение, загруженное в `R2`, и подумайте, что получится. Например, возьмите вот такое значение:

```
MOV R2, #11
```

Вы также заметите, что в программе есть дополнительная директива:

```
.data
```

Она сообщает ассемблеру, что код после нее следует рассматривать как раздел с данными, а не код на языке ассемблера.

Подраздел данных можно размещать и в начале кода. Но тогда нам пришлось бы обозначить начало кода ассемблера с помощью такой директивы:

```
.text
```

Ваш стиль структурирования файлов полностью зависит от того, каким образом вы собираетесь работать. Я предпочитаю размещать данные и области данных в конце программы, чтобы не было проблем с выравниванием. В *главе 5* мы отмечали, что машинный код ARM собирается на границах четырехбайтовых слов, т. е. начинается с адреса, который делится на четыре. А это может быть не так, если, например, использовалась строка из 10 символов. Исправить такое несоответствие можно с помощью директивы `align`, о которой мы поговорим позже.

Чтение с клавиатуры

Чтобы прочитать последовательность (или строку) символов ASCII с клавиатуры, нам нужно использовать функцию чтения. Это процедура `Syscall 3`. Параметры, необходимые для ее вызова, аналогичны процедуре `Syscall 4`:

- ◆ `R0` — определяет входной поток, значение 0 соответствует клавиатуре;
- ◆ `R1` — адрес буфера для размещения введенных символов;
- ◆ `R2` — количество вводимых символов;
- ◆ `R7` — номер процедуры `Syscall`, поэтому `R7 = 3`.

За основу для новой программы можно взять *программу 7.1*. Вы можете сделать ее копию в командной строке, используя команду `cp`:

```
cp prog7a.s prog7b.s
```

Теперь отредактируйте исходный файл, чтобы он содержал новую процедуру `_read` (*программа 7.2*).

ПРОГРАММА 7.2. Системный вызов 3 и чтение с клавиатуры

```
/* Использование Syscall 3 для чтения с клавиатуры */
.global _start
_start:
```

```

_read:
                                @ системный вызов чтения
MOV R7, #3                      @ номер системного вызова
MOV R0, #0                      @ Stdin - это клавиатура
MOV R2, #5                      @ считывание первых 5 символов
LDR R1,=string                 @ помещение строки по метке string:
SWI 0

```

```

_write:
                                @ системный вызов записи
MOV R7, #4                      @ Номер системного вызова
MOV R0, #1                      @ Stdout - это монитор
MOV R2, #19                    @ строка состоит из 19 символов
LDR R1,=string                 @ строка находится по метке string:
SWI 0

```

```

_exit:
                                @ выход из системного вызова
MOV R7, #1
SWI 0

```

```

.data
string:
.ascii "Hello World String\n"

```

Осталось еще определить строку ASCII. Я намеренно оставил исходный текст программы, чтобы вы увидели, что получится в результате использования функции. Метка `string` фактически является буфером, или местом для размещения данных, считываемых с клавиатуры. Мы могли бы просто определить пустую строку, например, так:

```
.ascii " " "
```

Есть и другие способы зарезервировать пустое место для хранения данных в программе, и их мы рассмотрим позже.

Регистр `R2` теперь используется для хранения количества символов, которые мы хотим считать с клавиатуры. Но это не в точности то количество символов, которое можно набрать. При выполнении команды `_read`: программа принимает весь ввод с клавиатуры, пока не будет нажата клавиша `<Return>`. Затем из них будет взято столько символов, сколько определено значением в `R2`. Таким образом, если вы на берете:

```
123456789
```

выведется только 12345 (первые пять символов). Все остальное станет обрабатываться так, как если бы была введена команда `bash`, и будет выдано сообщение об

ошибке (`bash` — это имя командной строки Raspberry Pi OS, с которой вы работаете). Процедура `_write`: выводит полученную строку, и получится вот это:

```
12345 World String
```

То есть `12345` заменит `Hello`.

Снова запустите программу и введите

```
12
```

Теперь выведется вот такая строка:

```
12
lo World String
```

Обратите внимание, что здесь появилась новая строка. Это потому, что символ `<Return>` тоже вставляется в строковый буфер. Мы вернемся к системным вызовам в *главе 18*.

Регистр `eax` и прочие

Большая часть документации по системным вызовам, с которой вы сталкиваетесь, написана для компьютеров без поддержки ARM и, в частности, для систем *i386*. Поэтому нам часто приходится работать с чужим набором ссылок на регистры. На рис. 7.1 показаны эти регистры и их эквиваленты в ARM, которые должны помочь вам разобраться, что и как.

<i>i386</i>	ARM	Функция
<code>eax</code>	R7	Номер системного вызова
<code>ebx</code>	R0	Аргумент 1
<code>ecx</code>	R1	Аргумент 2
<code>edx</code>	R2	Аргумент 3
<code>esi</code>	R3	Аргумент 4
<code>edi</code>	R4	Аргумент 5
<code>eax (on Return)</code>	R0	Возвращаемое значение или код ошибки

Рис. 7.1. Регистры системных вызовов в *i386* и ARM

Программа Make

Создавая новые исходные файлы, мы каждый раз выполняли сборку и привязку в командной строке. Это повторяющийся процесс, но его можно облегчить с помощью истории консоли. Используя клавиши со стрелками вверх и вниз в консоли, вы можете прокручивать ранее введенные команды и редактировать их.

А еще в GNU есть очень умная программа под названием Make. Это инструмент, который позволяет программистам создавать исполняемые файлы из одного управ-

ляющего файла. Если на вашем компьютере установлена эта программа, есть вероятность, что весь процесс ее работы будет управляться файлом программы Make. Make — очень сложный инструмент, и почитать о нем подробнее можно на веб-сайте GNU.

Программа 7.3 — это ее исходный файл (хоть и сохраненный без суффикса `.s`), который автоматизирует процесс сборки и компоновки. Он чрезвычайно гибкий и позволяет пользоваться многими возможностями. (Обратите внимание, что символы `#` в программе `make` и `@` в файлах ассемблера эквивалентны и определяют комментарии.)

ПРОГРАММА 7.3. Автоматизация сборки и привязки с помощью Make

```
PROGRAMS = prog7a prog7b

# Если мы передали цель в командной строке,
# она попадает в список программ, о которых мы знаем.
ifneq ($(MAKECMDGOALS),)
    ifneq ($(MAKECMDGOALS),clean)
        PROGRAMS = $(MAKECMDGOALS)
    endif
endif

# Правило по умолчанию, если ничего не указано в командной строке
all: $(PROGRAMS)

# Make знает, как компилировать файлы .s, поэтому нам
# достаточно связать их.
$(PROGRAMS): % : %.o
    ld -o $@ $<

clean:
    rm -f *.o $(PROGRAMS)
```

Создайте приведенный здесь файл и назовите его `makefile`. Теперь нет необходимости добавлять `.s` к имени файла — достаточно просто вызвать `makefile`. Убедитесь, что этот файл находится в том же каталоге, что и исходный файл. Также обратите внимание, что две строки кода:

```
ld -o $@ $<
```

и

```
rm -f *.o $(PROGRAMS)
```

для правильной работы должны иметь *отступ в один символ табуляции*.

Запустите `make`-файл, набрав в командной строке:

```
make
```

Переменная `PROGRAMS` (первая строка в `make`-файле) используется для хранения имен исходных файлов, которые нужно собрать и связать. Здесь вы можете ввести

одно или несколько имен — сколько захотите — через пробел. По сути, это все. Остальная часть программы соберет каждый из файлов, создаст объектные файлы и затем свяжет их.

В приведенном в *программе 7.3* листинге это исходные файлы с именами: `prog7a` и `prog7b`. Обратите внимание, что суффикс `.s` подразумевается, и указывать его явно не надо:

```
PROGRAMS = prog7a prog7b
```

Также подразумевается, что `make`-файл лежит в том же каталоге, что и файлы исходного кода. Если исходные файлы были ранее собраны и связаны, они будут перезаписаны при условии, что целевые файлы старше, чем связанные с ними исходные файлы. Если файлы не существуют, `make` выдаст сообщение об ошибке. Вы можете «заставить» программу перезаписать файлы командой:

```
make -B
```

Если вы хотите собрать и связать определенный файл или файлы, то можете ввести имя файла после команды следующим образом:

```
make prog7a
```

Эта команда соберет и свяжет исходный файл под названием `prog7a`, если он существует. Такой способ лучше, чем записывать имена в начале `make`-файла.

В командной строке наберите:

```
make clean
```

Эта команда удаляет файлы с расширением `.o` из каталога.

Имеет смысл завести `make`-файл в каждом каталоге, в котором вы создаете и сохраняете исходные файлы, которые необходимо собирать и связывать.

Утилита `Make` очень универсальна и может использоваться несколькими способами. Если вы хотите изучить ее более подробно, на сайте GNU по адресу: www.gnu.org/software/make/, вы найдете множество руководств и примеров.

Некоторые дополнительные примеры `make`-файлов представлены в следующих главах книги, а исходные файлы, которые могут быть загружены с моего сайта, сопровождаются необходимыми `make`-файлами, когда это имеет смысл. Если вы загрузили программные файлы с моего веб-сайта, то обнаружите, что `make`-файл (или аналогичный) имеется для каждой из программ, если это необходимо.

ПРИМЕЧАНИЕ К РУССКОМУ ИЗДАНИЮ

Напомним, что электронный архив с файлами приведенных в книге программ можно загрузить с FTP-сервера издательства «БХВ» по ссылке: <ftp://ftp.bhv.ru/9785977568012.zip> или со страницы книги на сайте <https://bhv.ru/> (см. приложение 4).

8. Логические операции

В компьютерных технологиях под логикой понимаются неарифметические операции, предполагающие принятие решений вида «да/нет». У процессора ARM есть три логических оператора: AND, OR и EOR. Эти логические операции выполняются между соответствующими битами двух чисел. Результат их бывает двух видов: да или нет. В двоичном виде они представлены числами 1 и 0. Эти операции полезны, когда нужно определить состояние отдельных битов в имеющихся данных.

У каждой операции есть четыре своих набора правил.

Логическое И (AND)

Четыре правила операции AND:

- ◆ $0 \text{ AND } 0 = 0$ (0 AND 0 дает 0);
- ◆ $1 \text{ AND } 0 = 0$ (1 AND 0 дает 0);
- ◆ $0 \text{ AND } 1 = 0$ (0 AND 1 дает 0);
- ◆ $1 \text{ AND } 1 = 1$ (1 AND 1 дает 1).

Операция AND дает результат 1 только в том случае, если оба задействованных в операции бита равны 1. Если в любом из этих битов находится 0, результирующий бит всегда будет равен 0. Например:

```
  1010
  0011
AND= 0010
```

В результате поднятым остается только бит 1, а остальные биты очищаются, потому что в каждом случае один из соответствующих проверяемых битов содержит 0. В этих логических операциях важно помнить, что бит переноса отсутствует. Проверки проводятся на отдельных битах, и мы не складываем и не вычитаем целые числа.

Основное назначение операции AND — «замаскировать» или «сохранить» нужные биты. Например, чтобы сохранить младший полубайт байта (биты от 0 до 3) и полностью очистить старший полубайт (биты с 4 по 7), можно использовать оператор

AND, маскируя оригинал значением 00001111. Если теперь мы применим эту операцию, например, к числу 10101100, то получим следующее:

```

10101100
00001111
AND= 00001100

```

Четыре старших бита очищаются, а состояние младших четырех битов сохраняется.

Логическое ИЛИ (OR)

Четыре правила операции OR:

- ◆ 0 OR 0 = 0 (0 OR 0 дает 0);
- ◆ 1 OR 0 = 1 (1 OR 0 дает 1);
- ◆ 0 OR 1 = 1 (0 OR 1 дает 1);
- ◆ 1 OR 1 = 1 (1 OR 1 дает 1).

Операция OR дает результат 1, если один или оба бита содержат 1. Результат 0 получится только в том случае, если ни один из битов не содержит 1. Например:

```

1010
0011
OR= 1011

```

Здесь нулевым остался только бит 2 результата, все остальные биты подняты, поскольку каждая пара проверенных битов содержит по крайней мере одну единицу.

Одним из распространенных способов использования операции ИЛИ является установка определенного бита (или битов). Ее еще иногда называют принудительной установкой. Например, если вы хотите принудительно установить бит 0 и бит 7, вам нужно выполнить операцию ИЛИ между нужным байтом и числом:

```

00110110
10000001
OR= 10110111

```

Исходные биты числа сохраняются, но бит 0 и бит 7 «принудительно» становятся равными 1. Изначально эти биты были очищены. Остальные биты остаются без изменений.

Исключающее ИЛИ (EOR)

Операция EOR подчиняется четырем правилам:

- ◆ 0 EOR 0 = 0 (0 EOR 0 дает 0);
- ◆ 1 EOR 0 = 1 (1 EOR 0 дает 1);
- ◆ 0 EOR 1 = 1 (0 EOR 1 дает 1);
- ◆ 1 EOR 1 = 0 (1 EOR 1 дает 0).

Бит становится равным 1, если два соответствующих проверяемых бита были различны. Если они одинаковы, результат равен 0:

```

0101
1110
EOR= 1011

```

Эта команда часто используется для генерации дополнительного кода или инвертирования числа. Это делается путем выполнения операции EOR с числом 11111111:

```

00110110
11111111
EOR= 11001001

```

Сравните результат с исходным числом — они полностью противоположны: единицы превратились в нули, а нули — в единицы.

Команда `mvn`, представленная в *главе 6*, по сути, выполняет операцию EOR над операндом2.

Команды логических операций

Эти операции выполняются одинаково, независимо от объема данных. В приведенных в предыдущем разделе примерах мы обрабатывали числа шириной в один байт. Но и для четырех байтов (или столько же байтов, сколько вам нужно) все будет работать так же. Операция производится непосредственно на используемых битах, и никакие флаги регистра состояния не задействованы и не учитываются, и переносы тоже не генерируются.

Для выполнения трех основных логических операций используются команды AND, ORR и EOR. Синтаксис их такой же, как и у предыдущих команд:

```

AND (<Суффикс>) <Место назначения>, <Операнд1>, <Операнд2>
ORR (<Суффикс>) <Место назначения>, <Операнд1>, <Операнд2>
EOR (<Суффикс>) <Место назначения>, <Операнд1>, <Операнд2>

```

В этих случаях *операнд1* является регистром, а *операнд2* может быть регистром или конкретным значением. Сами операции не влияют на флаги регистра состояния, но можно поднять их принудительно с помощью суффикса.

Приведем несколько примеров использования этих команд:

```

AND R0, R0, #1           @ сохранение бита 0 в R0
ORR R1, R1, #2           @ подъем бита 1 в R1
EOR R2, R2, #255         @ инвертирование младшего байта R2

```

Взглянем теперь на небольшой, но интересный фрагмент кода:

```

MOV R0, #129
AND R0, R0, #1
ORR R0, R0, #2
EOR R0, R0, #255

```

В результате получится значение 0xFC, и вот как мы пришли к нему (работая только с младшим байтом слова):

```
Load 129          10000001
AND with 1        00000001
Result            00000001
OR with 2         00000010
Result            00000011
EOR with 255     11111111
Result            11111100
```

Рассмотрим далее несколько практических примеров типичного применения команд ORR и EOR.

Команда ORR для преобразования регистра символов

В программе 8.1 показано, как с помощью команды ORR преобразовать символ из верхнего регистра в нижний. Например, возьмем символ А и преобразуем его в а. ASCII-код символа «А» — 0x41 (65), а символа «а» — 0x61 (97). Сравнив эти коды, мы увидим, что разница между «А» и «а» составляет 0x20.

Поскольку оба этих символа являются начальными для своего раздела алфавита, получается, что разница между значением верхнего и нижнего регистра всегда будет одинаковой. На рис. 8.1 показано, как это выглядит в виде двоичных чисел.

ASCII	Значение	Двоичное число
А	0x41	0100 0001
а	0x61	0110 0001
Разница	0x20	0010 0000

Рис. 8.1. Разница ASCII-кодов символов А и а

Думаю, вы уже поняли, что мы можем получить такую разницу с помощью команды ORR с двоичным числом 0010 0000, или 0x20 (32).

ПРОГРАММА 8.1. Преобразование регистра символов

```
/* Использование команды ORR для переключения регистра символов */
.global _start
_start:
_read:          @ системный вызов чтения
MOV R7, #3     @ номер системного вызова
MOV R0, #0     @ Stdin - это клавиатура
MOV R2, #1     @ считывание одного символа
LDR R1,=string @ строка на метке string:
SWI 0
```

```

_togglecase:
    LDR R1, =string      @ адрес символа
    LDR R0, [R1]         @ загрузка его в R0
    ORR R0, R0, #0x20    @ смена регистра символов
    STR R0, [R1]         @ сохранение символа в регистр R0

_write:
    @ системный вызов записи
    MOV R7, #4           @ номер системного вызова
    MOV R0, #1           @ Stdout - это монитор
    MOV R2, #1           @ строка состоит из 1 символа
    LDR R1,=string      @ строка на метке string:
    SWI 0

_exit:
    @ системный выход
    MOV R7, #1
    SWI 0

.data
string:      .ascii " "

```

Основной функционал программы 8.1 приведен в разделе `togglecase`. Процедура начинается с чтения символа с клавиатуры (мы вводим прописную букву и нажимаем клавишу <Return>), который затем сохраняется на метке `string`. Затем процедура `togglecase` помещает адрес сохраненного символа в `R1` и использует метод, называемый *косвенной адресацией*, для загрузки символа в регистр `R0` (об этой форме адресации мы поговорим в главе 15). Значение в `R0` затем маскируется числом `0x20`, а метод косвенной адресации используется для сохранения измененного содержимого `R0` обратно по адресу, хранящемуся в `R1`.

Обратите внимание, что в этой программе мы не проверяем, находится ли введенный символ в диапазоне от `A` до `Z`. Подумайте теперь сами: как написать программу для преобразования строчных букв в прописные?

Очистка бита командой `BIC`

Команда `BIC` (`Bit Clear`) поднимает или очищает отдельные биты в регистрах или ячейках памяти. Синтаксис ее следующий:

```
BIC (<Суффикс>) <Место назначения>, <Операнд1>, <Операнд2>
```

Команда `BIC` сбрасывает отдельные биты переданного значения в ноль:

```
BIC R0, R0, #1111 @ очистка младших 4 битов регистра R0.
```

Если в регистре `R0` лежит значение `0xFFFFFFFF`, то в приведенном примере будут очищены четыре младших бита, т. е. останется `0xFFFFFFFF0`:

```

R0:      11111111 11111111 11111111 11111111
BIC #0xF 00000000 00000000 00000000 00001111
Result is: 11111111 11111111 11111111 11110000

```

Математически, команда BIC выполняет логическую операцию AND NOT между операндом1 и операндом2.

Проверка флагов

Существуют две команды, предназначенные для проверки состояния битов в слове. Как и в случае с командой CME, у полученного результата не будет места назначения, но он отражается непосредственно в регистре состояния (поэтому использование суффикса s не требуется). Эти команды: TeSt BiTs (TST) и Test EQuivalence (TEQ). Их синтаксис:

```

TST <Операнд1>, <Операнд2>
TEQ <Операнд1>, <Операнд2>

```

TST — это команда проверки бита, в которой в качестве операнда2 задается маска проверки операнда1. С математической точки зрения команда выполняет логическое И и обновляет флаг нуля в зависимости от результата:

```
TST R0, #128 @ проверка, установлен ли бит b7 в регистре R0
```

Команда TEQ — это проверка равенства, выполняемая через логическую операцию EOR. Она позволяет узнать, совпадают ли определенные биты в регистрах:

```
TEQ R0, R1 @ проверка равенства регистров R0 и R1
```

С командами TST и TEQ можно использовать суффиксы, что позволяет проверить другие условия, а также сам флаг нуля (подробнее об этом в следующей главе).

В программе 8.2 команда TST используется для преобразования числа, содержащегося в R6, в двоичное число, которое выводится на экран. Само выводимое число помещается в R6. В этой программе есть несколько интересных моментов, с которыми мы еще не сталкивались, и подробнее мы рассмотрим их в следующих главах. Обратите внимание, как программа разбита на именованные разделы.

ПРОГРАММА 8.2. Вывод двоичного числа

```

/* Преобразование числа в двоичное и его вывод */
.global _start
_start:
    MOV R6, #251                @ помещение числа в R6
    MOV R10, #1                 @ настройка маски
    MOV R9, R10, LSL #31
    LDR R1, = string           @ указатель на метку string

_bits:
    TST R6, R9                  @ проверка числа по маске
    BEQ _print0

```

```

MOV R8, R6           @ сохранение числа
MOV R0, #49         @ ASCII-код символа '1'
STR R0, [R1]        @ сохранение 1 в строке
BL _write           @ вывод на экран
MOV R6, R8          @ загрузка числа
BAL _noprnt1

```

`_print0:`

```

MOV R8, R6           @ сохранение числа
MOV R0, #48         @ ASCII-код символа '0'
STR R0, [R1]        @ сохранение 0 в строке
BL _write
MOV R6, R8          @ загрузка числа

```

`_noprnt1:`

```

MOVS R9, R9, LSR #1 @ смещение битов маски
BNE _bits

```

`_exit:`

```

MOV R7, #1
SWI 0

```

`_write:`

```

MOV R0, #1
MOV R2, #1
MOV R7, #4
SWI 0
MOV PC, LR

```

`.data`

`string: .ascii " "`

Кое-где в программе вы увидите команду `LSL`. Она выполняет логический сдвиг влево и используется для сдвига битов в слове — в нашем случае влево. В программе команда используется следующим образом:

```

MOV R10, #1
MOV R9, R10, LSL #31

```

Здесь число `#1` помещается в `R10` через `R9` и `31` раз сдвигается влево, в результате чего поднятым получается только самый старший бит в регистре. Все дело в том, что мы не можем сразу загрузить нужное значение в регистр из-за ограничений по возможным используемым числам (подробнее об этом чуть позже).

Таким образом, получается следующее:

```

MOV R10, #1: 00000000 00000000 00000000 00000001
LSL #31      10000000 00000000 00000000 00000000
              << 31 раз сдвигается влево <<

```

Теперь вернемся к циклу `bits`:

```

_bits:
    TST R6, R9                @ проверка числа по маске
    BEQ _print0
    MOV R8, R6                @ сохранение числа
    MOV R0, #49               @ ASCII-код символа '1'
    STR R0, [R1]              @ сохранение 1 в строке
    BL _write                  @ вывод на экран
    MOV R6, R8                @ загрузка числа
    BAL _noprint1

_print0:
    MOV R8, R6                @ сохранение числа
    MOV R0, #48               @ ASCII-код символа '0'
    STR R0, [R1]              @ сохранение 0 в строке
    BL _write                  @ вывод на экран
    MOV R6, R8                @ загрузка числа

```

В регистре `R6` лежит наше число. Мы знаем, что в маске поднят самый старший бит (`b31`), и команда `TST` проверяет, есть ли он в нашем числе. Если да, выполнится переход `BEQ` и будет выведено число 1. В противном случае выводится 0. Обратите внимание, что мы всегда сохраняем значение в `R6`, поскольку именно это число мы просматриваем, а регистр `R0` используется для вывода цифр 1 или 0 в подпрограмме `_write`. Значения ASCII-символов «1» (49) и «0» (48) помещаются в `R0` и сохраняются на метке `string` (этот метод уже должен быть вам знаком, поскольку мы делали так уже не раз).

В любом случае теперь мы используем подпрограмму `_write`, и нам достаточно лишь один раз собрать ее в программе. Переход к подпрограмме выполняется вот так:

```
BL _write
```

Команда `BL` означает `Branch with Link`. Когда это происходит, адрес следующей команды сохраняется, и программа переходит к названной метке. Если вы посмотрите на конец подпрограммы `_write`, она заканчивается этой строкой:

```
MOV PC, LR
```

Эта строка помещает сохраненный в регистре связи адрес обратно в программный счетчик, тем самым возобновляя выполнение программы после `BL`. Подобные методики более подробно рассматриваются в *главе 10*, так что скоро вы поймете, что тут к чему.

В разделе `_noprint1` мы используем логический сдвиг вправо, чтобы сдвинуть бит маски на одно место вправо и обновить флаги регистра состояния с помощью суффикса `s`. Программа продолжает работу и выводит единицы и нули, пока не будут проверены все 32 бита:

```

_noprint1:
    MOVS R9, R9, LSR #1      @ смещение битов маски

```

Эта программа прекрасно иллюстрирует, как реализуются битовые паттерны. Запустив программу, просмотрите все цифры в выводе. Станет ясно, что это просто двоичное представление исходного числа.

Вы можете попытаться улучшить эту программу, чтобы она просила ввод числа с клавиатуры, а затем выводила его в двоичном формате. Для этого вам потребуется преобразовать значение символа ASCII в шестнадцатеричное и сохранить его в регистре. Далее в книге будет рассказано, как это сделать.

В программе 8.2 есть пара моментов, когда требуется значение ASCII для цифр 1 или 0. В этих случаях мы использовали номер ASCII как непосредственное значение:

```
MOV R0, #49
```

Но можно также сделать это с помощью самого символа:

```
MOV R0, #'1'
```

Символ ASCII заключен здесь в одинарные кавычки. Такой код более читабелен, т. к. не заставляет гадать, что за символ скрывается за переданным кодом.

Регистры системных вызовов

Одним из недостатков использования вызовов операционной системы является необходимость повышенного внимания при обращении с регистрами. Большинству системных вызовов для правильной работы необходима информация, передаваемая им через определенные регистры. Поэтому, если вы планируете использовать системные вызовы, нужно подумать об использовании регистров заранее. Так вы сэкономите себе много времени, которое иначе ушло бы на последующее редактирование.

9. Условное выполнение

Об идее использования суффиксов мы говорили в предыдущих главах, когда добавляли суффикс *s* к разным командам для принудительного обновления флагов регистра состояния. Например:

ADDS R0, R1, R2 @ R0=R1+R2 и установка флагов

Без суффикса *s* команда ADD не влияет на флаги состояния. Но *s* — лишь один из множества существующих суффиксов, которые можно использовать аналогичным образом для придания дополнительных функций почти всем командам в наборе команд ARM.

При этом почти все команды ARM поддерживают суффиксы, которые позволяют выполнить команду, только в том случае, если истинно проверяемое условие. Если условие ложно, команда игнорируется. Суффикс *CS* обозначает «Carry Set» и проверяет наличие флага переноса, поэтому команда, к которой он добавляется, будет выполняться только тогда, когда флаг переноса в момент выполнения поднят. С точки зрения программирования, это дает вам возможность ограничить каждую инструкцию определенным условием.

Список кодов условий весьма велик и показан в табл. 9.1.

Таблица 9.1. Коды условий на языке ассемблера ARM

Суффикс	Описание
EQ	Равно
NE	Не равно
VS	Переполнение
VC	Нет переполнения
AL	Безусловное исполнение
NV	Безусловное неисполнение
HI	Беззнаковое больше
LS	Беззнаковое меньше или равно
PL	Знак «плюс»

Таблица 9.1 (окончание)

Суффикс	Описание
MI	Знак «минус»
CS/HS	Имеется перенос / беззнаковое больше или равно
CC/LO	Нет переноса / беззнаковое меньше
GE	Знаковое больше или равно
LT	Знаковое меньше
GT	Знаковое больше
LE	Знаковое меньше или равно

Ассемблер GCC понимает эти коды, и вы можете использовать их в своих программах, добавляя соответствующие буквы в конце мнемоники команды. Можно также оставлять пробелы между командой и кодом условия, чтобы программа легче читалась. То есть можно и так:

```
MOVCS R0, R1
```

и так:

```
MOV CS R0, R1
```

В коде

```
MOV CS R0, R1
```

содержимое регистра R1 будет перемещено в регистр R0, только если установлен флаг переноса.

Аналогично код

```
MOV CC R0,R1
```

перемещает содержимое R1 в R0, если снят флаг переноса.

Некоторые суффиксы изменяют более одного флага, и для этих операций может потребоваться, чтобы были установлены или сброшены определенные комбинации флагов. Таким образом, можно удобно разделить коды условий на два вида: те, которые выполняются в зависимости от одного флага регистра состояния, и те, которые выполняются в зависимости от двух или более флагов.

Коды условий работают в зависимости от состояния флагов, но сами на них не влияют. Для установки флага нужно использовать инструкцию сравнения или вариант команды с суффиксом *s*. Если вы будете понимать работу двоичных и арифметических операций, то поймете и то, как флаги условий влияют на команды.

В этой книге далее мы рассмотрим примеры использования условного выполнения. Например, в *главе 10* наглядно показано, как использование условных кодов может значительно уменьшить размер вашей программы.

В этой главе содержится очень много новой информации. Даже для опытного программиста такое обилие материала может показаться сложным, поэтому не беспокойтесь, если вдруг что-то сразу не поймете. Возвращайтесь при необходимости к непонятному содержимому и перечитывайте уже изученное. Используйте также табл. 9.1 в качестве шпаргалки.

Коды состояния с одним флагом

К этой группе относятся следующие суффиксы:

EQ, NE, VS, VC, MI, PL, CC, AL и NV

Эти условные флаги работают парами. Например, условия EQ и NE проверяют флаг нуля: одно проверяет, поднят ли флаг, а другое — снят ли он. Если вы проверяете одно условие, и оно ложно, вам не нужно проверять альтернативное условие, поскольку по определению оно должно быть истинным, т. к. они взаимоисключающие и противоположные.

EQ: равно

Z = 1. Команды с суффиксом EQ выполняются только в том случае, если установлен флаг нуля (Zero). А это произойдет, если предыдущая операция закончилась нулевым результатом. Вычитание двух одинаковых чисел дает в результате ноль, и соответственно поднимается флаг нуля. Операция сравнения поднимает флаг нуля, если сравниваемые значения оказываются одинаковыми. Если результат операции не равен нулю, флаг нуля сбрасывается ($Z = 0$).

Пример:

MOVS R0, R1	@ помещение значения R1 в R0 и подъем флага
MOVEQ R0, #1	@ если в результате 0, в R0 помещается значение 1

Флаг нуля поднимается в случае, если из R1 в R0 будет перемещено значение 0. Тогда будет выполнена следующая команда и в R0 запишется значение 1. Команда не будет выполнена, если флаг нуля окажется сброшен, а это возможно, если значение в R0 было ненулевым.

NE: не равно

Z = 0. Команды с суффиксом NE выполняются только в том случае, если флаг нуля сброшен. А это происходит, если предыдущая операция не дала нулевой результат. Вычитание двух разных чисел сбрасывает флаг. Операция сравнения поднимает флаг нуля, если сравниваемые значения оказываются одинаковыми. Если результат операции не равен нулю, флаг нуля сбрасывается ($Z = 0$).

Пример:

CMP R5, R6	@ сравнение R6 с R5 и подъем флагов
ADDNE R5, R5, R6	@ если не равны, сложение R5 + R6 и помещение результата в R5

В этом коде команда `CMR` используется для сравнения содержимого регистров `R5` и `R6`. Если значения в регистрах не совпадают (и флаг нуля сброшен, $Z = 0$), то значения в `R5` и `R6` складываются и результат помещается в `R5`.

VS: переполнение

$V = 1$. Команды с суффиксом `vs` выполняются в случае, если поднят флаг переполнения. Этот флаг поднимается в результате арифметических операций, результат которых не вмещается в 32-битный регистр назначения, т. е. может возникнуть ситуация переполнения. В подобных случаях данные, помещаемые в регистр назначения, могут оказаться неверными, и тогда требуются корректирующие меры со стороны программиста. Примеры этого приведены в *главе 5*.

VC: нет переполнения

$V = 0$. Команды с суффиксом `vc` выполняются в случае, если флаг переполнения сброшен. Этот флаг поднимается в результате арифметических операций, результат которых не вмещается в 32-битный регистр назначения, т. е. может возникнуть ситуация переполнения. Если флаг снят, значит, переполнения не возникло, что и проверяется этим условием.

MI: знак «минус»

$N = 1$. Команды с суффиксом `mi` выполняются в случае, если поднят флаг минуса (Negative). Этот флаг поднимается после выполнения арифметических операций, дающих отрицательный результат. Это может произойти, если, например, вычесть большое число из меньшего. Логические операции тоже могут поднимать флаг минуса, если в их результате бит 31 регистра назначения становится равен единице.

Пример:

```
SUBS R1, R1, #1      @ вычитание 1 из R1 и подъем флагов
ADDMI R0,R0, #15    @ если флаг минуса поднят,
                   @ прибавление значения 0x0F к R0
```

В этом фрагменте кода команда `SUB` вычитает 1 из содержимого регистра `R1`, для обновления флагов используется суффикс `S`, а результат сохраняется в регистре `R1`. Команда `ADD` в следующей строке выполняется только в том случае, если установлен флаг `N`, и в этом случае к значению в регистре `R0` прибавляется 15.

PL: знак «плюс»

$N = 0$. Команды с суффиксом `pl` выполняются в случае сброса флага минуса. Он сбрасывается, если результат арифметической операции оказывается больше или равным нулю. Обратите внимание, что суффикс `EQ` проверяет только ноль, а вот суффикс `PL` проверяет неотрицательный результат. Важно понимать разницу.

Пример:

SUBS R1, R1, #1	@ вычитание 1 из R1 и подъем флагов
ADDMI R0, R0, #15	@ если флаг минуса поднят, @ прибавление к R0 значения 0x0F
ADDEPL R0, R0, #255	@ если флаг минуса сброшен, @ прибавление к R0 значения 0x0F

Этот пример демонстрирует, как использование условий позволяет получать от кода разные результаты. Код этого примера дополняет приведенный ранее пример суффикса MI: если результат оказывается положительным числом, к содержимому регистра R0 прибавляется число 255. Очевидно, что выполниться может только одна из этих команд и от результата команды SUBS зависит, какая именно сработает. Поскольку у команд ADD не использовался суффикс S, флаги состояния после команды CMP не изменились.

CS: имеется перенос (HS: беззнаковое больше или равно)

$C = 1$. Команды с суффиксом CS или HS выполняются в случае, если поднят флаг переноса. Этот флаг поднимается, когда арифметическая операция дает результат, превышающий размерность в 32 бита. Флаг переноса — это своего рода 33-й бит. Его также можно установить с помощью операции сдвига ARM, которую мы рассмотрим в *главе 11*.

Пример:

ADDS R0, R0, #255	@ прибавление 0xFF к значению R0 @ и сохранение результата в R0
ADDCS R1, R1, #15	@ если поднят перенос, прибавление 0x0F к R1 @ и сохранение результата в R1

CC: нет переноса (LO: беззнаковое меньше)

$C = 0$. Команды с суффиксом CC или LO выполняются в случае, если флаг переноса сброшен. Это происходит, если арифметическая операция создает результат, уступающий в пределах 32 битов. На флаг переноса также влияют операции сдвига процессора ARM, которые мы рассмотрим в *главе 11*.

Пример:

ADDS R0, R0, #255	@ прибавление 0xFF в значению R0 @ и сохранение результата в R0
ADDCS R1, R1, #15	@ если поднят перенос, прибавление 0x0F к R1 @ и сохранение результата в R1
ADDC R1, R1, #128	@ если поднят перенос, прибавление 0xF0 к R1 @ и сохранение результата в R1

Как и в примере с командой PL, здесь выполнится то или иное действие в зависимости от состояния флага переноса.

AL: безусловное исполнение

Команды с суффиксом AL выполняются всегда, независимо от флагов в регистре состояния. В целом команды без суффиксов и так выполняются всегда, поэтому суффикс AL используется со всеми командами по умолчанию.

Пример:

```
ADDAL, R0,R1,R2    @ сложение R1 и R2 и сохранение результата в R0
ADD R0,R1,R2       @ сложение R1 и R2 и сохранение результата в R0
```

Эти команды дают одинаковый результат.

Суффикс AL обычно используется с командой ветвления, чтобы она стала трехбуквенной и более читаемой:

```
B start            @ переход к метке start
BAL start          @ переход к метке start
```

NV: безусловное неисполнение

Команды с суффиксом NV не выполняются никогда, независимо от флагов в регистре состояния. Этот суффикс добавлен в синтаксис языка для полноты картины. Его можно использовать для создания свободного пространства в программе во время сборки. Это пространство можно использовать для хранения данных или изменений самой программы, а иногда для организации конвейеров (об этом см. в *главе 12*).

Пример:

```
ADDNV R0, R1, R2    @ сложение не выполнится никогда
```

Коды, проверяющие несколько флагов

В эту группу попадают шесть суффиксов:

HI, LS, GE, LT, GT и LE

Эти условия проверяют два или более флагов регистра состояния. Чаще всего они используются после команды CMP или CMN. Этот набор кодов тоже можно подразделить на две группы: те, которые работают с числами без знака (HI и LS), и те, которые работают с числами со знаком (GE, LT, GT и LE).

HI: беззнаковое больше

C = 1 И Z = 0. Команды с суффиксом HI выполняются только в том случае, если флаг переноса C (Carry) поднят, а флаг нуля Z (Zero) сброшен. Это происходит в случае, когда при сравнении *операнд1* оказывается больше *операнда2*.

Пример:

```
CMP R10, R5        @ сравнение значений в регистрах R10 и R5
MOVHI R10,#0       @ если R10 > R5, обнуление регистра R10
```

Важно помнить: это условие предполагает, что сравниваемые значения беззнаковые, т. е. не используются отрицательные значения в дополнительном коде.

LS: беззнаковое меньше или равно

C = 0 ИЛИ Z = 1. Команды с суффиксом LS выполняются только в том случае, если флаг переноса сброшен или флаг нуля поднят. Это происходит в случае, когда при сравнении *операнд1* оказывается меньше *операнда2*. Опять же, это условие тоже предполагает, что оба сравниваемых числа беззнаковые.

Пример:

```

CMP    R10, R5           @ сравнение значений в регистрах R10 и R5
ADDLS R10,R10,#1        @ если R10 <= R5, прибавление 1
                               @ и сохранение результата в R10

```

GE: знаковое больше или равно

(N = 1, V = 1) ИЛИ (N = 0, V = 0). Эта команда выполняется при равенстве значений флагов N (Negative) и V (Overflow). Это происходит, когда при сравнении *операнд1* оказывается больше *операнда2* или равен ему.

Пример:

```

CMP    R5, R6           @ сравнение содержимого R5 и R6
ADDGE R5,R5,#255       @ если R5 >= R6, прибавление 0xFF к R5

```

А вот это условие уже предполагает, что два сравниваемых значения являются числами со знаком.

LT: знаковое меньше

(N = 1, V = 0) ИЛИ (N = 0, V = 1). Команда выполняется, если флаги Negative и Overflow имеют разные значения. Это происходит, если *операнд1* меньше *операнда2*. Опять же, условие предполагает, что оба сравниваемых значения являются величинами со знаком.

Пример:

```

CMP    R5, #255         @ сравнение содержимого R5 с 0xFF
SUBLT R5,R5,R6         @ если R5 < 0xFF, вычитание R6 из R5,
                               @ сохранение результата в R5

```

GT: знаковое больше

(N = 1, V = 1) ИЛИ (N = 0, V = 0) И Z=0. Эта команда выполняется, если результат операции оказывается положительным числом, и при этом не нулем. *Операнд1* должен быть больше, чем *операнд2*, и предполагается, что используются числа со знаком. Таким образом, флаг минуса и флаг переполнения должны иметь одинаковые значения, а нулевой флаг должен быть снят.

Пример:

```

CMP R5, R6           @ сравнение R5 с R6
ADDGT R0,R1,R2      @ если R5 > R6, сложение R1 + R2 и
                    @ помещение результата в R0

```

LE: знаковое меньше или равно

(N = 1, V = 0) ИЛИ (N = 0, V = 1) ИЛИ Z = 1. Эта команда выполняется в случае, если при сравнении *операнд1* оказывается меньше *операнда2* или равен ему. Предполагается, что используются числа со знаком. Для этого и флаг минуса, и флаг переполнения должны быть разными, а флаг нуля должен быть поднят.

Пример:

```

CMP R5, #10         @ равно ли значение в регистре R5 0x0A?
SUBLE R0,R1,R2     @ если R5 <= 0x0A, вычитание R2 из R1 и
                    @ помещение результата в R0

```

Добавление суффикса S

Суффикс *s* можно комбинировать с условными суффиксами. В этом случае результат любого выполняемого действия также повлияет на флаги состояния. В предыдущих примерах мы видели, что использование флага после действия позволяет поступать в зависимости от результата условия. В этом случае предполагается, что флаги регистра состояния обновляться не будут. Если вы хотите, чтобы флаги регистра состояния обновлялись после условной операции, то после условного суффикса следует использовать также суффикс *s*:

```

ADDCSS R0, R1, R2   @ сложение R2 с R1, если Carry = 1, а также
                    @ обновление флагов регистра состояния

```

Суффикс *s* должен обязательно располагаться именно после кода условия, иначе ассемблер пропустит его.

Поэтому вот такой код:

```
ADDSCS R0, R1, R2
```

выдает ошибку.

10. Ветви и сравнения

В этой главе мы более подробно рассмотрим команды сравнения и самые экономичные способы их применения.

Команды ветвления

Стандартная программа машинного кода выполняется линейно, т. е. действия идут одно за другим. Команды ветвления позволяют передавать поток выполнения в другую точку программы, где она снова начинает выполняться линейно, пока не наткнется на другую команду ветвления. Команды ветвления чаще всего выполняются в двух вариантах:

```
B (<Суффикс>) <Метка>  
BL (<Суффикс>) <Метка>
```

Добавив к этому условные флаги, мы можем завести ветвь кода для каждого случая. Можно использовать команду в обособленно, но предпочтительнее добавлять суффикс `AL`, чтобы буква `v` не затерялась в коде большой программы:

```
BAL start
```

Но возможен и такой вариант:

```
B start
```

Метка — это некоторое место в программе на языке ассемблера. Расстояние, на которое может «прыгать» ветвь, ограничено. Оно составляет ± 32 Мбайт, т. к. это наибольший адрес, который может быть представлен в пространстве, выделенном для позиции метки. Именованная метка не хранится ни в машинном коде, ни в абсолютном адресе метки. Вместо этого сохраняется ее смещение по коду относительно текущей позиции. Когда процессор ARM встречает инструкцию ветвления, он обрабатывает следующее значение как положительное (вперед) или отрицательное (назад) смещение в регистре счетчика команд (Program Counter, PC) от текущего положения.

В *главе 13* мы более подробно рассмотрим регистр `R15` и обсудим, как работают ветви.

Регистр ссылок

Команда `Branch with Link`, или `BL`, позволяет передать управление другой части вашей программы (некоторой подпрограмме), а затем вернуться в основной код. `BL` работает как обычная команда ветвления, т. е. принимает в качестве места назначения адрес, обычно задаваемый в виде метки в программе. Перед переходом по ветви команда копирует содержимое счетчика команд (`R15`) в регистр ссылок (`R14`).

`BLEQ subroutine` @ Переход и сохранение значения `PC`,
 @ если поднят флаг `Z`

После завершения подпрограммы содержимое регистра ссылок (`Link Register`) передается в счетчик команд, возвращая управление вызывающему сегменту кода:

```
MOV R15, R14
```

Возможно, такая реализация команд на чипе `ARM` не слишком элегантна. Но этот процессор эффективен и хорошо делает свою работу. У других процессоров есть особые команды для вызова и подпрограмм, и возврата из них. Например, на микросхеме `6502` для перехода к подпрограммам и возврата к ним используются команды `JSR <метка>` и `RTS`.

Для перемещения возвратного адреса из `R14` обратно в счетчик команд используется команда `MOV`. Она не влияет на флаги регистра состояния, поэтому их состояние сохранится в том же виде, что и до возврата.

Важно помнить, что при выполнении команды `BL` содержимое регистра `R15` копируется в `R14`. Получается, что если работает одна подпрограмма и в этот момент вызывается другая, исходный адрес ссылки затирается новым адресом.

Если вызовы `BL` в вашей программе оказываются вложены друг в друга, регистр ссылок нужно каждый раз сохранять вручную. После этого следует записывать в него адрес возврата каждый раз, когда подпрограмма завершается. И тут крайне важна аккуратность. Повторная передача неправильного адреса обратно в счетчик команд, скорее всего, приведет к сбою вашей программы.

Одним из распространенных способов хранения таких вложенных адресов является *стек*. О нем мы поговорим в *главе 17*.

Использование команд сравнения

В своих программах вы часто будете проверять результат операции, а затем, в зависимости от этого результата, предпринимать какие-либо действия. Есть несколько команд, которые позволяют выполнить такую проверку, а также ряд команд, которые позволяют «прыгать» к другой части программы. Эти команды сравнения напрямую влияют на флаги регистра состояния, после чего вы можете указать нужные вам в зависимости от результата действия. В следующем коде выполняется подсчет от 1 до 50, при этом цикл организован с помощью команд сравнения и ветвления:

```

MOV R0, #1           @ инициализация счетчика
loop:
ADD R0, R0, #1      @ прибавление единицы
CMP R0, #50         @ сравнение с предельным значением
BLE loop

```

Цикл будет прибавлять 1 к значению регистра R0, которое изначально равно 1. Затем R0 сравнивается с 50, и если значение в регистре меньше или равно 50, срабатывает переход BLE. Этот цикл продолжается до достижения значения R0 = 50. Затем цикл выполняется еще раз, чтобы значение в R0 стало равно 51, потому что именно в этот момент команда CMP уже не позволит команде BLE совершить переход.

Приведенный код уже неплох, но мы можем сократить его еще сильнее, реализовав обратный отсчет:

```

MOV R0, #50         @ инициализация счетчика
loop:
SUBS R0, R0, #1     @ вычитание единицы
BNE loop           @ переход, если не получен ноль

```

С помощью команды SUBS мы уменьшаем значение в регистре и устанавливаем флаги, что позволяет избавиться от команды CMP. В целом для подсчета известного числа итераций, если значение счетчика ни для чего другого не нужно, лучше и эффективнее вести обратный отсчет. Так можно использовать меньше команд, и код будет выполняться быстрее.

Применяем дальновидное мышление

Поскольку команды сравнения, по сути, просто проверяют флаги в регистре состояния, заранее подумав о конечной цели кода, вы можете обойтись и без них. Чтобы не быть голословными, рассмотрим пример. В приведенном далее коде реализован цикл, который будет повторяться до тех пор, пока значения в R0 и R1 не сравняются. Если значение в R0 больше, чем в R1, программа вычитает R1 из R0 и помещает результат в R0. Если же R0 меньше R1, программа вычитает R0 из R1 и помещает результат в R1. Когда значения регистров сравняются, программа завершится:

```

MOV R0, #100        @ произвольные значения в R0 & R1
MOV R1, #20
loop:
CMP R0, R1          @ сравнение: Z = 1?
BEQ stop            @ если да, метка stop
BLT less            @ если R0 < R1, метка less
SUB R0, R0, R1      @ в противном случае вычитание R1 из R0
BAL loop            @ возврат к метке start
less:
SUB R1, R1, R0      @ вычитание R0 из R1
BAL loop            @ возврат к метке start
stop:

```

Этот код неплох и правильно выполняет свою работу, но мы можем сократить его, перейдя на условное выполнение команд:

```

MOV R0, #100          @ произвольные значения в R0 & R1
MOV R1, #20

loop:
CMP R0, R1           @ сравнение: Z = 1?
SUBGT R0,R0,R1       @ вычитание R1 из R0, если больше
SUBLT R1,R1,R0       @ иначе вычитание R0 из R1
BNE loop             @ переход в случае неравенства

```

Чтобы проверить условия «больше» и «меньше», мы можем напрямую использовать суффиксы GT и LT в конце команды вычитания SUB.

Эффективное использование условных операторов

В главе 8 мы увидели, как можно использовать команду TST для вывода двоичного числа. В программе 8.2 использовался приведенный далее код, в котором выводилось 1 или 0:

```

_bits:
TST R6, R9           @ проверка числа по маске
BEQ _print0
MOV R8, R6           @ сохранение числа
MOV R0, #49          @ ASCII-код символа '1'
STR R0, [R1]         @ сохранение 1 в строке
BL _write            @ вывод на экран
MOV R6, R8           @ загрузка числа
BAL _noprint1

_print0:
MOV R8, R6           @ сохранение числа
MOV R0, #48          @ ASCII-код символа '0'
STR R0, [R1]         @ сохранение 0 в строке
BL _write            @ вывод на экран
MOV R6, R8           @ загрузка числа

```

На первый взгляд, это вполне удобный код, позволяющий вывести на экран 1 или 0 в зависимости от результата теста. Да, этот код неплох, но в нем недостает грамотного использования набора команд ARM. Рассмотрим листинг программы 10.1 и новый раздел кода `_bits`, который теперь стал вдвое меньше исходного за счет использования условных команд. Теперь, в зависимости от результата команды TST, выполняется одна из команд MOV (какая именно — зависит от флага нуля). Такой вариант намного элегантнее и проще.

ПРОГРАММА 10.1. Условное выполнение позволяет сократить программу
--

```

/* Преобразование числа в двоичное и его вывод */
.global _start
_start:
    MOV R6, #251           @ помещение числа в R6
    MOV R10, #1           @ настройка маски
    MOV R9, R10, LSL #31
    LDR R1, = string      @ указатель на метку string
_bits:
    TST R6, R9            @ проверка числа по маске
    MOVEQ R0, #48         @ ASCII-код символа '0'
    MOVNE R0, #49        @ ASCII-код символа '1'
    STR R0, [R1]          @ сохранение 1 в строке
    MOV R8, R6            @ сохранение числа
    BL _write             @ вывод на экран
    MOV R6, R8            @ загрузка числа

    MOVS R9, R9, LSR #1  @ смещение битов маски
    BNE _bits
_exit:
    MOV R7, #1
    SWI 0

_write:
    MOV R0, #1
    MOV R2, #1
    MOV R7, #4
    SWI 0
    BX LR                @ Branch eXchange

.data
string: .ascii " "

```

Обмен ветвей

Команды Branch Exchange (BX) и Branch with Link (BLX) — это третий способ ветвления внутри программы. Отметим, что они чаще всего используются для выполнения входа в код Thumb — подмножество команд ARM, которые мы рассмотрим в главе 27. Пока вы не ознакомитесь с последствиями работы этих команд, стоит их избегать. А как можно переписать подпрограмму так, чтобы избавиться от команды BX?

11. Сдвиги и вращения

В ARM есть внутренний механизм, называемый «barrel shifter», который позволяет сдвигать биты в слове влево или вправо. Большинство микропроцессоров имеют отдельные команды, напрямую выполняющие эти действия. Но в ARM такие перемещения выполняются только в рамках других команд. Подобные операции весьма полезны, поскольку сдвиг битов влево или вправо дает возможность, например, быстро делить или умножать числа.

Вы можете выполнять три вида сдвигов: логический сдвиг, арифметический сдвиг и вращение. Вращение — единственная сдвиговая операция, у которой нет арифметической функции и которая используется исключительно для перемещения битов. В табл. 11.1 приведены шесть имеющихся типов перемещений битов.

Таблица 11.1. Команды сдвига Raspberry Pi

Мнемоника	Описание
LSL	Логический сдвиг влево
LSR	Логический сдвиг вправо
ASL	Арифметический сдвиг влево
ASR	Арифметический сдвиг вправо
ROR	Вращение вправо
RRX	Расширенное вращение вправо

Хотя barrel shifter во время переключения работает и «вращается», на практике его работа вполне прозрачна для пользователя.

Логические сдвиги

Логический сдвиг числа влево или вправо на одну позицию приводит к его увеличению или уменьшению вдвое. Используя разное число логических сдвигов, вы можете умножать и делить числа.

На рис. 11.1 показано, как одиночный логический сдвиг влево (LSL) перемещает биты в слове данных. После выполнения команды LSL старший бит (b31) выпадает и попадает во флаг переноса, а пустое место, образованное путем сдвига из b0 в b1, заполняется нулем.

LSL	C	Слово										
До	x	b31	b30	b29	b28	b27	...	b3	b2	b1	b0	
После		b31	b30	b29	b28	b27	b26	<<	b2	b1	b0	0

Рис. 11.1. Логический сдвиг битов влево

Рассмотрим однобайтное двоичное число 00010001. В десятичном виде это 17. Выполнив логический сдвиг этого числа на одну позицию влево (LSL #1), мы получим число 00100010, т. е. 34. Фактически мы умножили исходное число на два. В процессе умножения мы отбросили верхнюю цифру и вставили 0 в самый младший бит. Подробнее это показано на рис. 11.2.

	b7	b6	b5	b4	b3	b2	b1	b0	
До	0	0	0	1	0	0	0	1	#17
После	0	0	1	0	0	0	1	0	<0 #34

Рис. 11.2. Удвоение числа командой LSL

Здесь мы рассмотрели однобайтовое число. В ARM используются четырехбайтовые слова, и все четырехбайтовое слово будет сдвинуто влево. Бит, который изначально находился в b7, перемещается в следующий байт, т. е. в b8, и т. д. Самый старший бит, а именно бит 31, перемещается во флаг переноса. После этого можно проверить флаг переноса и посмотреть, возникло ли при умножении переполнение.

Как уже было сказано, у процессора ARM нет отдельных команд для сдвига, но вместо этого они реализованы как дополнения к *операнду2* для использования в других командах, и эти дополнения влияют на все 32 бита указанного регистра. Приведенный ранее пример можно изобразить в виде кода следующим образом:

```
MOV R1, #17
MOVS R0, R1, LSL#1
```

Обратите внимание на структуру синтаксиса этого фрагмента. *Операнд1* — это место назначения для результата (R0), а команда LSL выполняется на *операнде2* (R1). В этом примере логический сдвиг задается как непосредственное значение, но можно задавать его и через регистр, что позволяет задавать любое исходное значение. В команде сдвига можно использовать значение от 0 до 31. Таким образом, команда:

```
LSL #5
```

умножит исходное значение на два пять раз, т. е. на 32 ($2 \times 2 \times 2 \times 2 \times 2$). При этом команда LSL выполнится пять раз. Образовавшиеся младшие разряды будут заполнены нулями, а во флаге переноса окажется значение последнего бита, «выпавше-

го» из b_{31} . Все остальные выпавшие биты будут «потеряны». Это означает, что умножение может быть выполнено верно только при условии, что не будут утеряны никакие значимые биты. Следовательно, для больших чисел необходимо следить за тем, чтобы значение из флага переноса сохранялось. Иными словами, результат должен уместиться в 32 бита. Отметим, что такое умножение не работает, если используются числа в дополнительном коде.

Логический сдвиг вправо

На рис. 11.3 показано, как логический сдвиг вправо (LSR) влияет на биты в слове данных. Старший бит (b_{31}) сдвигается вправо, а на его месте появляется 0. Младший бит (b_0) попадает во флаг переноса.

LSR	C	Слово										
До	x	b_{31}	b_{30}	b_{29}	b_{28}	b_{27}	...	b_3	b_2	b_1	b_0	
После	b_0	0	b_{31}	b_{30}	b_{29}	b_{28}	>>	b_4	b_3	b_2	b_1	

Рис. 11.3. Логический сдвиг вправо

По сути, команда LSR делит число на два. На рис. 11.4 приведен пример того, как это работает. Изначально у нас число 34, и над ним мы выполняем команду LSR #1, получая в итоге наше исходное значение — 17. На верхнем конце (b_{31}) появляется 0, а любое значение, выпадающее справа (b_0), попадает во флаг переноса. Как и в случае с командой LSL, флаг переноса «ловит» выпадающие значения, чтобы при необходимости их можно было бы посмотреть.

	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0	
До	0	0	1	0	0	0	1	0	#34
После	0	0	0	1	0	0	0	1	>0 #17

Рис. 11.4. Деление числа на два командой LSR

Арифметический сдвиг вправо

При арифметическом сдвиге знаковый разряд сохраняется. В следующем примере сохраняется бит b_{31} , все остальные смещаются на одну позицию вправо, а b_0 опускается во флаг переноса. Здесь сдвиг выполняется лишь единожды, однако тот же принцип работал бы и для нескольких сдвигов. Но бит b_{31} , знаковый разряд, всегда сохраняется, а последний бит, выпавший из b_0 , попадает во флаг переноса. Это показано на рис. 11.5.

ASR	C	Слово										
До	x	b_{31}	b_{30}	b_{29}	b_{28}	b_{27}	...	b_3	b_2	b_1	b_0	
После	b_0	b_{31}	b_{31}	b_{30}	b_{29}	b_{28}	>>	b_4	b_3	b_2	b_1	

Рис. 11.5. Арифметический сдвиг вправо с сохранением знакового разряда

Преимущество команды `ASR` состоит в том, что при сдвиге учитывается знак данных, и поэтому ее можно использовать с отрицательными числами. Исходный знак числа переходит из `b31` в `b30`, что дает правильное деление как положительных, так и отрицательных чисел:

```
MOV R1, #255
MOV R2, #1
MOVS R0, R1, ASR R2
```

Приведенный здесь фрагмент кода поместит значение `0x7F` (128) в регистр `R0`, не меняя при этом `R1` и `R2`. Кроме того, поднимется флаг переноса.

Условные тесты можно также использовать с уже привычной для нас инструкцией `MOV`. Следующая строка кода выполнится только в случае, если поднят флаг переноса:

```
MOV CS S R0, R1, ASR R2
```

Арифметический сдвиг влево (`ASL`) работает так же, как `LSL`, и результат получается одинаковым. Однако вам следует использовать команду `LSL` вместо `ASL`, поскольку ряд ассемблеров могут не компилировать ее и выдавать сообщение об ошибке. Некоторые же лишь выдают предупреждение.

Вращение

Существуют две команды, которые позволяют вам прокручивать биты вправо, действуя при этом флаг переноса. Вращение вправо (`ROR`) перемещает биты из нижних разрядов и подает их обратно в верхние. Последний «прокрученный» бит также копируется во флаг переноса и перемещается в старший разряд. На рис. 11.6 показано, как это работает. У команд вращения нет прямо соответствующего им арифметического действия, и они используются просто для сдвига битов.

ROR>	C	Слово										
До	x	b31	b30	b29	b28	b27	...	b3	b2	b1	b0	
После	b0	b0	b31	b30	b29	b28	>>	b4	b3	b2	b1	

Рис. 11.6. Команда `ROR`

Следующий код:

```
MOV R1, #0xF000000F
MOVS R0, R1, ROR #4
```

даст результат `0xFF000000` и поднимает флаги минуса и переноса. Команда `ROR #4` сдвигает биты на четыре позиции вправо.

Верхние байты `0xF000` перемещаются вправо на четыре позиции, что дает нам число `0x0F00`. Младшие байты `0x000F` перемещаются вправо на четыре позиции, что дает `0x0000`. Число `0xF`, выпавшее из младшего байта, возвращается в верхние биты старшего байта, давая `0xFF00`. Разумеется, биты в середине тоже сдвигаются, но

поскольку они равны нулю, это не заметно. Наконец, создается копия последнего бита, который изначально находился в позиции бита 4, и она помещается во флаг переноса.

Расширенное вращение

Существует также расширенная версия команды Rotate Right под названием `RRX`:

```
MOV R0, R1, RRX
```

Команда `RRX` уникальна тем, что вы не можете сказать ей, сколько сделать сдвигов, поскольку она позволяет выполнить лишь один. Команда `RRX` сдвигает данные вправо на одну позицию. Значение флага переноса попадает в `b31`, а значение из `b0` перемещается в флаг переноса. На рис. 11.7 показано, как это работает.

Все биты сохраняются, меняется лишь их порядок. При этом команда `RRX` использует флаг переноса как 33-й бит.

<code>RRX></code>	C	Слово										
До	x	b31	b30	b29	b28	b27	...	b3	b2	b1	b0	
После	b0	x	b31	b30	b29	b28	>>	b4	b3	b2	b1	

Рис. 11.7. Сдвиг вправо с помощью расширенного вращения

Использование сдвигов и вращений

Команды сдвига и вращения можно использовать в сочетании с любой из следующих команд обработки данных:

```
ADC, ADD, AND,
BIC,
CMN, CMP,
EOR,
MOV, MVN,
ORR,
RSB,
SBC, SUB,
TEQ, TST
```

Эти команды также можно задействовать для управления значением индекса для команд `LDR` и `STR`, о чем я расскажу в главах 13 и 15. Там же будут приведены и другие примеры использования этой группы модификаторов. В главе 12 будет проиллюстрировано применение и некоторых логических команд.

Прямой постоянный диапазон

Мы уже видели, что в командах можно использовать прямые константы:

```
SUB R0, R1, #3
```

Здесь в *операнде2* указана прямая константа, в нашем случае 3. К сожалению, размер числа, которое может быть указано в константе, ограничен, а некоторые числа и вовсе нельзя использовать. Например, число 257.

Причина этого кроется в принципе кодирования команд ARM. Для хранения прямой константы в качестве операнда выделяется всего 12 битов (кодирование команд ARM в этой книге не рассматривается). Но вернемся к 12 битам. Процессор ARM использует эти биты следующим образом: 12-битное поле делится на две части: одна занимает 8 битов, вторая — 4. 8-битная часть служит для хранения числовой константы, а в 4-битном поле хранится одна из 16 возможных позиций (каждая из которых сдвигается на два), на которые 8-битовое значение может быть сдвинуто.

На рис. 11.8 приведена эта схема, показывающая, как 8-битное число располагается в пределах 32 битов в зависимости от значений битов положения (от 0 до 15). Знаки «+» на этой схеме заменяют нули, чтобы от чтения глаза не разбегались. В столбце ROR содержится значение, которое будет использоваться при сдвиге.

Bit31	bit0	Psn	ROR
+++++76543210		0	0
10+++++765432		1	2
3210+++++7654		2	4
543210+++++76		3	6
76543210+++++		4	8
++76543210+++++		5	10
+++76543210+++++		6	12
++++76543210+++++		7	14
+++++76543210+++++		8	16
+++++76543210+++++		9	18
+++++76543210+++++		10	20
+++++76543210+++++		11	22
+++++76543210+++++		12	24
+++++76543210+++++		13	26
+++++76543210+++++		14	28
+++++76543210++		15	30

Рис. 11.8. Расчет возможных констант

Для пояснения рассмотрим пару примеров. Предположим, мы хотим использовать константу 173. В двоичном формате она выглядит так:

```
00000000 00000000 00000000 10101101
```

Это значение умещается в 8-битном формате, поэтому сдвиг не требуется, а биты положения будут равны 0.

Давайте теперь рассмотрим число 19968. В двоичном формате оно выглядит так:

```
00000000 00000000 01001110 00000000
```

Если мы сравним это с образцами на рис. 11.8, то увидим, что в нашем случае значение сдвинуто в позицию 12. Чтобы использовать такое значение в качестве операнда, мы должны взять число 78 (01001110) и сдвинуть его вправо на 24.

Таким образом, мы получаем второй способ задать константу — через *сдвиг*. Это делается с использованием следующего синтаксиса:

Команда (<Суффикс>) <Оп1>, <Оп2>, <Оп3> <Сдвиг>

Пример:

```
MOV R1, #78
MOV R0, R1, ROR #24
```

Здесь в регистр R1 загружается значение 78, затем оно сдвигается вправо на 24 позиции, а результат помещается в R0. Получается число 19968. Конечно, мы можем использовать все эти значения как прямые константы, поскольку ассемблер допускает это, т. е. можно написать прямо вот так:

```
MOV R0, #19968
```

и ассемблер это примет. Проблемы начинаются, когда мы хотим использовать значения, которые нельзя рассчитать по схеме, показанной на рис. 11.8.

Число 257 в качестве прямой константы использовать нельзя, но его можно применить, предварительно сохранив, а затем задействовать регистр в качестве операнда. Такой код:

```
ADD R0, R1, #257
```

вызовет ошибку:

```
Invalid constant
```

Но можно написать и так:

```
MOV R2, #256           @ помещение в R2 числа 256
ADD R2, R2, #1        @ прибавление 1, чтобы получить 257
ADD R0, R1, R2        @ прибавление 257 к R1
                       @ и сохранение результата в R0
```

Можно использовать команду MOV, а можно опустить ее и применить логические команды напрямую, получив тот же результат:

```
LSL R0, R1, #1
LSR R0, R1, #2
LSR R0, R1, #3
ROR R0, R1, #4
RRX R0, R1
```

Движение вверх

Команда `MOVT` позволяет загружать в регистры значения несколько иным образом. В частности, эта команда дает возможность загружать значение из двух слов в верхние байты указанного регистра, не влияя при этом на два нижних слова. Например, мы можем применить ее для загрузки адреса в регистр:

```
MOV R1, #0x0008
MOVT R1, #0x3F20
```

В `R1` окажется значение `0x3F200008`.

Как правило, команда `MOVT` используется для загрузки значения, которое нельзя было бы указать в прямой константе. Например, если ассемблер выдает сообщение об ошибке:

```
Error: invalid constant (xxxxxxx) after fixtup.
```

где `xxxxxxx` — это указанное вами значение. Здесь ассемблер пытается решить проблему, но у него не получается, поэтому он и выдает сообщение об ошибке! Таким образом, вам нужно будет использовать комбинацию `MOV/MOVT`.

12. Умные числа

В *главе 6* мы рассмотрели две базовые команды умножения: `MUL` и `MLA`. Это первые команды умножения, появившиеся в ARM. Начиная с версии 3, в ARM появились дополнительные команды для работы со знаковыми и беззнаковыми числами длиной до 64 битов. Ряд наиболее полезных команд мы рассмотрим в этой книге. Некоторые из них применяются в особых случаях и в более сложных задачах — например, в цифровой обработке.

Длинное умножение

Команды `SMULL` и `UMULL` выполняют знаковое и беззнаковое умножение с использованием двух регистров, содержащих 32-битные операнды, и дают 64-битный результат, который хранится в двух регистрах назначения. Их синтаксис:

```
SMULL (<Суффикс>) <Нижний регистр>, <Верхний регистр>, <Оп1>, <Оп2>
UMULL (<Суффикс>) <Нижний регистр>, <Верхний регистр>, <Оп1>, <Оп2>
```

При умножении со знаком предполагается, что значения, передаваемые в *операнде1* и *операнде2*, представлены в дополнительном коде. В этих командах нельзя использовать регистр счетчика команд (Program Counter, PC) и следует избегать регистра указателя стека (Stack Pointer, SP), поскольку он не поддерживается в некоторых более поздних чипах ARM (хотя на Raspberry Pi работает). Само собой разумеется, что регистры назначения должны быть разными.

В следующем примере мы получим 64-битные числа из произведения двух беззнаковых 32-битных чисел, находящихся в регистрах R1 и R2. Результат попадет в пару регистров R3 и R4: младшее слово — в R3, а старшее — в R4:

```
UMULL R3, R4, R1, R2
```

Чтобы вы лучше поняли, как новые команды позволяют сократить объем кода, выполним в *программе 12.1* такое же умножение, но с помощью базовой команды `MUL`. Как и в приведенном только что примере, предполагается, что в R1 и R2 лежат два беззнаковых числа, а результат помещается в пару: R3 (младшее слово) и R4 (старшее слово). Значения в R1 и R2 по окончании выполнения будут не определены. Также используется дополнительный регистр R5.

ПРОГРАММА 12.1. Долгое и нудное умножение

```

/* Умножение без команды UMULL */
/* mult: весь этот код можно заменить одной командой! */

@ R1 = 32-битное число без знака 1 (младшее)
@ R2 = 32-битное число без знака 2 (старшее)

@ Результат:
@ R3 = Результат (младшая часть произведения)
@ R4 = Результат (старшая часть произведения)
@ R1 = Не определено, R2 = Не определено, R5 = Не определено

.global _start
_start:
    MOV R1, #0xF0000002      @ ну, поехали...
    MOV R2, #0x2            @ [R3, R4] = R1 * R2
mult:
    MOVS R4, R1, LSR #16    @ R4 - старшие 16 бит R1
    BIC R1, R1, R4, LSL #16 @ R1 - младшие 16 бит R1
    MOV R5, R2, LSR #16    @ R5 - старшие 16 бит R2
    BIC R2, R2, R5, LSL #16 @ R2 - младшие 16 бит R2

    MUL R3, R1, R2          @ умножение младших частей
    MUL R2, R4, R2          @ первое промежуточное умножение
    MUL R1, R5, R1          @ второе промежуточное умножение
    MULNE R4, R5, R4        @ умножение старших частей

    ADDS R1, R1, R2         @ прибавление первого
                            @ промежуточного результата
    ADDCS R4, R4, #0x10000  @ добавляем перенос к старшей
                            @ части результата
    ADDS R3, R3, R1, LSL #16 @ прибавление еще одного
                            @ промежуточного результата
    ADC R4, R4, R1, LSR #16 @ складываем младшую и старшую части

    MOV R7, #1             @ выход из системного вызова
    SWI 0

```

Команда `ADDS` после условного умножения `MULNE` используется вместо `MLA`, т. к. нам нужно сохранить флаг переноса для будущего сложения `ADDCS`.

Если вам непонятен этот пример, попробуйте просчитать его от руки или проработать с помощью `GDB` после прочтения главы 14, где вы научитесь пошагово выполнять код и просматривать содержимое регистров.

Умножение с накоплением

Команды SMLAL и UMAL являются эквивалентами команды MLA со знаком и без знака. Как и в предыдущих командах, значения со знаком или без знака, указанные в *операнде1* и *операнде2*, перемножаются, но теперь результат добавляется к значению, уже находящемуся в регистрах *destLo* и *destHi*:

```
SMLALS R1, R2, R5, R6
```

Существует также несколько интересных вариантов команды, которые применяются только при умножении со знаком.

Команда SMLAXY выполняет умножение с накоплением, принимает 16-битные операнды и 32-битный накопитель. Ее синтаксис весьма интересен:

```
SMLA<x><y> <Место назначения>, <Op1>, <Op2>, <Op3>
```

Здесь вместо *x* и *y* подставляются буквы *v* или *t*, которые обозначают Bottom и Top (низ и верх), т. е. два нижних или верхних байта *операнда1* и *операнда2* соответственно. *Операнд3* содержит значение, которое прибавляется к результату умножения *операнда1* и *операнда2*.

Например:

```
SMLAVTCC R0, R1, R2, R3
```

В этом случае, если флаг переноса сброшен, младшее полуслово регистра *R1* умножается на верхнее полуслово регистра *R2*. Результат будет добавлен к значению регистра *R3* и сохранится в *R0*.

Команда SMLAWY (знаковое умножение в ширину) работает схожим образом, но теперь два верхних или два нижних байта *операнда2* умножаются на *операнд1*. Старшие 32 бита результата (который может иметь длину до 48 битов) помещаются в регистр назначения. Получается умножение 16-битного числа на 32-битное с накоплением. Полный синтаксис команды:

```
SMLAW <y> <Место назначения>, <Операнд1>, <Операнд2>, <Операнд3>
```

Например:

```
SMLAWB R0, R5, R6, R7
```

В этом случае нижнее полуслово регистра *R6* умножается на полное слово в регистре *R5*, затем значение в *R7* прибавляется к результату, который помещается в *R0*.

Команды SMUAD и SMUSD работают с 16-битными значениями и выполняют знаковое умножение со сложением и знаковое умножение с вычитанием, позволяя при этом менять местами половины операндов. Их синтаксис:

```
SMUAD <X> (<Суффикс>) <Место назначения>, <Операнд1>, <Операнд2>
```

```
SMUSD <X> (<Суффикс>) <Место назначения>, <Операнд1>, <Операнд2>
```

Если в инструкцию включена буква *x*, будет выполнен обмен значениями в старшем и младшем полу словах *операнда2*. Если буквы *x* нет, обмен выполняться не будет. Затем эта команда перемножает содержимое двух нижних полу слов *операн-*

да1 и операнда2 и сохраняет результат, затем перемножает содержимое двух верхних полуслов операндов и сохраняет результат.

При выполнении команды SMUAD (двойное знаковое 16-битное умножение со сложением) полученные произведения складываются, а результат помещается в регистр назначения. При выполнении команды SMUSD (двойное знаковое 16-битное умножение с вычитанием) второе произведение (верхнее полуслово) вычитается из первого произведения.

Примеры:

```
SMUADXEQ R5, R6, R7
SMUSD    R5, R7, R9
```

Деление и остаток

В главе 6 мы говорили, что в ранних версиях Raspberry Pi, а именно в версиях А, В и Zero, не было поддержки команд деления. В более поздних чипах ARM появились команды SDIV и UDIV. В программе 12.2 показано, как можно выполнить деление с использованием двух 32-битных значений без специальных команд деления.

Предполагается, что делимое изначально находится в R1, а делитель — в R2. После выполнения команды в R3 попадет частное, в R1 — остаток, а в R2 останется исходный делитель. Проверок на предмет того, равен ли делитель нулю, не выполняется, но мы можем сделать это и сами. Помните, что команды SDIV и UDIV не вычисляют остаток, поэтому приведенная далее программа пригодится, если он вам понадобится.

По завершении программы в R0 окажется частное, поэтому команда консоли

```
echo $?
```

выведет 5. Остаток от деления (11) окажется в R1.

ПРОГРАММА 12.2. Долгое нудное деление

```
/* Деление без специальной команды */
/* В результате получим частное и остаток */
@ имеем: R1 = делимое, R2 = делитель
@ получаем: R3 = частное, R1 = остаток, R2 = все еще делитель
.global _start
_start:
    MOV R1, #111          @ деление 111/20
    MOV R2, #20
    MOV R4, R2           @ сохранение делителя
    CMP R4, R1, LSR #1

Div1: MOVLS R4, R4, LSL #1 @ деление надвое, пока
    CMP R4, R1, LSR #1   @ R4 x 2 больше делителя
    BLS Div1
    MOV R3, #0           @ инициализация частного
```

Div2: CMP R1, R4	@ можно ли вычесть R4?
SUBCS R1, R1, R4	@ если да, то вычитание
ADC R3, R3, R3	@ удваивание частного
bit	
MOV R4, R4, LSR #1	@ деление R4 надвое
CMP R4, R2	@ цикл работает, пока...
BHS Div2	@ ..делитель не исчезнет
MOV R0, R3	@ помещение частного в R0
MOV R7, #1	@ выход из системного вызова
SWI 0	

Умное умножение

В предыдущей главе мы рассмотрели простое умножение. Теперь, зная кое-что о сдвигах и битовых операторах, мы можем найти более простые способы выполнить умножение. В примерах, приведенных далее, мы используем R0, но вообще можно использовать любой регистр.

Если вы хотите выполнить умножение на два, вам следует использовать команду LSL напрямую:

```
MOV R0, R0, LSL #n
```

Где n — константа. Если вместо n подставим 4, получим:

```
R0 = R0 x 2 x 2 x 2 x 2
```

По сути, выполняется умножение на 2^n .

Чтобы умножить число на $(2^n) + 1$ — например: 3, 5, 9, 17 и т. д., надо написать следующее:

```
ADD R0, R0, R0, LSL #n
```

где n — произвольное значение.

Чтобы, наоборот, выполнить умножение на $(2^n) - 1$ — например: 3, 7, 15 и т. д., напишем так:

```
RSB R0, R0, R0, LSL #n
```

где n — произвольное значение.

Чтобы умножить число на 6, сначала нужно умножить его на три, а затем на два:

```
ADD R0, R0, R0, LSL #1
MOV R0, R0, LSL #1
```

Это только начало...

Мы рассмотрели лишь пару сложных арифметических команд, которые есть на чипе ARM в Raspberry Pi. На самом деле их много, гораздо больше, чем мы знаем, плюс существует еще несколько вариантов тех команд, которые мы уже рассмотрели. Лучший способ изучить их — написать базовые программы, которые будут загружать числа в регистры перед выполнением определенной команды, а затем пошагово выполнять программу с помощью GDB, исследуя значения в регистрах на каждом шаге.

13. Программный счетчик R15

Регистр 15 (Program Counter, PC) — это программный счетчик (регистр счетчика команд), и это весьма важный регистр. Если вы не будете относиться к нему с должным пиететом, ваша программа может дать сбой. При этом Raspberry Pi, скорее всего, «зависнет» и не сможет ничего сделать, пока вы не перезагрузите питание платы. Этот процесс не только отнимает много времени, но еще и изрядно раздражает. Он все равно будет происходить время от времени, но лучше свести такие случаи к минимуму!

Назначение регистра R15 весьма просто. Он отслеживает, в какой именно точке в текущий момент находится программа. В регистре хранятся 32-битные адреса физической памяти. По сути, в регистре PC содержится адрес следующей команды. Поэтому, если вы вдруг положите в него постороннее число, программа может «сломаться».

Регистр R15 может использоваться в командах разными способами. Вы можете задействовать его в командах обработки данных в роли *операнда1* или *операнда2*. Например:

```
ADD R0, R15, #8
```

Здесь R15 выступает в роли *операнда1*. Эта команда прибавляет 8 к значению (адресу) в R15 и сохраняет результат в R0.

Вот еще пример:

```
SUB R0, R9, R15
```

Здесь *операнд2*, т. е. значение в R15, вычитается из R9, а результат сохраняется в R0.

R15 также может использоваться в качестве регистра назначения. В этом случае обязательно следить, чтобы загружаемое в регистр значение подходило для счетчика программ, поскольку программа тут же попытается выполнить команду с этим адресом:

```
MOV R15, R14
```

Эта команда помещает значение из R14 в R15. Поскольку R14 является регистром ссылок, подобный метод позволяет вернуться в основную программу ранее вы-

званной процедуры. Как правило, подобные команды используются для передачи управления обратно в точку, откуда был вызван машинный код.

Конвейерная обработка

Важно понимать, как именно ARM выбирает, декодирует и выполняет команды. Конвейер команд — это встроенный механизм ARM, работа которого определяет скорость выполнения программы. Все дело в том, что, когда дело доходит до выполнения машинного кода, ARM почти одновременно делает три дела: выборку, декодирование и выполнение команды. Поскольку эти операции нельзя выполнять с одной и той же командой, ARM обрабатывает сразу три команды: одну выполняет, вторую декодирует, третью извлекает. После выполнения команды вся очередь сдвигается. Команда, извлеченная ранее, переходит к этапу декодирования, а та, которая декодировалась, отправляется на выполнение. Этот непрерывный конвейер проиллюстрирован на рис. 13.1.

	Выборка	Декодирование	Выполнение
Цикл 1	Op1	Пусто	Пусто
Цикл 2	Op2	Op1	Пусто
Цикл 3	Op3	Op2	Op1
Цикл 4	Op4	Op3	Op2

Рис. 13.1. Циклы выборки, декодирования и выполнения в процессоре ARM

В начале работы ARM за три цикла заполняет конвейер. После выполнения команды она отбрасывается, поскольку следующая команда занимает ее место. Именно благодаря этому параллельному процессу процессор ARM быстро обрабатывает данные. В процессе декодирования ARM определяет, какие регистры следует использовать для той или иной команды.

На рис. 13.1 в цикле 4 регистр PC содержит адрес Op4 — следующей в очереди на выборку команды. На рис. 13.2 показано, где в каждом цикле в регистре PC находится тот или иной элемент.

Этот трехступенчатый конвейер является «фишкой» микросхемы ARM. На самом деле современные процессоры ARM устроены еще сложнее, и у чипа ARM в ва-

Регистр	Действие
PC	Следующая выполняемая команда
PC-4	Декодируемая команда
PC-8	Выполняемая в текущий момент команда
PC-12	Предыдущая команда

Рис. 13.2. Так PC пропускает через себя команды

шей плате Raspberry Pi работает конвейер, на котором находится не менее восьми операций одновременно. Но для понимания излагаемого материала и рассматриваемых концепций подойдет и исходная модель конвейера (хотя мы вернемся к этому вопросу в *главе 27*). Всегда нужно учитывать влияние конвейерной обработки на работу программы, иначе при определенных обстоятельствах ваша программа может начать работать не так, как вы ожидали. Обратите внимание на вот такую команду:

```
MOV R15, R15
```

В результате выполнения этой команды следующая команда будет пропущена. Дело в том, что адрес, к которому осуществляется доступ в регистре PC, на два слова (восемь байтов) больше, чем адрес команды MOV. Когда операция записывается обратно в PC, выполнение возобновляется на пару слов (команд) дальше, и одна команда пропускается.

Помните, что адрес, хранящийся в PC, всегда на восемь байтов больше, чем адрес выполняемой в текущий момент команды.

Расчет ветвей

О ветвях мы говорили в *главе 10*. Давайте посмотрим, как они обрабатываются в счетчике программ.

Ветви в коде обычно выглядят так:

```
BAL Метка
```

В нашем случае *метка* — это именованная позиция в программе на языке ассемблера. Помните, что расстояние «прыжка» ветви ограничено и составляет ± 32 Мбайт, поскольку именно таков максимальный адрес, который может быть представлен в пространстве, выделенном для позиции метки. При этом, как и отмечалось ранее, хранится не абсолютный адрес метки, а ее смещение от текущей позиции. Когда ARM встречает инструкцию ветвления, он обрабатывает следующее значение как положительный (вперед) или отрицательный (назад) сдвиг в регистре PC.

Команды кодируются таким образом, что значение ветви представляет собой 24-битное смещение со знаком в дополнительном коде. Смещение слова сдвигается влево на два разряда (бита), формируя смещение в байтах. Оно и добавляется к PC. Взгляните на пример, показанный на рис. 13.3.

Первый столбец — это адрес команды. У нас есть две метки с адресами: $0x1C30$ и $0x2C30$. Первая команда

```
BEQ zero
```

находится по адресу $0x0100$, а вторая команда

```
BL notzero
```

находится по адресу $0x0120$.

Адрес	Метка	Команда
0x0100		BEQ zero
		...
		...
0x0120		BL notzero
		...
0x1C30	zero:	<команды>
		...
		...
0x2C30	notzero	<команды>

Рис. 13.3. Расчет ветвей

Из-за особенностей конвейерной обработки при выполнении команды `BEQ zero` извлекаемая команда окажется на две команды, или на восемь байтов, позже. Тогда получаем следующее байтовое смещение для команды `BEQ`:

$$0x1C30 - 0x0100 - 8 = 0x1B28$$

Итак, смещение слова для нулевой `BEQ zero`:

$$0x1B28/4 = 0x6CA$$

Для команды `BL notzero` расчет выглядит следующим образом:

$$0x23C0 - 0x0120 - 8 = 0x2B08$$

$$0x2B08/4 = 0xAC2$$

Ветви в «обратную» сторону работают аналогичным образом, и, опять же, следует учитывать особенности конвейера.

Используя относительные значения, или значения смещения для создания ответвлений, вы можете писать программы, которые можно перемещать целиком. То есть их можно загружать и запускать из любой части памяти. А если вы жестко запрограммируете адрес, код окажется привязан к месту.

14. Отладка с использованием GDB

В ОС Raspberry Pi «из коробки» имеется полный набор инструментов для отладки — GDB. Он окажется невероятно полезен, когда вы станете разбираться в программе и пытаться понять, почему что-то работает не по плану. Кроме того, он позволяет в целом узнать побольше о работе программы.

Когда ваша программа начинает работать некорректно, обычно происходит одно из двух. Вариант первый: программа выдает неправильный результат. Вариант второй: результат не выводится вообще, и система зависает, требуя полной перезагрузки. А еще оба эти варианта могут возникать одновременно!

В случае с неправильным результатом велика вероятность того, что неверно задана константа или адрес. Утешает, что программа в целом работает и в ней нет логических ошибок или ошибок ветвления. Поэтому нужно попытаться отследить, где происходит ошибка. Тип возвращаемого результата поможет вам понять, в чем проблема, после чего вам нужно будет изучить его и сделать несколько собственных выводов. Например, если вы получили результат на единицу больше, чем ожидалось (а «единица» не обязательно является числом), возможно, цикл выполняется чуть больше раз, чем следовало бы. Тогда нужно подкорректировать счетчик цикла или команды условного перехода. Было бы полезно точно знать, какое значение принимает счетчик цикла по ходу работы.

Когда все зависло...

Если плата зависла, все может быть немного сложнее. Возможно, программа попала в бесконечный цикл. Счетчик цикла не меняет значение, поэтому цикл бесконечно продолжает работу, пока питание не отключится. Возможно, вы неправильно управляли стеком или повредили счетчик команд.

Поиск подобных ошибок станет для вас повседневной задачей во время занятий программированием. Это неотъемлемая часть ремесла. Поэтому всегда полезно разрабатывать свои программы по частям, небольшими разделами или функциями. У каждой функции свое назначение, и каждую можно тестировать отдельно.

Если оказывается, что программа работает некорректно или зависает, важнее всего сначала определить, в какой момент это происходит. Лучший способ найти ошиб-

ку — своими глазами увидеть, до какого момента машинный код работает правильно, после чего выходит из строя. Это позволит вам, по крайней мере, сузить область поиска. Например, если код будет выводить что-то на экран, вы можете получить хоть какое-то представление о том, что происходит в коде и что вы пропустили.

Если у вас возникли проблемы с кодом и вы не можете понять, в какой именно его части возникает проблема, добавьте в код команды для вывода чего-нибудь на экран, чтобы было понятно, где вы находитесь, это поможет понять, где именно находится проблема.

Допустим, программа разделена на пять частей. Вы могли бы разместить соответствующий вызов подпрограммы `_write` в начале каждой части, как показано на рис. 14.1.

```
.area1
BL _write    @ Print A
...
.area2
BL _write    @ Print B
...
.area3
BL _write    @ Print C
...
.area4
BL _write    @ Print D
...
.area5
BL _write    @ Print E
...
```

Рис. 14.1. Поиск проблем с помощью функции стиля `_write`

Теперь после запуска программы при достижении каждой области на экране будет печататься буква. Допустим, получилось вот это:

ABC

после чего программа зависла. Это говорит о том, что проблема находится где-то в области 3, т. е. до буквы «D» дело не дошло. С этого момента вы можете сосредоточиться именно на этой части кода. Добавив дополнительные выходы на экран, вы можете вывести еще какие-нибудь буквы или цифры внутри области 3. Это сузит диапазон поиска и позволит вам начать искать ошибку в нужной области, рассмотрев ее более внимательно.

Сборка с GDB

GDB — это отладчик для проектов GNU. Он установлен в ОС Raspberry Pi «из коробки» и запускается из командной строки. GDB содержит великое множество инструментов, которые позволяют анализировать программы машинного кода разны-

ми способами из изолированной среды. Он способен работать на самых разных уровнях, и можно, не кривя душой, сказать, что у него есть команды почти на все случаи жизни. Его тоже можно настроить. Как и большинство программ GNU, GDB снабжен подробнейшей документацией, которую можно найти в Интернете. В этой главе мы рассмотрим несколько практических примеров, а в качестве подопытного кролика воспользуемся *программой 10.1*.

Чтобы можно было задействовать GDB, базовая программа должна быть собрана с использованием дополнительной директивы, т. к. требуется генерация дополнительной информации, которая может быть полезна для GDB:

```
as -g -o prog10a.o prog10a.s
ld -o prog10a prog10a.o
```

Параметр `-g` генерирует для отладчика дополнительные данные. Для вас эти изменения не заметны. После этого вы можете запустить GDB следующим образом:

```
gdb <имя файла>
```

где *имя файла* — это имя собранного файла, который нужно проанализировать. Команда:

```
gdb prog10a
```

запустит отладчик и загрузит информацию о файле `prog10a`. Если вы забыли указать имя файла, вы можете использовать команду `file` в приглашении GDB:

```
file prog10a
```

Теперь введем команду:

```
list
```

которая выдаст результат, показанный в листинге 14.1. (Возможно, вам потребуется нажать клавишу `<Return>`, чтобы продолжить ввод, — там, где будет стоять приглашение: **(gdb)**.) Цифры в начале строк — это их номера. Вы бы видели их, если бы при редактировании исходного кода у вас была включена соответствующая опция. В GDB вы можете использовать эти цифры со многими имеющимися у вас полезными командами.

ВАЖНО!

Номера строк, которые будут выведены у вас, могут отличаться от показанных здесь, равно как и адреса памяти в вашей системе тоже могут быть другими. В этом нет ничего страшного, вам лишь нужно учитывать наличие таких различий.

Листинг 14.1. Вывод загруженного файла в GDB

```
$ gdb prog10a
```

```
GNU gdb (Raspbian 8.2.1-2) 8.2.1
```

```
Copyright (C) 2018 Free Software Foundation, Inc.
```

```
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
```

```
This is free software: you are free to change and redistribute it.
```

```
There is NO WARRANTY, to the extent permitted by law.
```

```
Type "show copying" and "show warranty" for details.
```

This GDB was configured as "arm-linux-gnueabi".
 Type "show configuration" for configuration details.
 For bug reporting instructions, please see:
 <<http://www.gnu.org/software/gdb/bugs/>>.
 Find the GDB manual and other documentation resources online at:
 <<http://www.gnu.org/software/gdb/documentation/>>.

For help, type "help".
 Type "apropos word" to search for commands related to "word"...
 Reading symbols from prog10a...done.

```
(gdb) list
1      /**** Преобразование числа в двоичное и его вывод */
2      /*
3      /* Регистры: R6 = номер, R8 = сохранить, R9 = маска */
4      /* R7 необходим для системного вызова, R1 указывает на строку */
5
6      .global _start
7
8      _start:
9          MOV R6, #251           @ помещение числа в R6
10         MOV R10, #1           @ настройка маски
(gdb)
11         MOV R9, R10, LSL #31
12         LDR R1, = string      @ указатель на метку string
13
14     _bits:
15         TST R6, R9             @ проверка числа по маске
16         MOVEQ R0, #48         @ ASCII-код символа '0'
17         MOVNE R0, #49         @ ASCII-код символа '1'
18         STR R0, [R1]          @ сохранение 1 в строке
19         MOV R8, R6             @ сохранение числа
20         BL _write             @ вывод на экран
(gdb)
21         MOV R6, R8             @ загрузка числа
22         MOVS R9, R9, LSR #1    @ смещение битов маски
23         BNE _bits
24
25     _exit:
26         MOV R7, #1
27         SWI 0
28
29     _write:
30         MOV R0, #1
(gdb)
31         MOV R2, #1
32         MOV R7, #4
33         SWI 0
34         BX LR
35
```

```

36      .data
37      string: .ascii " "
(gdb)
Line number 38 out of range; prog10a.s has 37 lines.
(gdb)

```

Дизассемблер

Дизассемблер работает противоположно ассемблеру. Он принимает значения, хранящиеся в памяти, и преобразует их в код на языке ассемблера. Для примера введите в командной строке GDB:

```
disassemble _start
```

Вы получите результат, похожий на вывод в листинге 14.2.

Листинг 14.2. Дизассемблирование функции в GDB

```

Dump of assembler code for function _start:

0x00010074 <+0>:  mov    r6, #251      ;0xfb
0x00010078 <+4>:  mov    r10, #1
0x0001007c <+8>:  lsl    r9, r10, #31
0x00010080 <+12>: ldr    r1, [pc, #60] ;0x100c4 <_write+20>
End of assembler dump.
(gdb)

```

При сборке исходного кода параметр `-g` создает список меток или функций, определенных в исходном коде, что позволяет нам обращаться к ним напрямую из GDB. Первый столбец в сгенерированном листинге — это адрес в памяти, от которого собирается код (этот адрес у вас может быть иным). Во втором столбце в угловых скобках указано количество байтов от начала функции.

Обратите внимание, что последняя строка в этом листинге отличается от того, что было в исходном коде. Дизассемблер преобразовал оригинал:

```
LDR R1, = string
```

в абсолютный адрес. Получилось:

```
ldr r1, [pc, #60]
```

Команда загружает в `R1` адрес, который на 60 опережает текущий адрес в `PC` (`R1 = PC + 60`). То есть здесь учитывается конвейерная обработка. Фактический адрес равен `0x100c4` и указан после точки с запятой в конце строки, как и метка, на которую он ссылается!

Вы также можете дизассемблировать область памяти, указав начальный и конечный адреса. Используя параметр `/r` в начале, вы можете включить шестнадцатеричные коды операций и операнды. Например, код

```
disassemble /r _bits
```

выдаст результат, показанный в листинге 14.3.

Листинг 14.3. Использование параметра /r

```

Dump of assembler code for function _bits:
0x00010084 <+0>:  09 00 16 e1  tst    r6, r9
0x00010088 <+4>:  30 00 a0 03  moveq  r0, #48
0x0001008c <+8>:  31 00 a0 13  movne  r0, #49
0x00010090 <+12>: 00 00 81 e5  str    r0, [r1]
0x00010094 <+16>: 06 80 a0 e1  mov    r8, r6
0x00010098 <+20>: 04 00 00 eb  bl     0x100b0 <_write>
0x0001009c <+24>: 08 60 a0 e1  mov    r6, r8
0x000100a0 <+28>: a9 90 b0 e1  lsrs  r9, r9, #1
0x000100a4 <+32>: f6 ff ff 1a  bne   0x10084 <_bits>
End of assembler dump.
(gdb)

```

Третий набор цифр — это коды операций и операнды для каждой из команд. Если вы посмотрите, например, на строку, начинающуюся с адреса 0x00010098, то увидите, что было вычислено смещение для команды `bl` (0xb0). Во время работы текущее положение регистра `PC` отображается символом `\=>` слева от одного из адресов.

В табл. 14.1 приведены некоторые наиболее распространенные варианты дизассемблирования в GDB.

Таблица 14.1. Общие команды дизассемблирования

Команда	Описание
<code>disas <функция></code>	Дизассемблирование заданной функции. Пример: <code>disas _start</code>
<code>disas <адрес1>, <адрес2></code>	Дизассемблирование в диапазоне адресов с <code><адрес1></code> по <code><адрес2></code> . Пример: <code>disas 0x8084, 0x08A4</code>
<code>disas <строка></code>	Дизассемблирование по номеру строки. Пример: <code>disas 19</code>
<code>/r</code>	Включение 16-ричного кода операции в вывод. Пример: <code>disas /r _start</code>
<code>/m</code>	Включение информации в вывод. Пример: <code>disas /m _write</code>
<code>b <строка></code>	Точка останова в выбранной строке. Номер строки может быть также задан с помощью элемента <code><метка></code>
<code>r</code>	Запуск
<code>s</code>	Переход к следующей команде
<code>c</code>	Продолжить выполнение программы
<code>q</code>	Выход из GDB
<code>i r</code>	Вывод содержимого всех регистров
<code>l b</code>	Вывод списка точек останова
<code>delete <число></code>	Удаление точки останова с заданным номером

Таблица 14.1 (окончание)

Команда	Описание
<code>x/Nfu <выражение></code>	Вывод содержимого памяти по заданному выражению. <i>N</i> — число (Number), <i>f</i> — формат (<i>x</i> — шестнадцатеричный, <i>d</i> — десятичный, <i>t</i> — двоичный), <i>u</i> — размер блока (<i>b</i> — байты, <i>h</i> — полуслова, <i>w</i> — слова)
<code><Ctrl>+<C></code>	Прервать выполнение программы

Точки останова

Самое главное средство отладки в вашем арсенале — это, без сомнения, использование точек останова и возможность пошагового выполнения команд, что позволяет вам наблюдать за вашей программой в действии.

Точки останова — это временные знаки остановки в программе машинного кода, которые GDB позволяет вам размещать в любом месте программы. Когда вы запускаете свою программу из GDB, она будет останавливаться на каждой точке останова, сохраняя при этом все регистры. Вставив одну или несколько точек останова в программу машинного кода, вы сможете в нужных местах остановиться и посмотреть на содержимое регистров и флагов в этот момент. Такой инструмент может быть очень полезен, когда программа работает неверно. Изучив содержимое регистров и флагов, вы сможете сузить круг поиска и устранить виновника проблемы. К тому же, как вы уже догадались, это также отличный способ изучить работу каждой команды.

Точки останова можно устанавливать с помощью меток или номеров строк, используя команду `b` в приглашении GDB:

```
b _bits
```

В первом случае точка останова появится на адресе метки `_bits`. Визуально это будет выглядеть так:

```
Breakpoint 1 at 0x10084: file prog10a.s, line 15
```

Давайте установим вторую точку останова сразу после двух условных команд `mov`. Из текста программы мы видим, что это строка 18:

```
b 18
```

Получим:

```
Breakpoint 2 at 0x10090: file prog10a.s, line 18
```

Команда

```
info b
```

выведет список всех установленных на текущий момент точек останова:

```
1 breakpoint keep y          0x00010084 prog10a.s:15
2 breakpoint keep y          0x00010090 prog10a.s:18
```

Как можно видеть, здесь показаны две наши точки останова.

Удалить точки останова так же просто. Команда

```
delete 2
```

удалит точку останова 2.

Вы можете выполнять программы и заставлять их останавливаться на определенных точках останова. Предположим, у нас есть две наши точки останова. Введите:

```
run
```

В приглашении GDB выдаст:

```
Breakpoint 1, _bits () at prog10a.s:15
15 TST R6, R9      @ проверка числа по маске
```

Программа запустилась, но остановилась перед указанной командой в ожидании. Теперь можно получить дамп всего содержимого регистра, набрав:

```
info r
```

Получим список, подобный показанному в листинге 14.4.

Листинг 14.4. Дамп регистра после остановки на точке 1

```
(gdb) info r
r0                0x0                0
r1                0x200c8             131272
r2                0x0                0
r3                0x0                0
r4                0x0                0
r5                0x0                0
r6                0xfb              251
r7                0x0                0
r8                0x0                0
r9                0x80000000         2147483648
r10               0x1                0
r11               0x0                0
r12               0x0                0
sp                0x7efff3a0         0x7efff3a0
lr                0x0                0
pc                0x10084           0x10084 <_bits>
cpsr              0x10              0
fpscr             0x0                16
(gdb)
```

К этому моменту будет выполнен код, указанный в строке 14, что станет видно по содержимому регистров. Чтобы перейти к следующей точке останова, введите:

```
continue
```

а затем снова посмотрите содержимое регистров (листинг 14.5).

Листинг 14.5. Данные в регистрах после остановки на второй точке

```

Breakpoint 2, _bits () at prog10a.s:18
18          STR R0, [R1]      @ Store 1 in string
(gdb) info r
r0          0x30             48
r1          0x200c8         131272
r2          0x0             0
r3          0x0             0
r4          0x0             0
r5          0x0             0
r6          0xfb            251
r7          0x0             0
r8          0x0             0
r9          0x80000000      2147483648
r10         0x1             1
r11         0x0             0
r12         0x0             0
sp          0x7efff3a0      0x7efff3a0
lr          0x0             0
pc          0x10090         0x10090 <_bits+12>
cpsr       0x40000010      1073741840
fpscr      0x0             0
(gdb)

```

Здесь мы видим, что в регистр R0 записан ASCII-код символа «0», поэтому дальше выполнялась команда `MOVEQ`. Это можно понять по регистру состояния, в котором поднят флаг нуля. Обратите внимание, что PC показывает, в какой точке программы мы находимся.

Попробуйте установить третью точку останова в строке 25 (это можно сделать, пока программа запущена) и перейдите к ней. Вывод содержимого регистра покажет результат, приведенный в листинге 14.6.

Листинг 14.6. Данные в регистрах после третьей точки останова

```

Breakpoint 3, _bits () at prog10a.s:23
23          BNE _bits
(gdb) info r
r0          0x1             1
r1          0x200c8         131272
r2          0x0             0
r3          0x0             0
r4          0x0             0
r5          0x0             0
r6          0xfb            251
r7          0x4             4

```

```

r8          0xfb          251
r9          0x40000000     1073741824
r10         0x1           1
r11         0x0           0
r12         0x0           0
sp          0x7efff3a0    0x7efff3a0
lr          0x1009c      65692
pc          0x100a4      0x100a4 <_bits+32>
cpsr       0x10         16
fpscr      0x0          0
(gdb)

```

Здесь в R9 записывается новое значение маски. Кроме того, обратите внимание, что в регистре ссылок хранится адрес. Посмотрите на оригинальный код программы, чтобы понять, откуда взялись эти новые значения.

Вы можете выполнять свою программу в GDB построчно, просто нажимая клавишу <s>. Такое выполнение называется *пошаговым*. Попробуйте этот режим и проследите за выполнением программы через функцию `_write`. Вы также можете вывести содержимое регистра в любой момент.

GDB полностью интерактивен, и в табл. 14.2 показано еще несколько популярных команд, с которыми можно поэкспериментировать.

Таблица 14.2. Общие команды для работы с точками останова

Команда	Действие
<code>break <функция></code>	Установка точки останова на выбранной функции. Пример: <code>b_start</code>
<code>break <номер строки></code>	Установка точки останова на выбранной строке. Пример: <code>b 23</code>
<code>break <смещение></code>	Установка точки останова на строке, отстоящей от текущей на определенной число. Пример: <code>b+5</code>
<code>break *<адрес></code>	Установка точки останова на выбранном адресе. Пример: <code>b *0x8074</code>
<code>info break</code>	Вывод информации обо всех установленных точках останова
<code>delete <номер></code>	Удаление заданных точек останова. Если число не задано, удаляются все. Пример: <code>delete 2</code>

СОВЕТ: МЕТКИ ТОЧЕК ОСТАНОВА

Вместо того, чтобы держать в уме номера строк, в которых нужны точки останова, вы всегда можете разработать систему меток и использовать их в своих файлах с кодом. Обычно это должно происходить после загрузки данных из памяти или регистра, чтобы вы могли проверить правильность данных сразу после загрузки, после обработки или в конце программы.

Дамп памяти

Вы можете просматривать память напрямую: и разделы с кодом, и разделы с данными. Последнее особенно полезно, т. к. позволяет увидеть, как работа вашей программы влияет на данные. Если вы запуском своей программы очистили память или заполнили ее нулями, вы точно знаете, что будет лежать в памяти после выполнения программы.

Команда `x` выполняет вывод дампа памяти в различных форматах. Чтобы получить шестнадцатеричный дамп памяти самой программы, введите в приглашении GDB следующую команду:

```
x/22xw _start
```

Результат будет аналогичен приведенному в листинге 14.7.

Листинг 14.7. Шестнадцатеричный дамп памяти в GDB

```
(gdb) x/22xw _start
```

```
0x10074 <_start>:  0xe3a060fb  0xe3a0a001  0xe1a09f8a  0xe59f103c
0x10084 <_bits>:   0xe1160009  0x03a00030  0x13a00031  0xe5810000
0x10094 <_bits+16>: 0xe1a80006  0xeb000004  0xe1a06008  0xe1b090a9
0x100a4 <_bits+32>: 0x1afffff6  0xe3a07001  0xef000000  0xe3a00001
0x100b4 <_write+4>: 0xe3a02001  0xe3a07004  0xef000000  0xe12ffffe
0x100c4 <_write+20>: 0x000200c8  0x00154120
```

Синтаксис команды:

```
x/nfu <адрес>
```

Здесь символ `/` обозначает изменения значений по умолчанию, `f` — это шестнадцатеричный формат по умолчанию, `a` — размер блока. Значение `адрес` — это начальный адрес, с которого начинается выгрузка. В приведенном в листинге 14.8 примере `w` — это четырехбайтовое слово. Будут выведены 22 из них, начиная с `0x08`.

Можно использовать и другие размеры: `b` — байты, `h` — полуслова, `g` — гигантские слова (восемь байтов).

Когда вы указываете единицу измерения для `x`, она становится значением по умолчанию, пока вы снова не измените ее. При первом запуске GDB значение по умолчанию равняется `l`.

Команду `i` в сочетании с `x` можно использовать для дизассемблирования. Например, команда

```
x /13i 0x10084
```

дизассемблирует функцию `_bits`, как показано в листинге 14.8.

Листинг 14.8. Различные комбинации команд

```
(gdb) x/13i 0x10084

0x10084 <_bits>:      tst      r6, r9
0x10088 <_bits+4>:    moveq   r0, #48          ; 0x30
0x1008c <_bits+8>:    movne   r0, #49          ; 0x31
0x10090 <_bits+12>:   str     r0, [r1]
0x10094 <_bits+16>:   mov     r8, r9
0x10098 <_bits+20>:   bl     0x100b0 <_write>
0x1009c <_bits+24>:   mov     r8, r9
0x100a0 <_bits+28>:   lsrs   r9, r9, #1
=> 0x100a4 <_bits+32>: bne     0x10084 <_bits>
0x100a8 <-exit>:      mov     r1, #1
0x100ac <_exit+4>:    svc     0x00000000
0x100b0 <_write>:     mov     r0, #1
0x100b4 <_write+4>:  mov     r2, #1
```

Количество возможных вариантов использования команд почти бесконечно, и вам определенно следует распечатать себе копию руководства по GDB и держать ее под рукой. GDB и в самом деле является одним из тех инструментов, от которых вы всегда будете стремиться получить больше, а распечатанное руководство отлично подходит для заметок.

Наконец, набрав команду `quit`, вы выйдете из GDB и вернетесь в командную строку Raspberry Pi OS.

Сокращения

Большинство команд GDB можно писать в виде одной буквы:

```
info registers    >>  i r
continue          >>  c
step              >>  s
quit              >>  q
```

Параметры сборки GDB

Не стоит упускать возможность проработать пример длинного умножения из файла `prog12a` с использованием GDB. Построчное выполнение кода и вывод содержимого регистров дают немало информации. Такой анализ также поможет вам привыкнуть к методике работы с инструментом отладки.

Изменится и методика использования `make`-файла. Теперь для работы с отладочным кодом мы можем использовать `Make`, выполняя сборку с параметром `-g`. В *программе 14.1* приведен обновленный `make`-файл, который позволяет это сделать. Формат его немного изменился.

ПРОГРАММА 14.1. Гибкий make-файл для отладки кода
--

```

OBJX = prog10a
OBJJS = prog10a.o

ifdef GFLAG
    STATUS = -g
else
    STATUS =
endif

%.o : %.s
    as $(STATUS) $< -o $@

debugfile: $(OBJJS)
    ld -o $(OBJX) $(OBJJS)

gdbdebug: $(OBJX)
    gdb $(OBJX)

clean:
    rm -f *.o $(OBJJS)
    rm -f *.o $(OBJX)

```

Вам необходимо сохранить этот файл как make-файл, и главное тут — не перезаписать какой-нибудь из нужных вам файлов. В конечном итоге вы сами выработаете для себя методику работы, которая вам подходит.

Измените имена исходного make-файла в первых двух строках. Теперь вы можете собрать и связать его с параметром `-g` или без него. Важно сделать другое: запомнить «опцию принудительного выполнения» с помощью флага `-B`. Используйте, чтобы включить сведения об отладке, либо команду

```
make GFLAG=1 -B
```

либо

```
make GFLAG= -B
```

В обоих случаях вы увидите процесс сборки и связывания файла. Вы также можете сразу перейти в GDB, если нужно:

```
make gdbdebug
```

А удалить все файлы, кроме исходных, можно командой

```
make clean
```

15. Передача данных

В большинстве примеров, которые мы рассматривали до сих пор, все данные для команд берутся либо из регистра, либо из прямой константы или значения:

```
ADD R0, R1, R2
SUB R0, R1, #7
```

В имеющемся наборе регистров может храниться ограниченный объем информации, и, как правило, для выполнения операций с данными регистры нужно держать в чистоте. Как правило, данные создаются и хранятся в известных ячейках памяти. Это значит, нам необходимо управлять этими блоками памяти. Чтобы загружать и хранить данные в памяти, нужно знать два момента. Во-первых, адрес, где данные хранятся, а во-вторых, конечный пункт назначения, т. е. откуда они берутся и куда помещаются. В обоих случаях задействуются регистры, а указываемый адрес зависит от используемого режима адресации. В ARM есть три режима адресации:

- ◆ косвенная адресация;
- ◆ предварительно индексированная адресация;
- ◆ постиндексированная адресация.

Во всех трех случаях мы загружаем или сохраняем содержимое указанного регистра, но источник данных или место назначения при этом будут различными.

Директива *ADR*

В *главе 6* мы рассмотрели использование прямых констант и увидели, что, хотя инструкции `MOV` и `MVN` могут использоваться для загрузки констант в регистр, не любое значение может быть указано в виде константы. Причина этого в том, что их нельзя использовать для генерации всех доступных адресов памяти. Поэтому у ассемблера GCC есть метод загрузки любого 32-битного адреса. В простейшем виде он выглядит так:

```
ADR <Регистр>, <Метка>
```

Например:

```
ADR R0, datastart
```

Команда `ADR` похожа на команду `ARM`, но является *директивой*. Это часть ассемблера. Она выполняет много сложных действий, чтобы вычислить правильное значение. Когда ассемблер встречается эту директиву, он делает следующее:

1. Отмечает адрес, где собирается команда.
2. Записывает адрес метки.
3. Вычисляет смещение между двумя позициями в памяти.

Затем он использует эту информацию для другой команды — обычно `ADD` или `SUB`, чтобы восстановить местоположение адреса или метки, содержащей информацию.

Для примера стоит посмотреть, что мы пишем в программе и что собираем. Взгляните на код *программы 15.1*.

ПРОГРАММА 15.1. Использование директивы `ADR`

```

/ **** Использование директивы ADR **** /
.global _start
_start:
    ADR R0, value
    MOV R1, #15

_exit:
    MOV R7, #1
    SWI 0

value:
    .word 255

```

Программа 15.1 особо ничего не делает, но указывает `ADR` на метку данных и указывает `R0` как регистр назначения. После сборки с параметром `-g` команда выдаст результат, похожий на приведенный в листинге 15.1.

Листинг 15.1. Дизассемблирование программы 15.1

```

0x10054 <_start>:  add     r0, pc, #8
0x10058 <_start+4>:  mov     r1, #15
0x1005c <_exit>:     mov     r7, #1
0x10060 <_exit+4>:  svc     0x00000000
0x10064 <value>:   ;<undefined> instruction:  0x000000ff

```

Откройте `GDB` и введите:

```
x/5i _start
```

Вы увидите, что директива `ADR` не собрала адрес. Она собрала относительный адрес, который будет использоваться как смещение для счетчика программ. Здесь команда `ADD` прибавляет 8 к значению в регистре `PC`, адрес значения которого идет сразу после последней команды.

Также обратите внимание, что команда

```
SWI 0
```

превратилась в

```
SVC 0x00000000
```

Эти мнемоники взаимозаменяемы. В наши дни предпочтительным методом, вероятно, является `SVC`, но я по старинке использую `SWI`. В последней строке хранится значение 255. GDB пытался интерпретировать это, но у него не получается.

Введите в GDB:

```
x/5w _start
```

Результат будет аналогичен приведенному в листинге 15.2.

Листинг 15.2. Шестнадцатеричный дамп программы 15.1

```
(gdb) x/5w _start
0x10054 <_start>: 0xe28f0008 0xe3a0a100f 0xe3a07001 0xef00000
0x10064 <value>: 0x000000ff
```

Здесь хорошо видно число 255 (0xFF или 0x000000ff), на метке `value:`. Это демонстрирует нам еще одну новую возможность ассемблера. В одной из предыдущих глав мы использовали директиву `.string` для загрузки символьной строки ASCII в память. Директива `.word` позволяет нам сохранить в памяти слово, или четыре байта. Она показана в *программе 15.1*, и на нее можно ссылаться, используя именованную метку (мы рассмотрим директиву `.word` и другие директивы в *главе 18*).

Кроме того, мы увидели еще один особенно важный аспект псевдокоманды `ADR`. Значения, на которые она ссылается, всегда должны находиться в пределах раздела `.text`, т. е. исполняемого раздела кода. Мы помним, что использование директивы `.string`, к которой обращалась команда `LDR`, выполняется в разделе `.data`. Если же вы попытаетесь использовать `ADR` для доступа к информации в области данных, то получите сообщение об ошибке.

Косвенная адресация

ARM построен на архитектуре «загрузки и сохранения», но обращаться к ячейкам памяти напрямую возможности нет. Вы можете обращаться только опосредованно, через регистр. Прелесть косвенной адресации заключается в том, что она позволяет получить доступ ко всему пространству памяти ARM через один регистр.

Существуют две команды для чтения и записи данных в память:

- ◆ `LDR` — загрузка в регистр из памяти;
- ◆ `STR` — сохранение из регистра в память.

Косвенная адресация позволяет легко выполнять чтение или запись в ячейку памяти. Адрес самой ячейки хранится в регистре. Таким образом, доступ к местоположению адреса осуществляется косвенно. Преимущество этого метода состоит в том, что вы можете переключиться на работу с другой ячейкой памяти, просто изменив содержимое регистра. Таким методом очень удобно обрабатывать таблицы данных. Вместо того чтобы писать отдельную процедуру для каждой точки данных, можно разработать универсальную процедуру, которой передавался бы нужный адрес в момент вызова.

В своей простейшей форме косвенная адресация имеет следующий формат:

```
LDR (<Суффикс>) <Операнд1> [<Операнд2>]
```

```
STR (<Суффикс>) <Операнд1> [<Операнд2>]
```

Например:

```
LDR R0, [R1]           @ Загружаем в R0 то, что лежит по адресу R1
```

```
STR R0, [R2]           @ Сохраняем значение из R0 по адресу R2
```

В результате выполнения этих команд некоторое слово данных переместится из одной точки памяти в другую. На рис. 15.1 это показано на примере команды:

```
LDR R0, [R1]
```

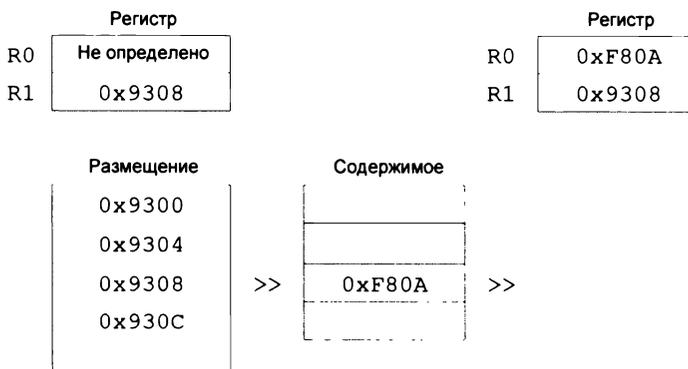


Рис. 15.1. Косвенная адресация памяти командой LDR R0, [R1]

Изначально в R1 содержится адрес памяти — в нашем случае: 0x9308, и по этому адресу лежит значение 0xF80A. Это значение загружается в R0. То есть по завершении команды в R0 будет лежать 0xF80A. Значение в R1 не изменится.

Все режимы адресации позволяют использовать суффиксы для условного выполнения. Например:

```
LDREQ R0, [R1]
```

Такая операция загрузки в R0 с адреса в R1 будет иметь место только в том случае, если установлен флаг Zero.

Команды ADR и LDR

В предыдущих примерах мы видели, что команду LDR можно использовать как псевдокоманду для загрузки адреса метки непосредственно в регистр. Например, команда

```
LDR R0, =string
```

загружает в R0 адрес метки string. Преимущество такого использования команды LDR заключается в том, что она может получать доступ ко всей памяти платы. Этот метод является предпочтительным, если вы используете разделы данных специально для хранения информации. А метки, к которым обращается команда ADR, должны находиться в разделах .text кода и внутри области исполняемого кода.

Предварительно индексированная адресация

Предварительно индексированная адресация дает возможность добавить к некоторому базовому адресу смещение, получая тем самым окончательный адрес. Смещение может быть константой, значением в регистре или сдвинутым содержимым регистра. Синтаксис команды:

```
LDR (<Суффикс>) <Место назначения>, [(<База>, (<Смещение>)]
STR (<Суффикс>) <Место назначения>, [(<База>, (<Смещение>)]
```

Константа или регистр смещения указываются как часть *операнда2* через запятую в квадратных скобках. Например:

```
LDR R0, [R1, #8]
```

Здесь к адресу в R1 добавляется значение 0x08, а четырехбайтовое значение, расположенное по этому адресу (R1 + 8), помещается в R0. Значение в R1 не изменяется. Схема работы этой команды показана на рис. 15.2. В R1 находится адрес памяти 0x9300. Он добавляется к указанному постоянному значению 8, чтобы получить окончательный адрес 0x9308. Содержимое, лежащее по этому адресу: 0xF80A, загружается в R0.

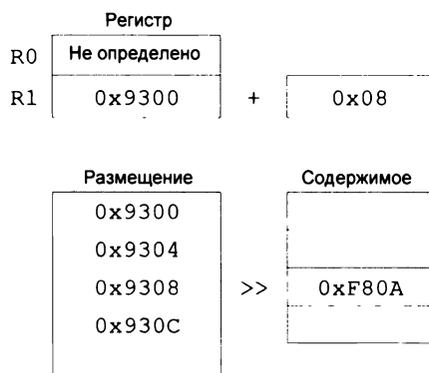


Рис. 15.2. Предварительно индексированная адресация

Вы также можете указывать два регистра внутри квадратных скобок:

```
STR R0, [R1, R2]
```

Эта команда при выполнении сохранит значение в R0 по адресу, полученному путем сложения содержимого регистров R1 и R2. Сами значения в R1 и R2 никак не меняются.

Значение смещения можно и вычесть, используя знак «минус»:

```
LDR R0, [R1, #-8]
STR R0, [R1, -R2]
```

Наконец, операнд смещения можно сдвигать с помощью одной из операций сдвига:

```
LDR R0, [R1, R2, LSR#4]
```

Значение в R2 сдвигается вправо на два бита и прибавляется к R1. В результате получается адрес данных для загрузки в R0. Эта конструкция полезна, когда речь идет о перемещении по данным, хранящимся в памяти, учитывая, что они расположены в четырехбайтовых блоках (размером с регистр) и что операция LSL #2 (которая дает результат $2 \times 2 = 4$) как раз и подводит «курсор» к границе следующего слова.

Следующий фрагмент кода заменяет третий элемент в четырехбайтовом списке вторым элементом. При этом адреса начала списка находятся в R1 (в нашем случае это 0x9300):

```
MOV R2, #4           @ четырехбайтовое смещение
LDR R4, [R1, R2]    @ загрузка R4 с адреса (0x9300 + 4)
STR R4, [R1, R2, LSL #1] @ сохранение R4 по адресу (0x9300 + 8)
```

В качестве смещения для первой команды задано значение 4. Затем команда LSL #1 сдвигает биты на четыре позиции. Позиция #4 в R2 становится #8 и добавляется к адресу в R1. Значение в R2 не меняется. Если вы хотите найти следующий элемент в списке, нужно будет увеличить либо R1, либо R2 на четыре. Но существует и гораздо более элегантный способ.

Доступ к байтам памяти

В *программе 15.2* показано использование предварительно индексированной косвенной адресации со смещением. Этот механизм задействуется для извлечения символов из строки, расположенной по базовому адресу. Кроме того, можно применить команду LDRB — для загрузки в регистр одного байта и команду STRB — для сохранения одного байта.

Символы, закодированные в ASCII, находятся в отдельных байтах, поэтому нам нужна именно команда LDRB, которая позволит загружать в указанное место отдельные байты памяти, а не слово. Для начала в R1 загружается адрес строки, а в R2 — значение смещения 26. Команда STRB дополняет LDRB, записывая в память один байт информации. В *программе 15.2* обе команды используются для замены одной строки на другую.

ПРОГРАММА 15.2. Использование предварительно индексированной косвенной адресации

```

/ * Использование предварительно индексированного адреса для перемещения
символов * /
.global _start
_start:
    LDR R1, =string      @ получение местонахождения первой строки
    LDR R3, =numbers     @ получение местонахождения второй строки
    MOV R2, #26          @ символы расположены по алфавиту

_loop:
    LDRB R0, [R1, R2]    @ получение байта по адресу R1 + R2
    STRB R0, [R3, R2]    @ сохранение байта по адресу R3 + R2
    SUBS R2, R2, #1      @ вычитание единицы, проверка флага
    BPL _loop            @ повтор цикла, пока результат положительный

_write:
    MOV R0, #1
    LDR R1, =numbers
    MOV R2, #26
    MOV R7, #4
    SWI 0

_exit:
    MOV R7, #1
    SWI 0

.data
string:
    .ascii "ABCDEFGHJKLMNOPQRSTUVWXYZ"
numbers:
    .ascii "01234567891011121314151617"

```

Сначала в разделе `_start` в регистры `R1` и `R3` загружаются адреса двух строк. `R2` используется для хранения значения счетчика, изначально равного 26, — количеству букв в алфавите.

Команда `LDRB` загружает байт с адреса `R1 + R2` в `R0`, а результат затем сохраняется в `R3 + R2`. Получается, что последняя буква в строке сохраняется по последнему адресу. После каждого прохода `R2` уменьшается на единицу, и пока число не станет равно нулю или меньше нуля, цикл будет повторяться. Когда `R2` достигнет нуля, чтение/запись завершаются, и процедура `_write` выводит новую строку.

В этих примерах мы не использовали прямые константы, но можно было бы и использовать, задавая с их помощью даже отрицательные значения. Вот пара примеров:

```

STR R0, [R1, #0xF0]
LDR R0, [R1, #-4]

```

Здесь в `R0` будут загружены данные, взятые с адреса, расположенного на одно слово ниже адреса, содержащегося в `R1`.

Обратная запись адреса

При вычислении местоположения в памяти слова или байта ARM складывает значения элементов, заключенных в квадратные скобки. Первое слагаемое — это регистр с адресом, а второе — другой регистр или константа. Результат сложения этих значений отбрасывается сразу после использования и не сохраняется.

Иногда бывает полезно сохранить вычисленный адрес, и в предварительно индексированной адресации это можно сделать, используя так называемую *обратную запись*. Для этого достаточно добавить символ ! в конце команды после закрывающей квадратной скобки:

```
LDR R0, [R1, R2]!  
LDRB R0, [R2, #10]!
```

В первом примере, вспомнив нашу предыдущую программу, предположим, что в R1 лежит адрес 0x9300, а в R2 — значение смещения, изначально равное 26. На первой итерации сумма R1 и R2 дает адрес, равный $0x9300 + 26$, т. е. 0x931A. С этого адреса берется некоторая информация, а затем значение 0x931A записывается обратно в R1.

Для прохода по массиву данных, хранящемуся в памяти, мы могли бы использовать инструкцию

```
LDR R0, [R1, #4]!
```

Значение 4 будет всякий раз прибавляться к R1, создавая шаг в одно слово. Значение в R1 обновляется, каждый раз превращаясь в $R1 + 4$. Обернув эту команду циклом, мы можем легко и быстро просмотреть память.

Постиндексированная адресация

В постиндексированной адресации функция обратной записи используется по умолчанию, а значение смещения нужно указывать обязательно. Само же смещение также обрабатывается иначе. Постиндексированная адресация записывается следующим образом:

```
LDR (<Суффикс>) <Место назначения>, [<Операнд1>], <Операнд2>
```

Прежде всего заметим, что обязательный *операнд2* находится за пределами квадратных скобок, обозначая тем самым разницу в режимах адресации. Вот несколько примеров использования синтаксиса команды:

```
LDR R0, [R1], R2  
STR R3, [R4], #4  
LDRB R6, [R1], R5, LSL#1
```

В постиндексированной адресации в качестве адреса источника или назначения принимается только содержимое базового регистра (слово или байт в зависимости от формата команды). Затем, уже после загрузки или сохранения, содержимое поля смещения (*операнд2*) добавляется к базовому регистру и результат записывается

в него. Таким образом, смещение добавляется после доступа к памяти, а не до него. На рис. 15.3 схематически показано, как это работает в команде:

```
LDR R0, [R1], #8
```

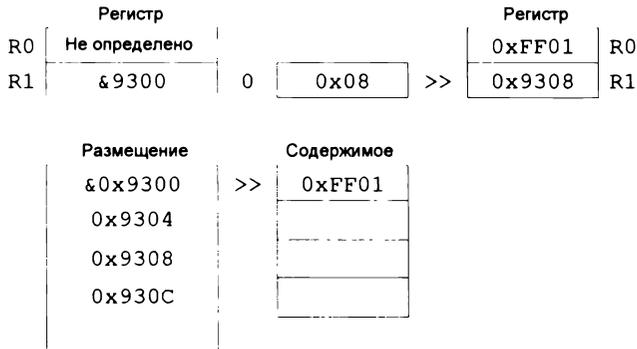


Рис. 15.3. Схема постиндексированной адресации

В левой части схемы показана расстановка до выполнения команды. На начальном этапе содержимое R0 еще не определено, в R1 содержится адрес 0x9300. По адресу 0x9300 лежит значение 0xFF01, которое извлекается и помещается в R0. К содержимому R1 добавляется промежуточное значение 8 (0x9300 + 08), и результат записывается обратно в R1, который теперь содержит значение 0x9308, что видно в правой части схемы.

Если бы мы выполнили строку LDR еще раз, то значение по адресу 0x9308 было бы извлечено и поместилось бы в R0, а после добавления к нему 8 в R1 стало бы лежать значение 0x9310.

В программе 15.3 представлена процедура машинного кода, в которой используется постиндексированная адресация для объединения двух строк в одну.

Обратите внимание, что в определениях данных для строки мы немного изменили директиву `.ascii`. Мы написали `.asciz` — эта директива помещает нулевой байт (0x0) в конец строки, который поможет нам понять, достигли ли мы конца строки во время загрузки, и сравнить части программы.

ПРОГРАММА 15.3. Использование постиндексированной адресации

```
/* Постиндексированная адресация и объединение строк */
.global _start
_start:
    LDR R2, =string1      @ адрес загрузки
    LDR R3, =string2     @ обеих строк
_loop:
    LDRB R0, [R3], #1    @ получение байта строки 2 и прибавление 1
    CMP R0, #0           @ достигнут ли конец строки?
    BNE _loop           @ если нет, переход к следующему байту
    SUB R3, R3, #1      @ если да, вычитание 1
```

```

_copyloop:
    LDRB R0, [R2], #1      @ получение байта из строки 1
    STRB R0, [R3], #1      @ добавление в конец строки 2
    CMP R0, #0             @ это ноль?
    BNE _copyloop         @ если нет, берется следующий символ

_write:
    MOV R0, #1             @ если 0, вывод строки
    LDR R1, =string2
    MOV R2, #24
    MOV R7, #4
    SWI 0

_exit:
    MOV R7, #1
    SWI 0

.data
string1:
    .asciz "ABCDEFGHJKLM"
string2:
    .asciz "012345678910"
padding:
    .ascii "          "

```

Байтовые условия

С командами загрузки и сохранения можно, как и ранее, применять условные суффиксы. Но при использовании байтового модификатора вместе с условными выражениями условие пишется первым, например:

```
LDREQB R0, [R1]
```

Обратите внимание, что условие `EQ` стоит перед модификатором байта `b`. В случае нарушения порядка появится сообщение об ошибке.

Относительная адресация через регистр *PC*

Помимо рассмотренных нами видов адресации ассемблер GCC сам реализует дополнительный режим псевдоадресации — относительную адресацию через *PC*. Мы уже пользовались этим методом ранее, но теперь стоит подчеркнуть его полезность, уделив ему особое внимание.

Общий формат команд, использующих относительную адресацию *PC*, следующий:

```
LDR <Место назначения>, <Адрес>
```

Как и прежде, адресатом всегда является регистр, в который — или из которого — передаются данные. Адрес — это либо число, либо метка ассемблера. В последнем

случае метка отмечает адрес, откуда берутся данные. Давайте взглянем на пару примеров:

```
LDR R0, 0x9300
STR R0, data
```

В первом случае слово, расположенное по адресу 0x9300, будет загружено в R0. Во втором случае место, в котором была собрана метка `data`, будет использоваться как место назначения для слова из R0.

Когда ассемблер встречается написанную в таком формате команду, он смотрит на адрес или расположение адреса и вычисляет расстояние от него до места назначения. Это расстояние называется *смещением*, и при добавлении к программному счетчику оно даст абсолютный адрес расположения данных. Зная это, ассемблер может скомпилировать инструкцию, использующую предварительно индексированную адресацию. Базовым регистром в этой команде будет программный счетчик R15. Если не учитывать эффекты конвейерной обработки, в PC окажется адрес команды. В поле смещения содержится значение смещения, рассчитанное ассемблером ранее, с поправкой на конвейерную обработку. (Этот метод аналогичен тому, что мы описывали в *главе 10*, говоря о ветвлении.)

Важно помнить, что существует ограничение значения смещения, которое можно использовать при предварительно индексированной адресации. Оно находится в диапазоне от -4096 до 4096, и смещение в относительной адресации PC должно лежать в пределах этого диапазона.

16. Передача блока

Эффективность — краеугольный камень дизайна чипа ARM. Работая с чипами, имеющими большое количество регистров и постоянную потребность в манипулировании и перемещении данных, сложно упорядочить целую серию команд для передачи блока (набора регистров) из одного места в другое. Команды LDM и STM позволяют упростить повторяющиеся операции загрузки и сохранения в регистры из памяти и наоборот.

Синтаксис команд:

```
LDM <Опции> (<Суффикс>) <Операнд1> (!), {<Регистры>}
```

```
STM <Опции> (<Суффикс>) <Операнд1> (!), {<Регистры>}
```

Здесь *Регистры* — это список участвующих в передаче регистров, заключенных в фигурные скобки и разделенных запятыми. Указывать их можно в любом порядке, а если вы хотите задать сразу несколько регистров, это делается с помощью дефиса: R5–R9.

Операнд1 — это регистр, который содержит адрес, обозначающий начало памяти, которая будет использоваться для операции. Этот адрес не изменяется:

```
STM R0, {R1, R5–R8}
```

В этом случае последовательно считывается и сохраняется содержимое регистров R1, R5, R6, R7 и R8 (пять слов, или 20 байтов), начиная с адреса, хранящегося в R0. Если бы в R0 лежал адрес 0x9300, R1 попал бы по этому адресу, R5 — по адресу 0x9304, R6 — по адресу 0x9308. Схема сохранения приведена на рис. 16.1.

В этом примере предполагается, что мы хотим, чтобы данные хранились в последовательно увеличивающихся ячейках памяти, но это не обязательно. ARM предоставляет параметры, которые позволяют обращаться к памяти в порядке возрастания или убывания, а также способ обработки шага приращения. Фактически есть четыре варианта, как показано на рис. 16.2.

Буквы I или D в суффиксе определяют, в каком направлении в памяти смещается адрес для следующего объекта: I — это увеличение, а D — уменьшение. Базовый адрес увеличивается или уменьшается на четыре байта за раз.

Регистр			Адрес в памяти	Значение
R0	0x9300	>>	0x9300	0xFF00FF00
R1	0xFF00FF00		0x9304	0x2A0D4AA
R2	0xFF		0x9308	0x953A
R3	0xA8FB		0x930C	0xF36BCA
R4	0xAF2		0x9310	0x101
R5	0x2A0D4AA			
R6	0x953A			
R7	0xF36BCA			
R8	0x101			

Рис. 16.1. Сохранение содержимого регистра в памяти

Суффикс	Значение
IA	Увеличить после
IB	Увеличить до
DA	Уменьшить после
DB	Уменьшить до

Рис. 16.2. Суффиксы для установки направления памяти

После каждой команды ARM будет выполняться одно из следующих действий:

- ◆ Инкремент: $\text{Адрес} = \text{Адрес} + 4 \times n$;
- ◆ Декремент: $\text{Адрес} = \text{Адрес} - 4 \times n$,

где n — количество регистров в списке.

Параметры A или B определяют, в какой момент меняется базовый адрес: до или после доступа к памяти. Здесь есть тонкая разница, и при отсутствии должной осторожности вы можете сделать так, что данные попадут не туда и результаты искажутся. Схемы взаимодействия приведены на рис. 16.3 и 16.4.

Левая модель на рис. 16.3 показывает работу сохранения в режиме «увеличение после». После сохранения первого регистра ($R0$) указатель памяти увеличивается и теперь содержит базовое значение $+4$, и по этому адресу размещается содержимое $R1$. С правой стороны показана модель «увеличение до». В момент выполнения четверка прибавляется к базовому значению сразу же, и содержимое $R0$ сохраняется по этому адресу.

На рис. 16.4 показаны те же действия, за исключением того, что в каждом случае 4 вычитается из базового значения либо до сохранения, либо после.

Базовый адрес STMIA {R0 – R6}

R6	Базовый адрес + 24
R5	Базовый адрес + 20
R4	Базовый адрес + 16
R3	Базовый адрес + 12
R2	Базовый адрес + 8
R1	Базовый адрес + 4
R0	<< Базовый адрес

Базовый адрес STMIB {R0 – R6}

R6	Базовый адрес + 28
R5	Базовый адрес + 24
R4	Базовый адрес + 20
R3	Базовый адрес + 16
R2	Базовый адрес + 12
R1	Базовый адрес + 8
R0	Базовый адрес + 4
	<< Базовый адрес

Рис. 16.3. Влияние на работу команды STM суффиксов IA и IB

Базовый адрес STMDA {R0 – R6}

R0	<< Базовый адрес
R1	Базовый адрес + 4
R2	Базовый адрес + 8
R3	Базовый адрес + 12
R4	Базовый адрес + 16
R5	Базовый адрес + 20
R6	Базовый адрес + 24

Базовый адрес STMDB {R0 – R6}

	<< Базовый адрес
R0	Базовый адрес + 4
R1	Базовый адрес + 8
R2	Базовый адрес + 12
R3	Базовый адрес + 16
R4	Базовый адрес + 20
R5	Базовый адрес + 24
R6	Базовый адрес + 28

Рис. 16.4. Влияние на работу команды STM суффиксов DA и DB

Обратная запись

Если команда явно не требует выполнения обратной записи, адрес, хранящийся в регистре, остается неизменным. Это значит, что его содержимое остается таким же, каким оно было до выполнения команды. Если же нам нужно, чтобы значение

в регистре обновлялось, т. е. выполнялась обратная запись, необходимо добавить символ !:

```
LDMIA R0!, {R2-R4}
STMDA R0!, {R5-R8, R10}
```

В адресный регистр (R0) в итоге попадает адрес, вычисленный после обработки последнего регистра в списке.

Команды STM и LDM имеют множество применений. Одним из наиболее очевидных является возможность сохранения и восстановления содержимого всех регистров. Если в R0 содержится адрес свободного блока памяти, то одной командой можно сохранить все регистры:

```
STMIA R0, {R1-R14}
```

или восстановить их:

```
LDM R0, {R1-R14}
```

если в R0 все так же лежит нужный адрес.

Не рекомендуем добавлять в этот список R15. Если вы выполните восстановление командой LDM, добавив в список R15, программа, вероятно, попадет в бесконечный цикл.

Функция обратной записи в командах блочной передачи предназначена для упрощения создания стеков, о которых мы побеседуем в следующей главе.

Процедура копирования блока

В программе 16.1 показано, как легко можно скопировать блок данных из одного места в памяти в другое. Для всей этой процедуры достаточно всего четырех строк ассемблера, а в результате мы получаем надежное копирование блока памяти любой длины при условии, что длина эта делится на восемь.

ПРОГРАММА 16.1. Перемещение блоков памяти

/* Программа для копирования блока памяти */

```
.global _start
_start:
    LDR R0, =begin           @ адреса загрузки
    LDR R1, =end             @ обеих строк
    LDR R2, =dest           @ адрес назначения

_blockcopy:
    LDMIA R0!, {R3-R4}
    STMIA R2!, {R3-R4}
    CMP R0, R1
    BNE _blockcopy
```

```

_exit:
    MOV R7, #1
    SWI 0

.section .data
begin:
    .word 0xFFFFFFFF
    .word 0xFFFFFFFF

end:
    .word 0
    .word 0

dest:
    .word 0
    .word 0

```

В этой процедуре регистры R3 и R4 используются для загрузки, а затем сохранения данных, поэтому лежащая в них до этого информация будет уничтожена, если ее не сохранить заблаговременно. Регистры R0, R1 и R2 содержат адреса, обозначающие начало и конец данных и начало сегмента в памяти, куда выполняется сохранение.

Чтобы проследить за работой этой процедуры, вы можете использовать GDB. Обязательно выполняйте сборку с параметром `-g`. Войдите в GDB и загрузите файл

```
gdb prog16a
```

Установите точку останова в процедуре `_exit` (она находится сразу после цикла копирования блока):

```
break _exit
```

Теперь запустите программу:

```
run
```

Программа запустится до точки останова, после чего введите:

```
x/2x &dest
```

Символ `&` здесь означает «местоположение `dest`». Если вы не используете этот символ, метки данных не распознаются. Когда отобразятся два слова памяти, вы увидите, что в них все биты установлены (все единицы), а это значит, что копирование блока работает, поскольку изначально они были нулями.

Эту процедуру можно использовать для обработки больших блоков памяти. Например, изменив две команды загрузки и сохранения следующим образом:

```

LDMIA R0!, {R3-R12}
STMIA R2!, {R3-R12}

```

Вы можете работать блоками по 40 байтов (10 регистров по 4 байта каждый). Но области данных должны быть подготовлены, иначе вместо использования меток вам придется писать абсолютные адреса в памяти.

17. Стеки

Стеки используются в компьютерных системах с незапамятных времен. Их название, в общем-то, говорящее: это наборы данных, но не простые, а такие, которыми вы, как программист, владеете и управляете. Управление стеками — это один из важнейших компонентов разработки программ. Если вы будете работать с ними правильно, программа в благодарность вам тоже будет работать хорошо. В противном случае вам придется выдергивать из розетки шнур питания платы.

Стеки похожи на стопки тарелок: вы можете долго ставить на верх стопки новые тарелки, но ежели вам понадобится взять тарелку из стопки, вам достанется последняя тарелка, которую вы положили на нее, т. е. самая верхняя. То есть последняя прибывшая тарелка выбывает первой. Собственно, эта структура так и называется: LIFO (Last In First Out, или «последним пришел — первым вышел»). Попробуйте вынуть тарелку из середины (или снизу!). Если не проявить осторожность, вся стопка рухнет. Ну вы поняли.

Тянитолкай ;-)

На заре домашних компьютеров в системах вроде микропроцессора 6502 стеки создавались очень легко. Вы просто могли помещать данные в стек («толкать») и извлекать («вытягивать») их из стека. Обычно вы даже не знали, где находится стек, т. к. этим вопросом занимался процессор. Однако, как программисту, вам нужно было следить за порядком попадания объектов в стек. Как правило, описанный принцип работает и по сей день, поскольку последовательность данных, извлеченных из стека, всегда извлекается из него в обратном порядке.

Команды STM и LDM и их производные используются для передачи (STM) данных в стеки ARM и извлечения (LDM) данных из них. Эти стеки представляют собой области памяти, которые мы, как программисты, определяем сами. Количество стеков, которые можно задействовать, не ограничено. Единственным ограничением является объем памяти, доступный для их реализации.

Регистр R13, также известный как *указатель стека* (Stack Pointer, SP), предназначен для хранения адреса текущей используемой ячейки стека, но вы можете задействовать для этой цели любой из доступных регистров. При использовании нескольких



Рис. 17.1. Простой стек, в котором каждый элемент стека имеет ширину четыре байта

стеков вам понадобится выделить больше регистров или указать, где вы храните адреса в памяти. На рис. 17.1 показан простой стек.

Реализовать простой стек можно с помощью следующих команд (важно отметить, что параметры команд `STM` и `LDM` противоположны):

```
STMIA SP!, {R0-R12, LR} @ помещение регистров в стек
LDMDB SP!, {R0-R12, PC} @ извлечение регистров из стека
```

Опции суффиксов `IA` и `DB` (о которых мы говорили в предыдущей главе) используются в тандеме для перемещения вверх по памяти и сохранения значений, а затем вниз по памяти и загрузки. Регистры `LR` и `PC` служат для хранения адреса программного счетчика, поэтому эти две строки позволяют эффективно сохранить содержимое регистра перед вызовом некоторой процедуры и восстановить все, как было, при возврате из нее.

Крайне важно здесь использовать обратную запись. Без обратной записи указатель стека не будет обновляться и стек попросту сломается, поскольку мы не будем знать, в какой его точке в текущий момент находимся.

Можно реализовать стек с двумя указателями. Первый — это базовая точка, указывающая на адрес в памяти, с которого начинается стек. Второй — указатель стека, указывающий на вершину стека. Базовый адрес не меняется, а указатель стека может выражаться в виде меняющегося адреса или смещения от базового указателя. Надеюсь, теперь вы понимаете, как можно использовать разные режимы адресации для организации разных типов стеков. Какой бы метод вы ни использовали, всегда нужно будет записывать, где начинается стек и где он должен заканчиваться. Не зная этих крайних адресов, вы столкнетесь с проблемами. А вот вопрос: указывает ли указатель стека на адрес следующей свободной ячейки в стеке или же на адрес последней использованной ячейки? Чтобы не думать об этом, для помещения и извлечения информации в стек и из него можно использовать псевдокоманды `PUSH` и `POP`:

```
PUSH {R0, R3, R5} @ помещение R0, R3, R5 в стек
POP {R0, R5-R8} @ извлечение R0, R5, R6, R7, R8 из стека
```

Рост стека

В архитектуре ARM стеки подразделяются по признаку метода их роста в памяти. В процессе роста стеки могут подниматься по памяти, а могут и спускаться по ней. Но тут, как в космосе: никакого верха или низа в памяти нет, и термины эти относительны. Например, мы стоим на 10-м этаже пустого 20-этажного дома. 10-й этаж — единственный вход, и на каждом этаже, сверху и снизу, по четыре квартиры. Приезжают восемь семей; вы можете разместить их на два этажа выше или на два ниже. Как это сделать?

С точки зрения компьютерной памяти стек, который растет (или *поднимается по памяти*), — это стек, адрес вершины которого увеличивается. Когда в такой стек помещается элемент, указатель стека увеличивает его адрес на четыре байта. Стек, который растет в памяти с уменьшением адреса, называется *нисходящим стеком*. Всего существует четыре типа стеков (табл. 17.1).

Таблица 17.1. Четыре типа стеков ARM

Постфикс	Значение
FA	Полный восходящий стек
FD	Полный нисходящий стек
EA	Пустой восходящий стек
ED	Пустой нисходящий стек

Когда указатель стека указывает на последний занятый адрес в стеке, стек называется *полным*. Когда указатель стека указывает следующее доступное пустое место в стеке, он называется *пустым* стеком. Обратите внимание, что слово «пустой» не означает отсутствие данных в стеке, а лишь указывает на то, что в адресе хранится следующее свободное слово стека.

Выбор вариантов «полный/пустой» и «восходящий/нисходящий» часто происходит сам собой и зависит только от того, как вы просматриваете свои данные в цикле. Например, иногда бывает проще реализовать нисходящий стек, если ваш код заточен на декрементные счетчики или если вам проще проверить флаг нуля.

В наборе команд ARM есть команды, реализующие все эти виды стеков, и они приведены в табл. 17.2.

Таблица 17.2. Команды для доступа к стекам

Пары команд	Значение
STMFD/LDMFD	Полный нисходящий стек
STMFA/LDMFA	Полный восходящий стек
STMED/LDMED	Пустой нисходящий стек
STMEA/LDMEA	Пустой восходящий стек

Вот пара примеров:

```
STMED R13!, {R1-R5, R6}
LDMFD R13!, {R1-R4, R6}
```

Никто не мешает вам использовать разные типы стеков в одной программе. Но не путайте их! Сейчас вы уже должны понимать, почему при использовании этих команд важна обратная запись.

Примеры стеков показаны на рис. 17.2 и 17.3. По умолчанию ARM реализует полный нисходящий стек.

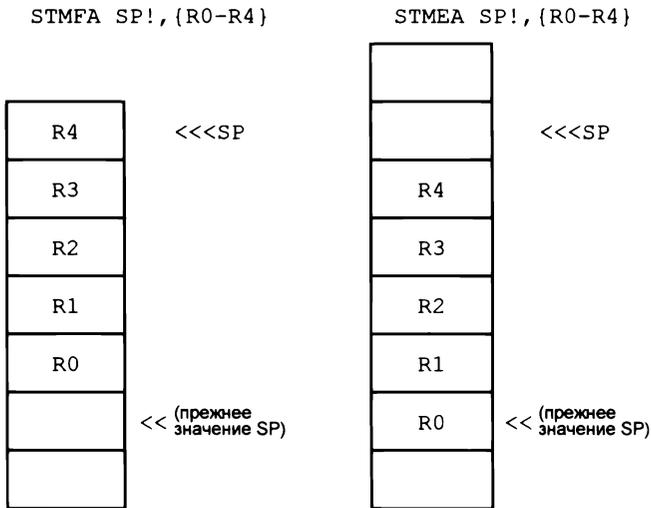


Рис. 17.2. Полный (слева) и пустой (справа) восходящие стеки

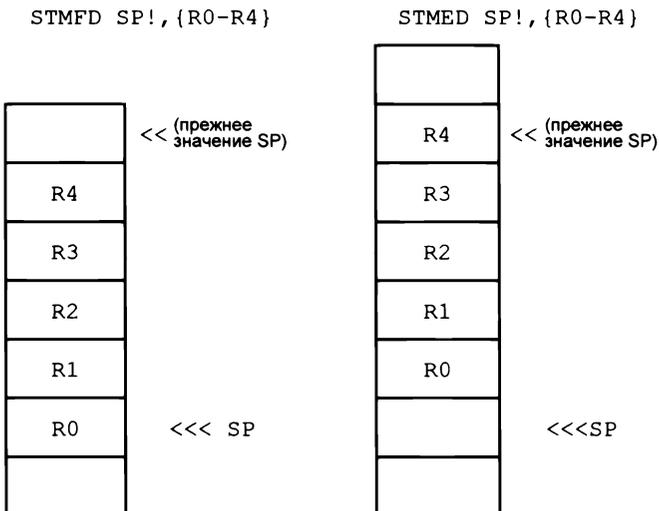


Рис. 17.3. Полный (слева) и пустой (справа) нисходящие стеки

Лучший способ понять, как работают стеки, — поэкспериментировать с ними. Попробуйте заполнить область памяти известными значениями, а затем попытайтесь переместить этот блок памяти в другое место через стек, сохранив при этом и саму информацию, и ее порядок.

Применение стеков

У стеков множество применений, и о некоторых из них мы уже упомянули:

- ◆ сохранение содержимого регистров;
- ◆ сохранение и обработка данных.

Есть и третье применение — сохранение адресов ссылок при вызове подпрограмм. Помещая адреса ссылок из регистра ссылок в стек, можно создавать вложенные (одна в другую) процедуры, не опасаясь потерять нить управления программой. Переходя к некоторой процедуре, вы помещаете регистр ссылок в стек. Затем вы можете вернуться из процедуры, вытащив адреса ссылок из стека и поместив их обратно в счетчик программ.

Стек также позволяет без труда менять местами содержимое регистров без необходимости использовать дополнительные регистры в качестве буфера. Вы просто помещаете необходимые регистры в стек, а затем извлекаете их в том порядке, в котором они вам нужны. Представьте себе ситуацию, когда необходимо поменять местами содержимое регистров:

Регистр-источник	Регистр назначения
R0	R3
R1	R4
R2	R6
R3	R5
R4	R0
R5	R1
R6	R2

На первый взгляд это выглядит сложно. Но со стеками все просто:

```

STMFD SP!, {R0-R6}
LDMFD SP!, {R3, R4, R6}
LDMFD SP!, {R5}
LDMFD SP!, {R0, R1, R2}

```

Первая строка помещает регистры с R0 по R6 в стек. Три верхних элемента в стеке (в порядке убывания): R0, R1 и R2. Согласно приведенной схеме они должны попасть в R3, R4 и R6, что и делается во второй строке.

Указатель стека теперь расположен в R3, который попадает в R5. В результате в стеке остаются только R4, R5 и R6, которые в последней строке извлекаются в R0, R1 и R2 соответственно. Не так страшен черт, как его малюют.

Работа в фрейме

Существует только одна «официальная» реализация стека, с которой будут работать команды, описанные в этой главе. Если вы реализуете свой собственный стек, вам придется самостоятельно управлять блоком памяти, в котором он находится. Для этого в программе должны быть метки.

Еще один метод, работающий с «настоящим» стеком, — это создание *фрейма* стека. То есть такой области в стеке, которую мы можем «зарезервировать» для своих целей. Для этого мы задаем указатель стека, создавая в стеке пустую область.

Например, нам нужно сохранить содержимое трех регистров, для чего потребуется 12 байтов (3 регистра × 4 байта = 12). В следующем фрагменте кода мы вручную задаем указатель стека, создавая в нем пробел:

```
SUB SP, #12      @ отделение 12 байтов от указателя
STR R1, [SP]
STR R2, [SP, #4]
STR R3, [SP, #8]
```

Регистры R1, R2 и R3 хранятся в созданной нами пустой области, как показано на рис. 17.4. Вам нужно будет не забыть закрыть эту область в стеке, когда закончите все, что хотели сделать:

```
ADD SP, #12
```



Рис. 17.4. Формирование пустой области фрейма в стеке

Этот метод довольно часто используется для сохранения содержимого регистра перед вызовом функции (см. главу 21).

Указатель фрейма

При частом использовании стека бывает трудно запомнить, где что находится и где какое смещение. Вместо использования команды SP и смещений мы можем вызвать указатель на фрейм стека (Frame Pointer, FP). Обычно для этого принято использовать регистр R11, но это не жесткое правило. Просто не забудьте поместить содержимое регистров в стек, чтобы его можно было восстановить позже. После создания фрейма стека мы можем установить указатель фрейма на следующее свободное место в стеке (помня, что он растет по убыванию адресов):

```
SUB FP, SP, #4  
SUB SP, #12
```

Теперь мы сможем использовать указатель фрейма для доступа к нашим переменным:

```
STR R1, [SP]  
STR R2, [FP, #-4]  
STR R3, [FP, #-8]
```

18. Директивы и макросы

GCC предоставляет множество дополнительных возможностей, помогающих писать программы машинного кода. Это, например, команды, которые позволяют хранить данные в ваших программах, а также передавать программе информацию в момент вызова из командной строки. Все директивы ассемблера начинаются с точки, и в GCC таких директив много. Мы уже видели некоторые из них в действии в наших программах. В этой главе мы рассмотрим часть из них более подробно.

Директивы хранения данных

Для хранения информации о символьной строке в программах используются две директивы:

```
.ascii "Это строка для печати."  
.asciz "В конец этой строки добавлен ноль"
```

Сама строка пишется в двойных кавычках. Символ *z* во второй директиве означает ноль, т. е. в конец строки добавляется нулевой байт (0x00). Это позволяет удобно пометить конец строки в памяти, поскольку при его поиске достаточно будет проверять флаг нуля. Обе директивы позволяют встраивать в них управляющие символы или символы escape-кода с помощью символа обратной косой черты (обратного слеша): \. В табл. 18.1 приведены некоторые из наиболее популярных и полезных символов.

Таблица 18.1. Популярные элементы управления с обратным слешем, используемые в строках

Управляющий символ	Действие
\b	Возврат на шаг
\f	Сдвиг вперед
\n	Перевод строки
\r	Возврат
\t	Табуляция

Таблица 18.1 (окончание)

Управляющий символ	Действие
\\	Добавление символа \
\"	Добавление символа "

Вот такая команда:

```
.ascii "1\t2\t3\r\n4\t5\t6\r\n7\t8\t9\r\n"
```

выведет простую, но аккуратно отформатированную таблицу через любую из показанных в этой книге процедур записи (не забудьте соответствующим образом подредактировать счетчик длины строки).

Когда ваши программы начнут становиться более сложными и реальными, вам потребуются хранить в них информацию. Информация может храниться в виде констант, адресов или вывода в консоль. В последнем случае используется строковый оператор. Помещая данные в тело машинного кода, мы можем быть уверены, что они «защищены».

В примере перемещения блока данных (см. главу 15) мы видели четкое указание, как с помощью директивы `.word` можно записывать четырехбайтовые слова информации в память. Помимо `.word` есть и другие директивы, которые могут создавать пространство данных аналогичным образом. В программе 18.1 показаны две из них: `.byte` и `.equ`.

ПРОГРАММА 18.1. Использование директив `.byte` и `.equ`

```
/* Использование директив byte и equ для суммирования набора чисел */
```

```
.global _start
_start:

    LDR R1, =values
    LDR R2, =endvalues
    MOV R0, #0

_loop:
    LDRB R3, [R1], #increment
    ADD R0, R0, R3
    CMP R1, R2
    BNE _loop

_exit:
    MOV R7, #1
    SWI 0

.data
.equ increment, 1
```

```
values:
    .byte 1,2,3,4,5,6,7,8,9
endvalues:
```

Директива `.byte` позволяет сохранять в памяти последовательность значений, указанных через запятые. В этой директиве такие значения должны находиться в диапазоне 0–255.

Директива `.equ` позволяет немедленно присвоить имени некоторое значение. Позже это имя можно использовать в ваших исходных файлах. Это удобно тем, что если вам нужно в какой-то момент изменить значение, вам просто надо будет изменить определение директивы `.equ`, а не ссылки на нее в исходном файле.

Если вы посмотрите на раздел `.data` программы 18.1, вы увидите, что константе `increment` присвоено значение 1. Она используется в качестве счетчика постиндексации в начале процедуры `_loop`.

Метка `values:` обозначает начало определения директивы `.byte`. Вторая метка, называемая `endvalues:`, служит для обозначения конца последовательности `.byte`. Это удобный метод, который можно использовать при работе с таблицами или массивами данных, поскольку с помощью команды `cmp` можно проверить, достигнут ли конец последовательности.

Если вы соберете и запустите программу 18.1, а затем введете:

```
echo $?
```

вам вернется значение 45 — сумма байтов. В табл. 18.2 приведено несколько важных директив для работы данными.

Таблица 18.2. Важные директивы для работы с данными

Директива	Назначение
<code>.equ</code>	Присвоение константы метке. Например: <code>.equ one, 1</code>
<code>.byte</code>	Сохранить в память значения шириной в 1 байт, разделенные запятыми. Пример: <code>.byte 1,2,3,55,255</code>
<code>.word</code>	Сохранить в память значения шириной 4 байта, разделенные запятыми. Пример: <code>.word 0xFFFFFFFF, 0xFF</code>

Выравнивание данных

Если вы собираетесь хранить данные в исполняемых сегментах разделов `.text` вашей программы, могут возникнуть проблемы. Все коды операций должны начинаться на границе слова. Если используемый вами текст или данные не полностью заполняют пространство до четырехбайтовой границы, ассемблер «обидится» и сообщит об ошибке:

```
Unaligned opcodes detected in executable segment
```

Вернемся к *программе 18.1*. Если вы добавите в конец раздела `_start` следующие строки:

```
BAL _loop
_string:
.ascii "12345"
```

и попытаете собрать код, то получите указанную ошибку. Это можно исправить, добавив после определения `.ascii` следующую директиву:

```
.align 2
```

Директива заполняет пространство до границы следующего слова нулями. Вы можете проверить работу этой директивы с помощью GDB.

Обычно использовать директиву `.align` за пределами исполняемых разделов кода не требуется. Любые определения, сделанные в разделах данных, как правило, сохраняются ассемблером в конце файла, что сразу решает проблему с выравниванием.

Макросы

Макрос — это фрагмент кода, который может иметь любую длину и определяется именем. Определение макроса можно вызвать из программы, используя имя макроса. Во время сборки блок ассемблера, приведенный в определении макроса, вставляется вместо каждого имени макроса в листинге.

Многие программисты пишут свои библиотеки макросов, которые они могут использовать в различных обстоятельствах. Достаточно просто загрузить нужные макросы, а затем при необходимости вызвать макрос из своей программы. Не путайте это с псевдокодом, показанным в листинге 2.1, — для перехода к различным частям программы используются вызовы процедур. Макросы создают линейный код, т. е. одну длинную программу! Но в любом случае мы под одним именем имеем в виду группу команд.

Макросы не заменяют процедуры, поскольку имя макроса просто заменяется его кодом и, следовательно, программа остается линейной. Длинные макросы, многократно используемые в программе, значительно увеличивают окончательный размер кода. В этом случае лучше использовать процедуры, поскольку их код не вставляется в исходный код при вызове.

Макросы полезны, когда требуется выполнять трудные или сложные вычисления, в которых легко допустить опечатку. Вы можете использовать внутри макроса постоянные данные и передавать ему при необходимости нужную информацию. Макросы также позволяют избежать затрат на вызов процедуры и возврат из нее, если сама процедура представляет собой всего несколько команд.

В *программе 18.2* определен простой макрос `addtwo`, принимающий два параметра: `val1` и `val2`, которые передаются в `R1` и `R2` соответственно и складываются, а результат помещается в `R0`.

ПРОГРАММА 18.2. Реализация простого макроса
--

```

/* Реализуем простой макрос #1 */

.global _start
_start:

.macro addtwo val1, val2
    MOV R1, #\val1
    MOV R2, #\val2
    ADD R0, R1, R2
.endm

    addtwo 3, 4

    MOV R7, #1           @ выход через системный вызов
    SWI 0

```

Директива `.macro` используется для определения макроса, которому мы даем имя `addtwo`, а для параметров я задал имена: `val1` и `val2`. Как видите, определение макроса завершается директивой `.endm`.

Обратите внимание, что внутри определения макроса перед двумя именованными параметрами стоит обратный слеш. Он говорит компилятору, что это именно параметры, а не абсолютные значения. Самая распространенная ошибка при написании макросов — это забыть поставить обратный слеш перед параметрами.

Вызвать макрос легко: просто вставьте его имя в ассемблер и укажите нужные параметры. Если вы запустите *программу 18.2*, а затем введете:

```
echo $?
```

то получите результат: 7.

Стоит взглянуть на код, созданный *программой 18.2*. Если вы собрали и скомпоновали его с помощью опции `-g`, то можете посмотреть код в GDB с помощью команды

```
x/20i _start
```

Команда выдаст результат, похожий на приведенный в листинге 18.1.

Листинг 18.1. Дизассемблированный вывод программы 18.2

```

0x10054 <_start>:   mov    r1, #3
0x10058 <_start+4>:  mov    r2, #4
0x1005c <_start+8>:  add    r0, r1, r2
0x10560 <_start+12>: mov    r7, #1
0x10064 <_start+16>: svc    0x00000000
0x10068 Cannot access memory at address 0x10068

```

Обратите внимание, что в ассемблированный код были переданы непосредственные значения в первых двух строках: `<_start>` и `<_start+4>`. Это не вызов подпрограммы. Просто нужный код был вставлен в нужную точку.

Программа 18.3 представляет собой модифицированную версию той же задачи, только теперь команда `MLA` используется для сложения произведений каждого умножения. На этот раз мы определяем макрос `multtwo` три раза, чтобы передать макросу три набора значений для вычисления:

$$(2*2) + (3*4) + (5*6)$$

Однако здесь есть нюанс: на этом этапе проверка ошибок не производится.

ПРОГРАММА 18.3. Многократный вызов макроса

```
/* Реализуем простой макрос #2 */

.global _start
_start:

.macro multtwo val1, val2
    MOV R1, #\val1
    MOV R2, #\val2
    MLA R0, R1, R2, R0
.endm

    MOV R0, #0
    multtwo 2, 2
    multtwo 3, 4
    multtwo 5, 6

    MOV R7, #1          @ выход через системный вызов
    SWI 0
```

Соберите и запустите эту программу. Появится приглашение, после чего вы можете получить результат, используя команду:

```
echo ?$
```

Должно получиться: 46.

Дизассемблирование этого кода даст результат, похожий на приведенный в листинге 18.2, если вы посмотрите на него с помощью GDB и команды:

```
x/20i _start
```

Листинг 18.2. Дизассемблированный вывод программы 18.3

```
0x10054  <_start>:      mov     r0, #0
0x10058  <_start+4>:    mov     r1, #2
0x1005c  <_start+8>:    mov     r2, #2
```

```

0x10560  <_start+12>:      mla     r0, r1, r2, r0
0x10064  <_start+16>:      mov     r1, #3
0x10068  <_start+20>:      mov     r2, #4
0x1006c  <_start+24>:      mla     r0, r1, r2, r0
0x10070  <_start+28>:      mov     r1, #5
0x10074  <_start+32>:      mov     r1, #6
0x10078  <_start+36>:      mla     r0, r1, r2, r0
0x1007c  <_start+40>:      mov     r7, #1
0x10080  <_start+44>:      svc     0x00000000
0x10084:  Cannot access memory at address 0x10084

```

Еще раз обратите внимание на то, как макрос встроился в код. Это позволяет нам выделить несколько особенностей работы с макросами:

- ◆ окончательный размер кода собранного файла будет больше, чем можно ожидать (теоретически это может создать проблему со скоростью выполнения программы, но в среде RISC это обычно не проблема);
- ◆ отладка затрудняется, т. к. легко потеряться в длинных повторах кода;
- ◆ потребуется больше усилий по сохранению содержимого регистров в нужных местах, если это необходимо.

Включение макросов

Программы 18.2 и 18.3 демонстрируют нам полезность макросов, но в них макрос определен в той же программе, в которой вызывается. Основным преимуществом макросов является возможность создать из них библиотеку, которая позволит вам просто подключать нужный макрос или библиотеку макросов, если они, конечно, правильно написаны!

Давайте повторно рассмотрим ту же задачу (суммирование набора чисел), создав простой файл с макросами, содержащий две функции из *программ 18.2 и 18.3*. Он приведен в *программе 18.4*, которая на самом деле является не стандартной программой, а лишь содержит определения макросов. А *программа 18.5* — это программа тестирования для всего этого.

ПРОГРАММА 18.4. Файл макроса `AddMult`

```

/* Макросы: Addtwo и MultTwo */
.macro addtwo val1, val2
    @ При выходе в R1, R2 содержатся val1, val2
    @ В R0 лежит результат
    MOV R1, #\val1
    MOV R2, #\val2
    ADD R0, R1, R2
.endm
.macro multtwo val1, val2
    @ При выходе в R1, R2 содержатся val1, val2
    @ В R0 лежит накопленный результат

```

```

MOV R1, #\val1
MOV R2, #\val2
MLA R0, R1, R2, R0
.endm

```

Программу 18.4 можно сохранить как обычный исходный файл с расширением `.s`. Нет необходимости собирать и связывать его, т. к. это будет сделано, когда он окажется включен в основную программу при вызове. Достаточно того, чтобы имя файла, используемое во включении (вторая строка программы 18.5), совпадало с именем, которое использовалось для сохранения файла определения макроса (а еще они должны лежать в одной папке).

ПРОГРАММА 18.5. Проверка подключения макроса

```

/* Тестирование внешних макросов */
#include "Prog18d.s"
.global _start

_start:
    MOV R0, #0

_add:
    addtwo 3, 4

_mult:
    multtwo 2, 2

_exit:
    MOV R7, #1          @ выход через системный вызов
    SWI 0

```

Соберите и запустите файл `Prog18-5.s` в обычном режиме — вы должны получить ответ: 11.

Дизассемблировав исполняемый файл через GDB, вы получите вывод, похожий на приведенный в листинге 18.3.

Листинг 18.3. Дизассемблированный вывод программы 18.5

```

0x10054  <_start>:      mov     r0, #0
0x10058  <_add>:          mov     r1, #3
0x1005c  <_add+4>:        mov     r2, #4
0x10560  <_add+8>:        add     r0, r1, r2
0x10064  <_mult>:         mov     r1, #2
0x10068  <_mult+4>:       mov     r2, #2
0x1006c  <_mult+8>:       mov     r0, r1, r2, r0
0x10070  <_exit>:        mov     R7, #1

```

```
0x10074  <_exit+8>:          svc      0x00000000
0x10078:  Cannot access memory at address 0x10078
```

При сборке можно добавить в исходный файл дополнительные метки. Тогда, используя GDB, особенно в сборке с несколькими макросами, вам станет проще ориентироваться в разделах кода и ссылках на них.

Команды ARM для загрузки 0 в регистр R0 были приведены в начале кода. Они выполняются, даже если в этом нет необходимости, т. к. в результате макроса `addtwo` содержимое R0 все равно будет перезаписано. Результат процедуры `addtwo` переносился в программу `multtwo` и накапливался.

Опять же, здесь мы передали макросам статические значения, чтобы проиллюстрировать сам процесс того, как макрос ведет себя при сборке. Но можно также для передачи значений в такие макропрограммы использовать память, если мы не знаем, какими будут используемые значения на момент создания макроса. Это можно сделать и с помощью стека, используя методы, описанные в предыдущей главе. Однако будьте осторожны при использовании фреймов стека в макросах, поскольку лавина ошибок в настройке стека может иметь катастрофические последствия для ваших данных и управления программой.

СОВЕТ

Если в вашем коде есть несколько меток и вы не уверены, где искать начало, а где конец, тогда можно упростить дизассемблирование кода подобным образом:

```
disassemble _start, _exit+8
```

У меня это работает, поскольку я стараюсь быть последовательным в вопросах входа и выхода из кода ассемблера.

Многие из более крупных программ в этой книге могут быть сконструированы и реализованы как макросы. Я показал вам лишь несколько.

Пока что все наши программы представляли собой единый линейный файл, чтобы его было легче читать и понимать. Но с этим мы уже освоились и готовы идти дальше.

19. Работа с файлами

Файлы весьма важны для работы вашей Raspberry Pi. Практически все, что вы делаете, так или иначе связано с использованием файлов. Raspberry Pi OS в основном как раз и занимается управлением файлами. У этой системы имеется инфраструктура, позволяющая программам взаимодействовать с ней и выполнять большинство операций с файлами. Это могут быть самые разные операции: от создания файлов и открытия и закрытия файлов до многих других операций, наличие которых мы привыкли принимать как должное.

В *главе 8* мы видели, как можно взять строку текста и изменить ее или преобразовать из верхнего регистра в нижний. Сам текст ASCII у нас указывался в виде строки как часть самой программы. То есть мы знали, где находится строка, поскольку ей соответствовала именованная метка. А что, если бы нужная нам информация находилась в файле на SD-карте или USB-накопителе?

Файлы являются фундаментальным элементом всех компьютерных операций, особенно в ОС Raspberry Pi (Raspbian). В этой главе мы рассмотрим, как создавать, открывать, закрывать, читать и записывать файлы. В табл. 19.1 приведены системные вызовы, которые мы будем использовать с этой целью.

Таблица 19.1. Системные вызовы, используемые для работы с файлами в программе 19.1 (см. далее)

Операция	Описание	Вызов	№
Read	Чтение из файла	<code>sys_read</code>	3
Write	Запись в файл	<code>sys_write</code>	4
Open	Открыть или создать файл	<code>sys_open</code>	5
Close	Закрывать файл	<code>sys_close</code>	6
Create	Создать новый файл	<code>sys_create</code>	8
Sync	Очистить файл	<code>sys_sync</code>	118

Мы уже использовали некоторые из этих вызовов раньше, и большинству из них для работы требуется дополнительная информация. Имейте в виду, что номер сис-

темного вызова (приведенный в табл. 19.1) должен быть предварительно загружен в регистр R7, а нужные дополнительные сведения лежат в регистрах R0–R6. Не все регистры нужны для всех вызовов, но если вы не знаете точно, какие именно нужны, можете предположить, что все. Регистр R0 часто используется для возвращаемой информации — например, номера ошибки или результата.

В программе 19.1 показано, как использовать эти вызовы. Это может быть просмотр содержимого файла или чтение первых его 26 символов в буфер памяти перед записью в новый файл. В нашем примере предполагается, что файлы находятся в текущем каталоге или в том же каталоге, что и сама программа. А 26 символов здесь — это просто алфавит заглавных букв, который мы преобразуем в нижний регистр, а затем запишем его в новый файл. Программа также проиллюстрирует проверку некоторых файлов на наличие ошибок.

ПРОГРАММА 19.1. Создание файлов и доступ к ним

```

/* Создание файла и доступ к нему с помощью системного вызова */
/* Создание и открытие файла, чтение из файла, запись в файл */

.global _start

_start:
@ Открытие файла для чтения.
@ Предполагается, что файл находится в текущей папке.
@ В противном случае генерируется сообщение об ошибке (error1).

    LDR R0, =inputFile      @ адрес имени файла
    MOV R1, #o_rdonly      @ флаг "только для чтения"
    MOV R2, #s_rdwr
    MOV R7, #sys_open      @ вызов открытия файла
    SWI 0

    MOVS R8, R0            @ сохранение флага в R8
    BPL moveon            @ при положительном значении
                          @ переход на moveon

    MOV R0, #1             @ вывод на экран
    LDR R1, =error1        @ адрес сообщения error1
    MOV R2, #18            @ длина строки
    MOV R7, #4             @ запись кода
    SWI 0
    B finish              @ прерывание программы

moveon:
@ Создать и/или открыть файл для записи
    LDR R0, =outputFile
    MOV R1, #(o_create+o_wronly)
    MOV R2, #s_rdwr      @ права доступа

```

```

MOV R7, #sys_open      @ загрузка системного вызова 5
SWI 0                  @ вызов
MOVS R9, R0            @ сохранение флага
BPL readlinein        @ при положительном значении переход
MOV R0, #1             @ при отсутствии файла error2
LDR R1, =error2
MOV R2, #18
MOV R7, #4
SWI 0
B finish              @ прерывание программы

```

```

readlinein:           @ считывание строки из InFile.txt
MOV R0, R8            @ дескриптор файла R8 > R0
LDR R1, =inbuffer    @ расположение inbuffer
MOV R2, #alphabet    @ длина алфавита
MOV R7, #sys_read
SWI 0                 @ InFile >> InBuffer
MOV R10, R0           @ сохранение байт в R10
MOV R1, #0
LDR R0,=inbuffer
STRB R1, [R0, R10]   @ запись символа окончания в буфер

```

```

convertUpperCase:
PUSH {R8}
PUSH {R9}
MOV R8, #0            @ счетчик

```

```

loop:
LDR R0, =inbuffer    @ перемещение файла со входа на выход
LDRB R1, [R0, R8]    @ сравнение через ORR
ORR R1, R1, #0x20
LDR R0, =outbuffer
STRB R1, [R0, R8]
ADD R8, #1           @ инкрементный счетчик
CMP R8, #26          @ достигнута ли длина алфавита?
BNE loop             @ если нет, продолжаем
POP {R9}             @ восстановление файлов
POP {R8}

```

```

writebuffer:
MOV R0, R9
LDR R1,=outbuffer    @ адрес выходного буфера
MOV R2, #alphabet    @ длина алфавита
MOV R7, #sys_write   @ запись буфера после преобразования
SWI 0
MOV R1, #0

```

```

@ закрытие файла 'infile'
    MOV R0, R8
    MOV R7, #sys_fsync
    SWI 0
    MOV R0, R8
    MOV R7, #sys_close
    SWI 0

@ закрытие файла 'outfile'
    MOV R0, R9
    MOV R7, #sys_fsync
    SWI 0
    MOV R0, R9
    MOV R7, #sys_close
    SWI 0

finish:
    MOV R0, #0 @ код возврата 0
    MOV R7, #1
    SWI 0

.equ sys_open, 5
.equ sys_read, 3
.equ sys_write, 4
.equ sys_close, 6
.equ sys_fsync, 118
.equ o_rdonly, 0
.equ s_rdwr, 0666
.equ o_wronly, 1
.equ o_create, 0100
.equ alphabet, 26 @ длина файла в байтах

.data
inputFile: .asciz "infile.txt"
outputFile: .asciz "outfile.txt"
error1: .asciz "Input file error \n"
error2: .asciz "Output file error\n"
inbuffer: .fill (alphabet+1), 1, 65
outbuffer: .fill (alphabet+1), 1, 66

```

В последней части программы 19.1 есть несколько операторов `equ`, определяющих константы (речь идет о строках ниже метки `finish:`). В областях раздела `.data` хранится строковая информация. Эти области невероятно важны для программы, поскольку в них содержатся значения системных вызовов, значения флагов и имена файлов. Такие названия и определения меток являются общепринятыми и поэтому способствуют удобочитаемости программ. Впрочем, многие программисты созда-

ют свои собственные определения файлов макросов, которые затем можно использовать в директиве `.include` ради обеспечения согласованности, если будет нужно.

Для открытия файла системе требуется минимум три параметра, которые лежат в первых трех регистрах:

- ◆ R0 — указатель на имя файла, которое нужно открыть, хранится в виде строки ASCII с завершающим `null`;
- ◆ R1 — флаг, определяющий режим работы с файлом: чтение, запись, чтение/запись;
- ◆ R2 — значения режима доступа.

Первые строки программы открывают файл, из которого мы хотим считать данные. Системный номер вызова: 5. Если запрашиваемого файла не существует, генерируется ошибка. То есть имя файла важно для правильной работы, и его адрес мы загружаем в R0. В регистр R1 помещается код, определяющий, что мы делаем с файлом. В нашем случае это код 0, поскольку файл открыт только для чтения. В R2 определяются значения разрешений — в нашем случае значение 0666 (восьмеричное 666).

После возврата из вызова в R0 будет лежать подробная информация о возникших ошибках. Если это положительное число, значит, файл был успешно найден и открыт. Если значение отрицательное, файл не удалось либо найти, либо открыть. Продолжать работу в таком случае невозможно, поэтому в следующем фрагменте кода выводится строка сообщения `error1`.

Если все в порядке, значение (оно содержит дескриптор для файла) передается в R8, делается небольшой переход и работа программы продолжается с метки `moveon`.

Успешно открыв файл (`inputFile`), мы делаем то же самое для выходного файла (`outputFile`). Правда, в этом случае, если выходного файла не существует, системный вызов его создаст. Информация снова передается через вызов `sys_open`. Если все в порядке, происходит переход к метке `readlinein`, а в противном случае на экран выводится сообщение `error2` и программа завершается.

На метке `readlinein` мы сначала перемещаем дескриптор файла, ранее сохраненный в R8, обратно в R0. В регистре R1 лежит адрес, по которому мы сохраняем содержимое читаемого файла, а затем перед вызовом `sys-read` туда же помещается число, определяющее количество байтов, которые нам нужно считать. Байты считываются и сохраняются в определенном месте в памяти RPi. По возвращении из вызова количество записанных байтов оказывается в R0. Его мы перемещаем в R10, чтобы была сохранена информация о том, где мы закончили. Затем в конец буфера записывается ноль.

Разделы `convertupcase` и `loop` выполняют обработку информации, считанной из нашего открытого файла. Это может быть любой нужный вам алгоритм. Конкретно в нашем случае мы считываем каждый байт из `inbuffer`, выполняем операцию ORR между ним и значением `0x20` для преобразования значения в нижний регистр и сохраняем его обратно по адресу `inbuffer`. Поскольку для работы с содержимым нам

нужны регистры R8 и R9, их мы временно помещаем в стек. По завершении значения R8 и R9 восстанавливаются, и содержимое `inbuffer` записывается в открытый выходной файл (с использованием восстановленного R9 в качестве файлового дескриптора).

Наконец, оба файла закрываются функциями: `sys_fsync` и `sys_close`, опять же, с использованием соответствующих файловых дескрипторов, расположенных в регистрах R8 и R9.

Отметим пару моментов. Перед запуском программы создайте входной файл `infile.txt` с помощью подходящего текстового редактора и напишите в начале файла строку

```
ABCDEFGHIJKLMNORSTUVWXYZ
```

Сохраните файл в одной папке с программой 19.1. Убедитесь, что имя файла совпадает с указанным в программе в разделе `.data`.

В программе 19.1 метки `inbuffer` и `outbuffer` используются в качестве буферов для чтения, преобразования и записи информации. В обоих случаях мы поместим в буферы строки из букв A (64) и B (66), чтобы отличать их друг от друга. Если программа работает правильно, в них будут использоваться буквы верхнего и нижнего регистров соответственно. Это удобно, т. к. для отладки программы в любой удобный момент вы можете выгрузить эти разделы памяти, посмотреть, что в них лежит, и понять тем самым, на каком моменте выполнения находится программа.

Соберите и скомпонуйте программу 19.1. Убедитесь, что файл `infile.txt` находится в той же папке, а затем запустите программу. Если все в порядке, в папке появится файл `outfile.txt`. Откройте файл `outfile.txt` и проверьте результат.

Если в код закрадутся какие-либо логические ошибки, их можно будет отследить по содержимому файла `outfile.txt`. Одно из преимуществ использования в буферах памяти символьных строк, таких как буквы A и B, заключается в том, что если в файле `outfile.txt` появляется какой-то неверный результат, вы можете точно определить, в какой момент возникла ошибка.

Если программа работает правильно, вы при желании всегда можете заменить A и B на 0.

Права доступа к файлам

Ранее при открытии и чтении файлов нам нужно было задавать несколько значений, которые мы называли флагами и режимами. Мы помещали их в регистры R1 и R2 как часть процесса системного вызова и задавали их числами.

В листинге 19.1 приведена сводка каталога, содержащего файлы, необходимые и/или созданные в результате выполнения программы 19.1. Вы можете получить такой список, введя в консоль следующую команду:

```
ls -l
```

В результате отобразятся все лежащие в этом каталоге подкаталоги и файлы.

Листинг 19.1. Атрибуты файла

```
-rwxr-xr-x  1  pi  pi  2384  Sep  25  09:54  prog19a
-rw-r--r--  1  pi  pi  2496  Sep  25  09:54  prog19a.o
-rw-r--r--  1  pi  pi  2813  Sep  25  09:54  prog19a.s
-rw-r--r--  1  pi  pi   28   Sep  25  09:54  infile.txt
-rw-r--r--  1  pi  pi   26   Sep  25  09:54  outfile.txt
```

Первый столбец — это столбец с атрибутами файла. Эти атрибуты определяются десятью символами:

```
-rwxr-xr-x
```

Если это файл, то после первого символа (-) следующие девять символов идут наборами из трех символов и определяют, может ли файл быть прочитан (r), записан (w), выполнен (x) или нет (-). Три группы символов отвечают за разрешения «владельца», «группы» и «других пользователей».

Например, первый файл из этого примера может быть прочитан, в него можно записывать значения, и его можно выполнить. Если строка начинается с буквы d, то перед нами каталог (папка), а не файл.

Атрибуты файла `infile.txt`:

```
-rw-r--r-- 1 pi pi 28 Sep 25 09:39 infile.txt
```

Здесь мы видим, что первый символ в описании этого файла не d, а дефис (-), и знаем, что (`infile.txt`) — это файл, а не каталог. Затем идут разрешения владельца — `rw-`, т. е. владелец может читать и писать, но не может выполнять файл. Может показаться странным, что у владельца нет всех трех разрешений, но разрешение `x` тут и не требуется, поскольку это текстовый файл, который читается текстовым редактором и не является исполняемым. Разрешения группы установлены на `r--`, поэтому группа может читать файл, но не может каким-либо образом записывать/редактировать его — это, по сути, похоже на настройку чего-либо только для чтения. Те же разрешения применяются и ко всем остальным пользователям.

Сравните это с файлом `prog19a`, атрибуты которого:

```
-rwxr-xr-x 1 pi pi 2384 Sep 25 09:54 prog19a
```

Здесь первая буква — тоже дефис (-). И мы знаем, что это файл, а не каталог. Затем разрешения владельца: `rwx`, так что у владельца есть возможность читать, записывать и выполнять файл. Разрешения для других групп заданы так: `rx`, поэтому они могут читать и выполнять файл, но не могут ни записывать, ни редактировать его.

Как эти разрешения соотносятся с числовыми значениями, которые мы использовали в программе? В регистре, отвечающем за режим, мы указали трехзначное восьмеричное число.

Чтобы получить это трехзначное число, вам нужно подумать, какие разрешения должны быть у каждой группы. Каждая операция представлена числом:

r=4

w=2

x=1

Вернемся к атрибутам файла prog19a. Разбив его на три группы по три, мы получим:

Owner: rwx = 4+2+1 = 7

Group: r-x = 4+0+1 = 5

Other: r-x = 4+0+1 = 5

То есть нам нужно значение 0755 (помните, что ведущий 0 означает основание 8).

У нас также есть два текстовых файла:

Owner: rw--4+2+0=6

Group: r---4+0+0=4

Other: r---4+0+0=4

То есть значение для них получается: 0644.

20. Использование библиотеки *libc*

Ассемблер и компоновщик, которые мы использовали для написания и создания машинных программ, — это лишь небольшая часть компилятора GCC. В самом начале книги я упоминал, что компилятор GNU GCC — это компилятор C. Он принимает программы, написанные на языке программирования C, и преобразовывает их в машинный код. Даже круче — он берет исходный файл C и переводит его в исходный файл на языке ассемблера, который, в свою очередь, переводится в исполняемую программу машинного кода. Мы с вами работали лишь с последней парой процессов, но это лишь верхушка айсберга.

Хотя эта книга не о программировании на C, это не значит, что мы не можем использовать инструментарий языка C и компилятора GCC. Например, *libc* — стандартную библиотеку функций C. Как показано в предыдущей главе, мы можем задействовать системный вызов операционной системы для выполнения часто используемых операций: таких как ввод/вывод, управление памятью и операций со строками.

Использование функций языка C в ассемблере

У языка C нет встроенных средств для выполнения этих функций, но зато есть интерфейс, позволяющий получить к ним доступ без необходимости вникать во внутреннее устройство системного вызова. Кроме того, многие компоненты программ, которые вы, возможно, захотите написать, уже есть в готовом виде в *libc* или других библиотеках C, и их можно подключить к вашему коду. То есть существуют уже собранные библиотеки, сразу готовые к подключению во время компиляции. На рис. 20.1 схематически показано, где библиотека *libc* находится в общем интерфейсе.

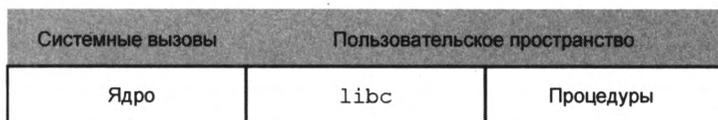


Рис. 20.1. *libc* и пользовательское пространство

Ядром в нашем случае является ОС Raspberry Pi. Область над ядром — это пространство пользователя. Здесь находятся наши файлы. Вспомните адреса, которые отображались, когда мы использовали утилиту GDB для дизассемблирования программ. Код библиотеки `libc` находится непосредственно поверх ядра, а код наших приложений находится поверх него. Хотя для нашей работы это не имеет особого значения, на схеме видно, насколько просто функциям `libc` подключиться к ядру. Обычно приложение, будучи часто написанным на C, использует интерфейс `libc` для доступа к системным вызовам. Реже программист напрямую обращается к системному вызову.

Основная причина использования системных вызовов напрямую вместо `libc` заключается в экономии места и потенциальном ускорении. Некоторые программисты также считают такой подход к программированию более «чистым» по сравнению с кодом, создаваемым после интеграции с `libc`. Кроме того, библиотека `libc` тоже занимает некоторый объем памяти, и большая часть ее функционала нам не нужна. Обычно это не проблема, но для шустрой и небольшой процедуры, где важна скорость и затраты памяти, от `libc` стоит отказаться, поскольку технически ваша собственная программа становится процедурой, использующей ресурсы, предоставляемые `libc`.

Пользователю важно иметь под руками экземпляр «Справочного руководства по библиотеке GNU C». Но не для того, чтобы изучать C (хотя и это неплохо, чтобы осознать его фундаментальное значение для системного и прикладного программирования), а ради информации о функциях, которыми можно воспользоваться. В описаниях функций сказано, какая информация им требуется, а какая информация возвращается. В этой главе мы рассмотрим несколько проработанных примеров по ним, опираясь на упомянутый документ. Справочное руководство по библиотеке GNU C можно скачать с веб-сайта GNU. Еще один быстрый способ найти информацию — онлайн-руководство. В командной строке введите:

```
man printf
```

и перед вами появится много информации об использовании и директивах указанной функции C. Команда `man` означает «руководство», и она выводит справку по указанной функции.

Структура файла исходного кода

Формат файла исходного кода, используемого с полным компилятором GCC, немного отличается от того, которые мы видели до сих пор. Создать его столь же просто, а компилировать его даже проще, поскольку нам не нужно выполнять этапы сборки и компоновки отдельно — это делается с помощью одной команды. Взгляните на *программу 20.1*. Это дополненная версия кода вывода строки, которая у нас называлась *программой 7.1*.

ПРОГРАММА 20.1. Структура исходного файла GCC

```

/* Вывод строки с помощью libc - новые требования          */
/* строка должна заканчиваться нулем, используется функция printf */

        .global main
        .func main
main:
        STMFd SP!, {LR}           @ сохранение LR
        LDR R0, =string          @ в R0 хранится указатель на метку string
        BL printf                @ вызов libc
        LDMFD SP!, {PC}          @ восстановление PC

_exit:
        MOV PC, LR               @ выход

.data
string:
        .asciz "Hello World String\n"

```

Прежде всего отметим, что определение `global _start` было заменено на `global main`:

```

        .global main
        .func main
main:

```

Эта структура важна, поскольку именно с ее помощью компилятор сообщает `libc`, где находится основная программа. Поскольку все процедуры `C` и `lib C` написаны как именованные функции, мы должны объявить эту основную часть нашего кода как функцию, а затем с помощью метки указать, где именно начинается функция. Именно это делают приведенные три строки (по сути, они выполняют ту же задачу, что и метка `_start` при использовании только компоновщика ассемблера, несмотря на добавление определения функции).

Две команды в начале и в конце основной функции сохраняют, а затем восстанавливают регистр ссылок в стеке. Об этих командах и их использовании мы говорили в *главе 17*. Строго говоря, здесь они не нужны, но сохранение регистра ссылок тем не менее является хорошим тоном. Так что будем придерживаться этого правила.

Функция `printf` из библиотеки `libc` выводит строку `asciz`, определенную в конце листинга программы. Сама `printf` — это не команда `C`, а функция, определенная в библиотеке, которую мы можем использовать. Это очень универсальная функция, и перед ее вызовом нужно лишь передать в регистр `R0` адрес строки. Функция `printf` всегда требует, чтобы строка оканчивалась нулем, поэтому используется директива `asciz` (и всегда нужно использовать именно ее).

Наконец, мы отказались от привычной функции выхода в пользу более простой команды `MOV`. Можно было бы применить и команду `SWI`, но полному компилятору

GCC подойдет именно такой метод выхода, более распространенный в мире программирования. Впрочем, вы можете задействовать метод `swt`, если для отображения возвращаемого содержимого вам хочется пользоваться командой `echo`.

Если теперь мы попробуем собрать и связать эту команду описанным ранее образом, ничего не выйдет, потому что у программы нет точки входа `_start`. Однако компиляцию с помощью GCC можно выполнить за один шаг:

```
gcc <Опции> <Целевой_файл> <Исходный_файл.s>
```

Так что для программы 20.1 компиляция выполняется следующим образом:

```
gcc -o prog20a prog20a.s
```

Теперь выполним программу с помощью следующей команды:

```
./prog20a
```

Исследование исполняемого файла

На этом этапе стоит посмотреть на полученный код с помощью GDB. Перекомпилируйте файл, добавив параметр `-g` для создания данных отладки:

```
gcc -g -o prog20a prog20a.s
```

а затем откройте дизассемблер:

```
gdb prog20a
```

Если вы сейчас дизассемблируете код командой

```
disassemble main
```

или

```
x/44i main
```

то по используемым меткам должны увидеть, что библиотечный компонент файла находится после метки `_exit`. Однако если вы просмотрите код, то увидите ответвления к адресам, расположенным ранее вашей основной точки входа. Изучите эти области. Вы увидите, что метки в программе говорят о том, что это код инициализации `libc`. Мы почти подключили `libc` к нашей программе! В коде программы вы, вероятно, заметите некоторые команды, с которыми мы ранее не встречались (расскажем о них в следующем примере программы).

Подобное изучение листингов программ отлично помогает понять больше в программировании машинного кода. Помните, что вы можете выполнить этот код пошагово и в любое время вывести значения регистров с помощью GDB, чтобы лучше понять, что и как происходит в программе.

Функция `printf` удивительно универсальна. В программе 20.2 показано, как передать в `printf` значения и использовать их при печати результатов.

ПРОГРАММА 20.2. Передача параметров в <code>printf</code>

```

/* Печать и передача строки с использованием libc */
/* используемые параметры функции в printf */

.global main
.func main
main:
    PUSH {LR}           @ псевдодиректива
    LDR R0, =string     @ R0 указывает на строку
    MOV R1, #10         @ первое значение в R1
    MOV R2, #15         @ второе значение в R2
    MOV R3, #25         @ результат в R3
    BL printf           @ вызов libc
    POP {PC}           @ возвращение исходного значения в PC

_exit:
    MOV PC, LR         @ выход

.data
string:
    .asciz "If you add %d and %d you get %d.\n"

```

Первое, на что следует обратить внимание, — это то, что команды входа и выхода раздела `main:` изменились. Мы используем команды: `PUSH` и `POP`. Это директивы компилятора, а не команды ARM, но они делают то же самое, что и используемые в *программе 20.1*. Работать с ними намного проще, т. к. вам не приходится слишком много думать о том, какой тип стека вы собираетесь использовать и в каком порядке задействуются регуляторы стека (но если вы решите воспользоваться другим ассемблером, директивы могут измениться и оказаться несовместимыми с вашим кодом. И почти наверняка вам придется скорректировать формат кода с помощью новой программы сборки).

Определение строки выполняется с тремя параметрами. Они обозначены символом `%`. Если вы скомпилируете и запустите эту программу, то получите следующий результат:

If you add 10 and 15 you get 25.

Из листинга *программы 20.2* мы видим, что эти три значения были помещены в `R1`, `R2` и `R3`. Используя функций `libc`, такие как `printf`, мы обычно следуем стандартному способу передавать и возвращать в них информацию (мы рассмотрим его в следующей главе, посвященной написанию функций).

В табл. 20.1 приведены некоторые параметры вывода функции `printf`. Список небольшой, но тем не менее открывает некоторые возможности для экспериментов с *программой 20.2*.

Таблица 20.1. Параметры вывода функции *printf*

Код	Назначение
<code>%d</code>	Вывод целого числа в десятичном формате со знаком
<code>%o</code>	Вывод целого числа в восьмеричном формате без знака
<code>%u</code>	Вывод целого числа в десятичном формате без знака
<code>%x</code>	Вывод целого числа в шестнадцатеричном формате без знака (строчными буквами)
<code>%X</code>	Вывод целого числа в шестнадцатеричном формате без знака (заглавными буквами)
<code>%s</code>	Вывод одного символа
<code>%%</code>	Вывод символа <code>%</code>

Ввод чисел с помощью функции *scanf*

Вы, вероятно, подумали, что функция `scanf` выполняет задачу, обратную `printf`, но я прощаю вам это заблуждение. Функция `scanf` принимает строку символов, введенную с клавиатуры, преобразует ее в числовое значение и сохраняет в памяти. Например, если при использовании `scanf` вы набрали:

```
255
```

то `scanf` сохранит двоичный эквивалент числа в памяти. В шестнадцатеричном формате оно выглядит так:

```
0xFF
```

Мы решили обсудить здесь именно эту функцию, а не ее обратного `printf` собрата, потому что на ее примере можно проиллюстрировать другой метод обмена данными с функцией библиотеки *libc*. Не все функции ожидают данные одинаковым образом. Вам нужно будет иметь это в виду, когда вы научитесь получать доступ к функциям *libc* и писать свои собственные функции.

Как и `printf`, функция `scanf` распознает множество различных форматов, и вы можете уделить время изучению и экспериментам с обеими. В следующем примере мы будем использовать целочисленные значения в формате `%d`, представленном ранее в *программе 20.2*.

Формат использования функции `scanf` следующий (но на C он не буквально такой):

```
scanf <Входящий_формат>, <Переменная>
```

или

```
scanf "%d", integernumber
```

Порядок использования функции `scanf`:

1. Объявляем переменную памяти, содержащую адрес строки форматирования. В нашем примере это будет строка `%d`.

2. Объявляем переменную памяти, содержащую адрес, куда должно быть помещено значение.

3. Освобождаем место в стеке для сохранения преобразованной строки ASCII.

Обратите внимание, как мы указываем, где лежат данные: мы передаем адреса их расположения. Это важно, т. к. означает, что для использования косвенных адресов мы должны объявить переменные, и в нашем случае у нас есть директива `.word` в текстовой области. Сами определения строк должны оставаться за пределами текстового раздела и находиться в сегменте данных кода. Также важно знать, что функция `scanf` сохраняет свой результат в стеке, поэтому, чтобы предотвратить его повреждение, нам нужно сдвинуть стек на одно слово, выделив в нем безопасное место. *Программа 20.3* поможет вам с этим разобраться (номера строк нужны лишь для пояснения программы, и при вводе указывать их не надо).

ПРОГРАММА 20.3. Чтение и преобразование числа с помощью функции `scanf`

```

1  /* Чтение числа с помощью scanf */
2  /* через регистры и стек          */
3
4  .global main
5  .func main
6  main:
7      PUSH {LR}
8      SUB SP, SP, #4           @ освобождение места в стеке
9      LDR R0, addr_format     @ получение адреса формата
10     MOV R1, SP              @ помещение SP в R1 и
11     BL scanf               @ сохранение записи в стеке
12     LDR R2, [SP]
13     LDR R3, addr_number
14     STR R2, [R3]
15     ADD SP, SP, #4
16     POP {PC}               @ восстановление PC
17
18 _exit:
19     MOV PC, LR @ выход
20
21 /* поскольку scanf нужны адреса строк, */
22 /* сборка их в текстовой области      */
23
24 addr_format: .word scanfomat
25 addr_number: .word number
26
27 .data
28 number:     .word 0
29 scanfomat: .asciz "%d"

```

Рассмотрим этот код. Строки 6 и 18 должны быть вам знакомы — они являются частью стандартной процедуры. В строке 8 мы сдвигаем указатель стека на четыре байта, чтобы освободить место для функции `scanf`. Перед вызовом `scanf` нам нужно поместить адрес `SP` в `R1` (строка 10) и адрес строки формата в `R0` (строка 9). Строка формата содержит информацию о значении, которое будет введено с клавиатуры и прочитано функцией `scanf`. За счет этого мы гарантируем правильное преобразование значения.

После вызова `scanf` (строка 11) преобразованное двоичное значение хранится в стеке, поэтому оно извлекается (строка 12), а адрес, где оно должно быть сохранено, помещается в строку 13. Затем значение сохраняется с помощью косвенной адресации (строка 14). Теперь мы должны вернуть на место указатель, добавив четыре дополнительных байта, которые мы изначально вычли из него (строка 15).

В строках с 25-й по 29-ю мы создаем адреса, указывающие на фактические данные, которые будут использоваться. Фактические данные находятся в подразделе `.data` (строки 27–29), и адреса этих двух мест хранятся в адресах длины слова в текстовой области, определенной строками 24 и 25. Таким образом, при сборке четырехбайтовое пространство, созданное в строке 24, будет содержать адрес строки `%d`. Это строка форматирования, с которой мы уже знакомы. Строка 25 создает место для адреса, куда будет помещен результат, возвращаемый функцией `scanf`.

Когда вы запустите эту программу, то не увидите приглашения к вводу. Просто введите число, например `255`, и нажмите клавишу `<Return>`. Приглашение появится. Скомпилировав программу с включенной отладочной информацией (опция `-g`), вы можете использовать `GDB` для пошагового выполнения кода и опроса регистров на каждом этапе. Это очень полезное упражнение, и то, что сейчас кажется запутанным способом, на самом деле после некоторого осознания оказывается проще простого!

В приведенной далее программе 20.4 мы добавим функционала, а именно — некоторое взаимодействие с помощью `printf`: запрос значения и последующую печать результата.

Обратите внимание, что в этой программе ожидается ввод информации в десятичном формате, но результат отображается в шестнадцатеричном формате — см. последнюю строку программы.

Вывод информации

Если у вас нет опыта работы с языком `C`, вам может быть интересно, как лучше всего получить всю информацию о функции и понять, как ее использовать. Хороший вопрос. Если честно, никакого официального ресурса на эту тему нет, и приходится работать методом проб и ошибок, а также искать информацию самим. Можно полазить по веб-сайтам и форумам. Еще вы можете посмотреть, что делает функция, создав для нее исходный код и проанализировав его с помощью `GDB`. По мере того как ваши знания о машинном коде `ARM` будут расти, этот вариант станет вам ближе, и в одной из следующих глав мы рассмотрим, как это сделать.

ПРОГРАММА 20.4. Объединение *scanf* и *printf*

```

/* Чтение числа с помощью функции scanf */
/* и вывод с помощью функции printf */

.global main
.func main
main:
    PUSH {LR}          @ использование псевдодирективы
    SUB SP, SP, #4     @ отступ на слово в стеке

    LDR R0, addr_messin    @ получение адреса сообщения
    BL printf             @ и вывод его

    LDR R0, addr_format    @ получение адреса формата
    MOV R1, SP             @ размещение SP в R1
    BL scanf              @ и сохранение записи в стеке

    LDR R1, [SP]          @ получение адреса scanf
    LDR R0, addr_messout   @ получение адреса сообщения
    BL printf             @ вывод всего

    ADD SP, SP, #4        @ подстройка стека
    POP {PC}             @ восстановление регистра PC

_exit:
    MOV PC, LR           @ выход

addr_messin: .word messagein
addr_format: .word scanfformat
addr_messout: .word messageout

.data
messagein: .asciz "Enter your number: "
scanfformat: .asciz "%d"
messageout: .asciz "Your number was 0x%X\n"

```

21. Пишем функции

Функции — это базовые строительные блоки, которые вы можете использовать для создания своих программ. У функции есть имя и некоторая цель, и пишется она таким образом, чтобы при каждом вызове возвращался результат. Функция принимает информацию и выдает результат в зависимости от нее. Она также может передавать информацию обратно в вызывающую программу. Все функции имеют определенную структуру, и если мы хотим написать функцию сами, то тоже должны ее соблюдать.

В идеале, прежде чем приступить к написанию программы, вам следует продумать ее структуру и, как мы видели в *главе 2*, попытаться спланировать программу в целом как набор небольших подпрограмм, вызываемых из основной управляющей программы. Каждая из этих подпрограмм тоже может, в свою очередь, вызывать более мелкие подпрограммы. При таком разбиении любой фрагмент кода, который используется многократно, можно было бы оформить как функцию, особенно если вы хотите позже использовать ее в других программах. Таким образом и создается эффективный многоразовый код.

Но написание такого кода сопряжено с определенными усилиями, потому что функция должна соответствовать определенным стандартам, определяющим условия входа и выхода. Очевидно, что длина функции должна быть больше нескольких строк, чтобы затраты на само использование функции стоили того.

Часто этап планирования выполняется программистом не так хорошо, как следовало бы. Мне нравится возвращаться к коду после написания и пытаться его реструктурировать. Это полезное занятие, и один из методов реструктуризации — посмотреть, что можно разбить на функции.

Стандарты функций

В предыдущей главе мы уже рассмотрели пару функций `libc`, а именно: `printf` и `scanf`, используемые для получения и возврата информации. Мы также увидели, что передавать информацию в эти функции можно через регистры `R0`, `R1`, `R2` и `R3`. Конкретные используемые регистры определяются тщательно разработанным стандартом *двоичного интерфейса приложения* (Application Binary Interface, ABI).

Именно он определяет, как должны выполняться функции. Смысл его разработки заключается в том, что если все программисты будут следовать стандарту и писать свои функции одинаково, то все они смогут использовать написанные друг другом функции (программисты на С пользуются стандартом AAPCS, проработанным еще подробнее). Насколько нам известно, влияние на код у обоих стандартов аналогично, и при желании вы можете почитать и о том и о другом в Интернете.

В табл. 21.1 представлено назначение всех регистров при вызове функций. Если обобщить эти сведения, получаем, что функция должна работать следующим образом:

- ◆ она может свободно изменять значения регистров R0, R1, R2 и R3 и находить в них информацию, необходимую для своей работы;
- ◆ она может свободно изменять регистры R4–R12 при условии, что перед возвратом в вызывающую процедуру значения будут восстановлены;
- ◆ она может изменять указатель стека при условии, что позже будет восстановлено значение, сохраненное при входе в функцию;
- ◆ она должна сохранять адрес, находящийся в регистре связи, чтобы программа позже могла правильно вернуться в вызывающую программу;
- ◆ она не должна в своей работе опираться на содержимое регистров состояния. Для функции состояние флагов N, Z, C и V неизвестно.

Итак, давайте теперь разберемся в этом чуть подробнее.

Таблица 21.1. Назначения регистров для функций

Регистр	Назначение	Сохранение значения
R0	Аргумент и результат	Нет
R1	Аргумент	Нет
R2	Аргумент	Нет
R3	Аргумент	Нет
R4	Регистр общего назначения	Да
R5	Регистр общего назначения	Да
R6	Регистр общего назначения	Да
R7	Регистр общего назначения	Да
R8	Регистр общего назначения	Да
R9	Регистр общего назначения	Да
R10	Регистр общего назначения	Да
R11	Регистр общего назначения	Да
R12	Регистр общего назначения	Да
LR	Адрес возврата	Нет
SP	Указатель стека	Да

Использование регистров

Стандарт гласит, что регистры `R0`, `R1`, `R2` и `R3` (именно в этом порядке) могут являться для функции источником входных данных. Но это только в том случае, если функции требуется четыре элемента данных. Если достаточно одного, он помещается в первый регистр `R0`, если двух, то второй помещается в `R1`, и аналогично для `R2` и `R3`. Если функции достаточно одного входного аргумента, то не имеет значения, что находится в других регистрах, поскольку они не будут использоваться. Если функция возвращает значение, оно всегда будет находиться в `R0` — по крайней мере первый байт. Поэтому мы можем использовать функцию `echo $?`, чтобы вернуть значение.

Кроме того, необходимо сохранить значения регистров с `R4` по `R12` включительно, чтобы, когда вызывающей программе обратно возвращается управление от функции, содержимое `R4–R12` вернулось в состояние до вызова функции. Но это не значит, что мы не можем использовать эти регистры. Если они нужны в процессе работы функции, то одно из первых действий, которое она должна делать (но не обязательно самое первое, как станет видно позднее), — это поместить их содержимое в стек, а затем восстановить его из стека. Для этого достаточно двух команд:

```
STMFDF SP!, {R4-R12}    @ сохранение регистров R4-R12
LDMFDF SP!, {R4-R12}    @ восстановление регистров R4-R12
```

Больше трех

Вы можете спросить, что произойдет, если требуется передать в вызываемую функцию более четырех элементов? Правило на этот счет следующее: мы можем снова изменить указатель стека (`SP`) при условии, что мы точно вернем его на место после завершения функции. Однако это правило не всегда строго верно, потому что, если нам нужно передать в функцию больше информации, управление `SP` должно осуществляться вызывающей подпрограммой, особенно когда объем данных неизвестен. Если функции всегда нужны дополнительные четыре элемента данных, то и сама функция может управлять `SP`, но в этом случае вам нужно быть очень внимательными.

Как вы помните, в *программе 20.2* мы вызывали функцию `printf` для отображения трех элементов данных, передавая данные через регистры `R1–R3`. В *программе 21.1* мы станем передавать вызывающей программе шесть значений. Строки в ее листинге для удобства пронумерованы.

Программа 21.1 почти идентична *программе 20.2*, за исключением фрагмента со строки 13, где мы начнем брать значения слов, хранящиеся в строках 31–33 включительно, и помещать их в стек. К тому времени, когда мы дойдем до строки 21, в `R0` у нас будет содержаться адрес строки, а в `R1`, `R2` и `R3` — значения 1, 2 и 3 соответственно. В стеке будут — сверху вниз — лежать числа 6, 5 и 4. И хотя числа эти всего лишь одноразрядные, для программы это слова со значениями, каждое из которых занимает четыре байта.

ПРОГРАММА 21.1. Передача значений функции через стек

```

1  /** Вывод строки с помощью printf **/
2  /** и передача ей параметров      **/
3  /** через регистры и стек        **/
4
5      .global main
6      .func main
7  main:
8      PUSH {LR}           @ использование псевдодирективы
9      LDR R0, =string     @ R0 указывает на строку
10     MOV R1, #1          @ Первое значение в R1
11     MOV R2, #2          @ Второе значение в R2
12     MOV R3, #3          @ Результат в R3
13     LDR R7,=value1     @ получение адреса параметра
14     LDR R8, [R7]       @ загрузка value1 в R8
15     PUSH {R8}          @ помещение в стек
16     LDR R7,=value2     @ то же самое для value2
17     LDR R8, [R7]
18     PUSH {R8}
19     LDR R7,=value3     @ то же самое для value3
20     LDR R8, [R7]
21     PUSH {R8}
22     BL printf          @ вызов libc
23     ADD SP, SP, #12    @ балансировка стека
24     POP {PC}           @ восстановление PC
25
26 _exit:
27     MOV PC, LR         @ выход
28
29     .data
30 string:                .asciz "Values are: %d, %d, %d and %d\n"
31 value1:                .word 4
32 value2:                .word 5
33 value3:                .word 6

```

Если вы запустите эту программу, то увидите на экране следующий результат:

Values are: 1, 2, 3 and 6

Дело в том, что мы сказали функции `printf` вывести дополнительное значение, и искала функция его на верхушке стека. Добавление еще пары `d%` в строку `printf` позволяет вывести еще два дополнительных значения. Если вы хотите, чтобы числа отображались в правильном порядке, нужно будет изменить порядок отправки чисел в стек.

Строка 23 тоже любопытная. Мы сдвинули `SP` на 12 байтов за счет трех команд `PUSH`. В этой строке `SP` сдвигается обратно на эти же 12 байтов, что обеспечивает

работоспособность системы. Того же самого результата можно было бы добиться с помощью трех команд POP, но показанный метод более изящен для восстановления исходного состояния, если вы сбрасываете в стек большой объем данных.

Если в процессе работы функции используются регистры, мы должны сохранить их содержимое (и восстановить его после завершения функции). Самый простой способ сделать это — поместить значения из регистров в стек, но это может привести к тому, что нужные функции данные тоже попадут в стек и не дойдут до нее. В таких ситуациях лучший выход — сохранить содержимое регистра в области памяти, которую вы отвели под рабочее пространство. В качестве альтернативы вы можете сначала поместить данные в стек, при необходимости «подвинув» указатель SP до того, как поместить туда параметры функции.

Если вы пишете функцию, которой через стек передается дополнительная информация, и требуется сохранение регистров, вы можете задействовать стек для всего и сразу, поскольку стеком вы управляете сами. Если ваша функция ожидала трех элементов в стеке, а вам нужно было сохранить R4, вы можете трижды обратиться к стеку, используя простое смещение:

```
LDR R4, SP+4 @ получение первого значения данных в стеке
```

Второй элемент будет лежать по адресу SP + 8, а третий — в SP + 12.

Помните, что эти команды не изменяют само значение SP, поэтому вы должны сбросить его по завершении, как мы уже говорили ранее (если сложно все это представить, нарисуйте стек прямо на бумажке и попробуйте определить, где находится каждый элемент данных).

Сохранение ссылок и флагов

При написании функций всегда следует помнить, что по завершении функция должна будет вернуть управление программой туда, откуда она его забрала. Поэтому крайне важно не потерять значение в регистре ссылок. Если в какой-то момент нужно будет вызвать другую функцию или подпрограмму, вы можете использовать для этого команды BL или BLX. В этом случае содержимое LR будет перезаписано и, следовательно, утрачено. Поэтому значение регистра ссылок нужно надежно припрятать в рабочем пространстве памяти или поместить в стек для последующего восстановления.

Сохранять флаги состояния необходимости нет. Функции обычно не работают с ними. Но функции может потребоваться наличие того или иного флага в момент возврата обратно в вызывающий код, и этот способ позволяет вернуть информацию из функции, учитывая, что один из стандартных способов возврата значения — это регистр R0. Например, при возврате из функции можно поднять флаг N, чтобы указать на наличие ошибки, а сам код ошибки поместить в R0.

Надежные процедуры вывода

Чтобы мы могли попрактиковаться в использовании функций, а также увидеть, как можно создавать библиотеки подпрограмм, которые можно будет использовать в различных своих проектах, идеально подойдут процедуры сортировки. Обработка и сортировка данных — отличный вариант занять компьютер полезным делом. Остаток этой главы мы посвятим созданию гибкой процедуры сортировки.

Надежные процедуры вывода на экран тоже нужны. Мы рассмотрели некоторые из них в предыдущих главах, поэтому здесь используется функция `printf`, которую можно подстраивать в соответствии с вашими потребностями. В следующем примере будет выведен на печать вектор значений ширины слов. *Программа 21.2* — это сама процедура вывода, а *программа 21.3* позволит нам написать текст, с которым она будет работать. Не удивительно, что нам потребуется библиотека `libc`, поскольку в программе используется функция `printf`. Поэтому сборку нужно будет выполнить с помощью `GCC`. Для работы подпрограмме необходим адрес вектора, содержащего элементы слова, количество элементов и пара указателей, чтобы подпрограмма могла отслеживать свое положение в векторе.

ПРОГРАММА 21.2. Вывод вектора слов

```

/* printf - Эта процедура печатает вектор слов          */
/* R0 = адрес вектора                                  */
/* R1 = количество элементов вектора                  */
/* R2 = строка указателя для печати первого элемента */
/* R3 = строка указателя для печати следующих элементов */

.global _printf
.equ _wsize,4
.align 2
_printf:
    STMFDP SP!, {R4, R5, R6, R7, LR}
    CMP R1, #0           @ выход, если нет элементов
    BLE _last
    MOV R4, R0           @ сохранение параметров локально
    MOV R5, R1
    MOV R6, R2
    MOV R7, R3
    LDR R1, [R4], #_wsize @ загрузка первого элемента
    MOV R0, R6           @ адрес первой строки
    BL printf           @ вывод его
    SUBS R5, R5, #1     @ декрементный счетчик
    BEQ _last          @ выход, если ноль

_printf_loop:
    LDR R1, [R4], #_wsize @ загрузка следующего элемента
    MOV R0, R7           @ адрес следующей строки
    BL printf           @ вывод его

```

```

SUBS R5, R5, #1      @ декрементный счетчик
BNE _printw_loop    @ продолжение цикла, если больше
_last:
LDMFD SP!, {R4, R5, R6, R7, PC} @ восстановление значения
                                   @ и возврат

```

В приведенном далее тестовом коде (*программа 21.3*) после метки `values` идет список элементов — всего их 11. Их вы можете редактировать по своему усмотрению.

ПРОГРАММА 21.3. Проверка `printw`

```

/* Тестовая процедура printw */
.equ _size,4
.equ _items, 11
.global main
.align 2
.section .rodata
first: .asciz "Vector of words - values : %d"
rest: .asciz ", %d"
final: .asciz "\n"
values: .word 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60

.align 2
.text

main:
    LDR R0, =values
    MOV R1, =_items
    LDR R2, =first
    LDR R3, =rest
    BL _printw
    LDR R0, =final
    BL printf

    MOV R7, #1
    SWI 0

```

Предполагая, что оба исходных файла программ находятся в вашем текущем каталоге, вы можете собрать их следующим образом:

```
gcc -o testprint prog21b.s prog21c.s
```

Затем вы можете выполнить код с помощью команды

```
./testprint
```

Поскольку мы использовали оба файла в одной сборке, применять директиву `include` не требуется.

Воспользуемся *программой 21.2* для вывода результатов следующей процедуры.

Пузырьковая сортировка

Пузырьковая сортировка, иногда называемая *сортировкой по убыванию*, является простейшим алгоритмом сортировки, принцип работы которого заключается в многократном обмене местами соседних элементов, если они стоят в неправильном порядке. Алгоритм проходит по списку и повторяется до тех пор, пока список не будет отсортирован. Алгоритм назван «пузырьковой» сортировкой потому, что меньшие или большие элементы как бы «всплывают» вверх по списку. Комментарии в коде должны помочь вам понять, что происходит.

Этот простой алгоритм плохо работает в реальном мире и используется в основном в учебных целях. В жизни используются более эффективные алгоритмы: быстрая сортировка, временная сортировка или сортировка слиянием, и они уже встроены в популярные языки программирования, такие как Python.

В коде *программы 21.4* мы проверим пузырьковую сортировку. Тестовый набор (см. *программу 21.5*) состоит из 16 элементов и включает отрицательные числа со знаком. Затем процедура печатает последовательность до сортировки и после нее. Количество элементов вы можете изменить по своему усмотрению.

ПРОГРАММА 21.4. Процедура пузырьковой сортировки

```

/* Пузырьковая сортировка векторов слов          */
/* R0 = начало вектора элементов                  */
/* R1 = количество элементов для сортировки      */
/* R4 = текущий элемент                          */
/* R5 = внутренний счетчик                       */
/* R6 = флаг keep_going                         */
/* R7 = первый элемент                          */
/* R8 = второй элемент                          */
/* R9 = регистр обмена                           */

.global _bubble
.text
_bubble:
    STMFD SP!, {R4, R5, R6, R7, R8, R9, LR}
    CMP    R1, #1           @ должно быть > 1
    BLE   _exit            @ выход, если ничего не нужно делать

    SUB   R5, R1, #1       @ установка внутреннего счетчика
    MOV   R4, R0           @ установка текущего указателя
    MOV   R6, #0           @ регистр для обмена

_loop:
    LDR   R7, [R4], #size @ загрузка элемента
    LDR   R8, [R4]        @ и следующего элемента
    CMP   R7, R8          @ сравнение их
    BLE   no_swap         @ переход, если второй элемент больше

```

```

MOV   R6, #1           @ поднятие флага keep_going
SUB   R4, #size        @ сброс на первый элемент
LDR   R9, [R4]         @ загрузка слова по нужному адресу
STR   R8, [R4]         @ сохранение меньшего значения обратно
                                @ в память
STR   R9, [R4, #size]! @ завершение обмена

```

no_swap:

```

SUBS  R5, #1           @ декрементный счетчик
BNE   _loop           @ продолжение цикла, если алгоритм
                                @ не закончен

```

end_inner:

```

CMP   R6, #0           @ проверка, закончена ли работа
BEQ   _exit           @ выход из цикла, если флаг сброшен
MOV   R6, #0           @ сброс флага
MOV   R4, R0           @ сброс указателя
SUB   R5, R1, #1       @ сброс счетчика
B     _loop           @ и по новой

```

_exit:

```
LDMFD SP!, {R4, R5, R6, R7, R8, R9, PC}
```

.data

```
.equ size,4
```

ПРОГРАММА 21.5. Тест пузырьковой сортировки

```
/* Тест пузырьковой сортировки */
```

.global main

main:

```

LDR R0, =values
MOV R1, #items
LDR R2, =before
LDR R3, =comma
BL _printw
LDR R0, =new_line
BL printf
LDR R0, =values
MOV R1, #items
BL _bubble

LDR R0, =values
MOV R1, #items
LDR R2, =after
LDR R3, =comma

```

```
BL _printw
LDR R0, =new_line
BL printf
```

```
MOV R7, #1
SWI #0
```

```
.equ items,16
```

```
.data
```

```
before: .asciz "Order before sorting, values are : %d"
```

```
after: .asciz "Order after sorting, values are : %d"
```

```
comma: .asciz ", %d"
```

```
new_line: .asciz "\n"
```

```
values: .word 12, 2, 235, -64, 28, 315, 456, 63, 134, 97, 221,
-453, 190333, 145, 117, 5
```

Убедитесь, что исходный код *программы 21.2* находится в том же каталоге, что и *программы 21.4* и *21.5*, соберите файлы с помощью команды

```
gcc -o testbubble prog21b.s prog21d.s prog21e.s
```

и выполните с помощью команды

```
./testbubble
```

22. Дизассемблирование программ на C

Как отмечалось ранее, эта книга не посвящена программированию на C, но, углубляясь в ассемблер ARM во время работы на своей Raspberry Pi, вы неизбежно затронете и язык C. По крайней мере, вы, скорее всего, захотите узнать больше о функциях библиотеки `libc`, которые затем можно будет использовать в своих собственных программах. Возможно, вы захотите взглянуть на машинный код функций `libc`, посмотреть, как они работают, и извлечь из этого полезный опыт. Этот процесс называется *обратным проектированием* (reverse engineering), и он весьма важен в процессе разработки программного обеспечения, поскольку программисту всегда интересно посмотреть, как другие программисты решили ту или иную задачу, а также улучшить на основе этого свой результат.

Функции `libc` хорошо задокументированы с точки зрения программиста на C, а вот информации об их работе на уровне машинного кода не так много. Оно и понятно. Учитывая, что язык C относительно прост и программисту доступны буквально тысячи примеров программ (в руководствах, в Интернете и на форумах), проще простого написать программу на C, в которой вы реализуете некоторую функцию. Эту программу затем можно скомпилировать в исходный файл на языке ассемблера, а потом изучить, исследовать и доработать его для своих целей.

В GCC и GDB есть нужные для этого инструменты, а в этой главе рассказано о том, как это делается. Некоторые моменты окажутся для вас сложными, но когда вы ближе познакомитесь с тем, как GCC преобразует C в машинный код, генерируемый ими код тоже станет понятнее.

GCC — он как швейцарский нож

GCC чем-то похож на швейцарский нож — он должен быть в состоянии справиться с любой задачей и действовать при этом методично. Наше использование ассемблера и компоновщика в качестве автономных инструментов показывает, что GCC — это не монолитный инструмент, а некий контроллер для нескольких программ GNU, который запускает их в нужном порядке для получения результата. А лучше всего здесь то, что мы можем остановить этот процесс в любой точке процесса, в которой нам это нужно.

GCC позволяет скомпилировать программу C в исполняемый файл с помощью всего одной команды:

```
gcc -o tornado tornado.c
```

Эта команда возьмет программу на языке C с именем `tornado.c` и преобразует ее в исполняемый файл с именем `tornado`. Это делается в несколько шагов:

- ◆ препроцессор (CPP) — берет исходный код C и собирает информацию обо всех директивах `#define` и `#include`, чтобы получить список всех переменных и единиц информации и файлов, которые ему понадобятся;
- ◆ GCC — создает исходный код на языке ассемблера. Преобразование основывается на некотором наборе базовых правил, и компилятор создает необходимые разделы кода в виде набора «строительных» блоков, вызывающих функции в `libc`. Он эффективно организует ваши данные и информацию, используя правила работы функций, которые мы обсуждали в предыдущей главе, а затем вставляет вызовы функций в нужные места. Затем он собирает код, необходимый для обработки любой возвращаемой информации перед повторным запуском процесса. Как только этот процесс завершается, получается файл кода с расширением `.s`;
- ◆ ассемблер (AS) — берет файл исходного кода и преобразует его в объектный файл (`.o`), этот процесс мы также рассматривали ранее;
- ◆ компоновщик (LD) — берет файл объектного кода и добавляет к нему все необходимые дополнительные файлы и библиотеки. Опять же, этот процесс мы уже описывали ранее.

В главе 14 мы видели, что при использовании GDB созданный GCC окончательный код представляет собой не просто линейный набор кодовых строк, а интегрированный код, позволяющий получить решение для любой переданной ему программы на языке C!

GCC не позволяет получить надежный и хорошо продуманный машинный код. Если затраты памяти и скорость для вас важны, вам нужно вручную писать код на самом низком уровне. Сама операционная система Raspberry Pi (Raspbian) разработана в основном на C. Но критически важные ее области написаны прямо на ассемблере. Скомпилированный код можно оптимизировать на этапе исходного кода, и у GCC есть несколько вариантов автоматизации этой задачи. В книге это не рассматривается, но мы, глядя на исходный файл ассемблера, созданный для выполнения определенной задачи, будем оптимизировать его и отсекаем все лишнее, пока не останется только нужный код, выполняющий нашу задачу.

Простой фреймворк C

Структура программы на языке C достаточно проста. *Программа 22.1* демонстрирует использование функции `putchar()` для вывода на экран одного символа ASCII. В программе всего шесть строк.

ПРОГРАММА 22.1. Простая программа на языке С для вывода символа

```
#include <stdio.h>

int main()
{
    putchar('*');
    return 0;
}
```

Первая строка сообщает компилятору, что в сборку нужно включить файл `stdio.h`. Это файл, в котором написаны все стандартные интерфейсы ввода/вывода, включая функции вроде `printf` и нужную нам функцию `putchar()`. Эта часть файла будет обрабатываться упомянутым ранее препроцессором, как и любая строка, начинающаяся с символа решетки `#`.

Сама программа берет начало с функции под именем `main()`. Все функции начинаются с открывающей фигурной скобки `{` и заканчиваются закрывающей скобкой `}`. Все, что находится между открывающей и закрывающей фигурными скобками, считается частью функции. Функция `main()`, как и все функции С, возвращает какое-то значение. Тип возвращаемого значения для `main()` — `int`, т. е. эта функция после завершения возвращает операционной системе целое число. В нашем случае она возвращает целочисленное значение `0`. Возврат операционной системе значения можно использовать как индикатор успеха или неудачи, а можно также возвращать коды ошибок для описания причины отказа.

Надеюсь, вы уже поняли, что функция `main()` в приведенной программе на С связана с функцией `main()` в наших исходных файлах ассемблера (и в меньшей степени с функцией `start` в тех программах, которые мы рассмотрели в начале книги).

Далее идет суть программы — функция `putchar()`, — в которой мы говорим вывести на экран звездочку.

Программу 22.1 можно ввести с помощью текстового редактора (Vim или Geany), но у ее файла должно быть расширение `.c`:

```
prog22a.c
```

Чтобы скомпилировать эту программу на С в исполняемый файл и запустить его, выполните команду:

```
gcc -o prog22a prog22a.c
./prog22a
```

На экране должна появиться звездочка. Смотрите внимательно, ее легко не заметить!

Создание файла ассемблера

Учитывая то, что мы узнали в последних нескольких главах, вы могли уже догадаться, какой код на языке ассемблера получится для этой программы на С. Функ-

ция выводит один символ. Следовательно, мы ожидаем, что согласно стандарту AAPCS код ASCII для печатаемого символа будет загружен в R0. Никакой другой информации не передается. Подпрограмма завершается, возвращая значение 0, и его тоже стоит ожидать в регистре R0, опять же, в соответствии со стандартом. Поехали писать!

Чтобы создать код ассемблера из файла .c, используется директива `-S` (именно заглавная s), таким образом:

```
gcc -S -o prog22a.s prog22a.c
```

GCC создаст файл с именем `prog22-1.s`, в котором будет храниться код на ассемблере. Откройте его в текстовом редакторе, и вы должны увидеть строки кода, похожие на приведенные в листинге 22.1.

Листинг 22.1. Ассемблер, созданный GCC из программы C

```
.arch armv6
    .eabi_attribute 28, 1
    .eabi_attribute 20, 1
    .eabi_attribute 21, 1
    .eabi_attribute 23, 3
    .eabi_attribute 24, 1
    .eabi_attribute 25, 1
    .eabi_attribute 26, 2
    .eabi_attribute 30, 6
    .eabi_attribute 34, 1
    .eabi_attribute 18, 4
    .file "prog21a.c"
    .text
    .align 2
    .global main
    .arch armv6
    .syntax unified
    .arm
    .fpu vfp
    .type main, %function

main:
    @ args = 0, pretend = 0, frame = 0
    @ frame_needed = 1, uses_anonymous_args = 0
    push {fp, lr}
    add fp, sp, #4
    mov r0, #42
    bl putchar
    mov r3, #0
    mov r0, r3
    pop {fp, pc}
    .size main, .-main
```

```
.ident      GCC: (Raspbian 8.3.0-6+rpi1) 8.3.0
.section   .note.GNU-stack,"",%progbits
```

Здесь вы быстро найдете тело программы. Фактически из 30 с лишним строк ассемблера лишь несколько выполняют поставленную задачу. Всю дополнительную информацию можно отбросить, но для GCC она была важна на этапе преобразования. Выполняемое преобразование очень методично и работает одинаково для всех программ простым перебором. GCC создает рабочее пространство для каждой функции, с которой работает в процессе преобразования, и вынужден защищать это рабочее пространство. В целом это достигается путем разделения стека на области, используемые функциями, — *стековые фреймы* (понятие фрейма стека мы ввели в *главе 17*). Фрейм отслеживается с помощью *указателя фрейма* (Frame Pointer, FP), о котором мы также говорили в *главе 17*.

Указатель отмечает начало фрейма стека, и каждая функция создает свой собственный фрейм стека, используя его для управления своими локальными переменными. Таким образом, фрейм стека является важным элементом программы на языке C, а при создании файла ассемблера он уже не столь существен. Но само понимание принципов его работы для нас важно и позволяет деконструировать ассемблер, полученный из C, но об этом позже.

Вернемся к коду. На этом этапе рекомендуется создать резервную копию файла кода на ассемблере. Мы начнем отсекаать лишнее и менять код, создавая компактный исходный файл меньшего размера, который можно будет собрать и протестировать. Команда

```
cp prog22a.s prog22a.so
```

создаст копию файла с расширением `.so`, которое означает, что это оригинал кода.

Строительные блоки

Теперь наша цель — убрать из этого кода все лишнее, чтобы мы могли собрать и связать его напрямую и получить тот же результат, что дает программа на C. GCC создает код ассемблера из ряда базовых строительных блоков. Первые 15 строк содержат некоторую информацию. Например, тут есть директива, определяющая набор инструкций, для которого должен быть скомпилирован этот код (`armv6`), и конкретный модуль с плавающей запятой, который будет использоваться (`vfp`). Атрибуты `eabi` (бинарный интерфейс встроенного приложения) определяют стандартные соглашения для форматов файлов, типов данных, а также использование регистров и опций, имеющих или не имеющих в ЦП. Все эти настройки атрибутов для наших целей не важны, и их можно удалить.

Далее следует имя файла и определение раздела `.text`. Для нашей сборки эти строки тоже не нужны.

В результате у нас получился код, приведенный в листинге 22.2 (показан с номерами строк для удобства рассмотрения). Первые три строки определяют функцию

`main()`, но нам нужно будет немного отредактировать их для использования в нашей сборке. Затем идет пара комментариев (строки 4 и 5), в которых компилятор описывает, как реализовано управление фреймом стека. Опять же, они нам не понадобятся, и их можно удалить.

Строки 6 и 12 работают в тандеме и сохраняют адреса, содержащиеся в указателе фрейма (`fp`) и регистре ссылок (`lr`). Указатель фрейма нам нужен, и значение `lr` перед вызовом подпрограммы `putchar` нужно сохранить, поэтому нам подойдут команды `PUSH` и `POP`. Строку 7 также можно удалить, поскольку нам не нужно обрабатывать ничего, связанного с `fp`.

Листинг 22.2. Код ассемблера без лишних директив

```
1      .global main
2      .type    main,    %function
3  main:
4      @ args = 0, pretend = 0, frame = 0
5      @ frame_needed = 1, uses_anonymous_args = 0
6      push    {fp, lr}
7      add     fp, sp, #4
8      mov     r0, #42
9      bl     putchar
10     mov     r3, #0
11     mov     r0, r3
12     pop     {fp, pc}
```

Суть нашего кода находится в строках 8 и 9. Код ASCII символа звездочки (42) перемещается в регистр `R0`, а затем вызывается функция `putchar()`. В строках 10 и 11 мы помещаем 0 в `R0`. Не забывайте, что исходная функция на С должна возвращать ноль. Для наших целей эти две строки нам не нужны, и их можно удалить. Остается лишь код, приведенный в *программе 22.2*.

ПРОГРАММА 22.2. Окончательный вариант кода

```
      .global main
      .func main
main:
      PUSH {LR}
      MOV R0, #42
      BL  putchar
      POP {PC}
```

Этот код правильно соберется и запустится. В нашем случае исходная программа на С была проще некуда, но методика, использованная для создания и сокращения исходного кода на ассемблере, весьма надежна и может применяться практически для любых программ.

Пример функции *printf*

Мы уже исследовали использование функции `printf` в программах на языке ассемблера, и мы рассмотрим его снова, на этот раз с точки зрения C, поскольку этот метод позволяет понять, как создаются программы на C, а это полезно при изучении ассемблера. *Программа 22.3* — это написанная на C программа `hello world`. На первый взгляд, эта программа не сильно отличается от нашего предыдущего примера с `putchar`. Но разница есть.

ПРОГРАММА 22.3. Код программы `hello world` на C

```
#include <stdio.h>
int main()
{
    printf("hello world");
    return 0;
}
```

Вы можете преобразовать этот код в ассемблер с помощью команды

```
gcc -S -o prog22c.s prog22c.c
```

В листинге 22.3 показан полученный код ассемблера без начальных директив. Опять же, у вас может получиться не буквально идентичный результат, но в целом все будет так же.

Листинг 22.3. Код ассемблера для программы `printf` на языке C

```
.section    .rodata
.align     2

.LC0:
.ascii    "hello world\000"
.text
.align     2
.global   main
.type    main, %function

main:
    @ args = 0, pretend = 0, frame = 0
    @ frame_needed = 1, uses_anonymous_args = 0
    push    {fp, lr}
    add     fp, sp, #4
    ldr     r0, .L3
    bl      printf
    mov     r3, #0
    mov     r0, r3
    pop     {fp, pc}
```

```
.L4:
        .align      2
.L3:
        .word      .LC0
```

Учитывая наши знания о функции `printf` и рассмотренные ранее примеры, раздел `main`: программы будет простым — здесь нет ничего нового. Интересны нам другие места, а именно, отмеченные метками: `LC0`, `L4` и `L3`. Метка `L4` нам не нужна, равно как и директива `.align 2`, — метка точно находится на границе слова, поскольку она идет непосредственно после кода. Метка `L3` отмечает зарезервированное слово, где хранится адрес текста `hello world`, помеченного меткой `LC0`. Подобный метод мы использовали во время экспериментов с функцией `scanf` в *программе 20.3*, а вот в программе `printf` (*программа 20.1*) применен другой метод. Их стоило бы сравнить.

В качестве последнего упражнения вы можете попробовать скомпилировать пример `scanf`, т. к. в коде этой функции сочетается сразу несколько таких методов, а информация берется из стека. *Программа 22.4* — это то, что вам нужно. Если вы скомпилируете и запустите ее код, вам будет предложено ввести с клавиатуры число. Больше ничего лишнего тут не будет, т. к. нас интересует лишь простая функция.

Напоминаем, что функция `scanf` использует стек для хранения и передачи преобразованного числового значения, поэтому вам нужно будет управлять стеком и указателем фрейма через код ассемблера.

ПРОГРАММА 22.4. Код для работы с функцией `scanf`

```
#include <stdio.h>
int main()
{
    int myvariable;
        scanf("%d", &myvariable);
        return 0;
}
```

Переменные указателя фрейма

Необходимо сказать пару слов об указателе фрейма во время такого анализа кода. Если в исходных программах на C есть переменные, то для указания на эти значения будет использоваться указатель фрейма, поэтому при деконструкции фреймы очень нужны. Рассмотрим две строки ассемблера:

```
LDR R2, {FP, #-8}
MOV R0, R2
```

Первая строка показывает, что переменная находится по адресу `FP - 8`, и она перемещается в `R0`. Если в исходном коде C есть несколько переменных, вам нужно

будет путем поиска похожих строк кода определить, где каждая из них находится в стеке. В вашем случае у ассемблера не будет указателя фрейма, или ему будет присвоено значение, которое часто оказывается равным `SP`, поэтому вам придется переместить переменные в помеченные места (в ARM в качестве указателя фрейма обычно используется регистр `R11`, но в разных ОС возможны различия).

Дизассемблирование системных вызовов

В *главе 7* мы говорили о системных вызовах и о том, как можно взаимодействовать с реальными функциями операционной системы напрямую. Позже мы узнали о библиотеке `libc` и о том, как можно использовать ее встроенные функции вроде `printf`. Использование `libc` влечет за собой затраты памяти, с которыми вы, возможно, будете не готовы мириться в условиях ее дефицита, и тогда придется прибегнуть к автономным системным вызовам. В принципе, это нормально, но проблема в том, что информации о системных вызовах Raspberry Pi OS (Raspbian) крайне мало. На тему популярных системных вызовов вроде вывода строки на экран в Интернете примеров много. А вот если вы хотите создать каталог или список каталогов, все становится сложнее. Зато можно добыть информацию, если вы воспользуетесь здравым смыслом и немного займетесь реверс-инжинирингом.

В *приложении 3* приведен список первых 193 системных вызовов. Чаще всего используются именно они. На сайте, посвященном этой книге, также имеются ссылки на неофициальный сайт, где можно найти дополнительную информацию о системных вызовах. Еще один вариант — попробовать разобраться в этом самостоятельно!

Приведенные здесь примеры уже были рассмотрены нами ранее, и мы специально преобразовывали из C именно их, чтобы вы лучше поняли, как работает GCC в процессе компиляции. При изучении новых функций у вас будет меньше информации, однако GCC работает по определенным правилам, и процессы будут аналогичны. Информация передается в регистрах и через стек (и через указатель фрейма), поэтому обычно достаточно просто определить, что где лежит, а дальше дело пойдет.

Возможно, будет проще открыть второе окно редактора и вручную написать код на ассемблере, опираясь на код, полученный от GCC, а не редактировать его. Это особенно удобно, если в коде много обращений к данным, т. к. таблицы часто беспорядочно перемещаются по памяти.

23. Функции GPIO

ПРИМЕЧАНИЕ

Программам, приведенным в этой главе, могут потребоваться изменения в зависимости от версии вашей Raspberry Pi. В коде ассемблера в соответствующих местах главы будут приведены подробности о необходимых изменениях.

Порт GPIO был одной из важнейших особенностей Raspberry Pi с момента выпуска платы. Он устанавливался на все модели, причем на большинстве из них есть даже готовый разъем, позволяющий легко подключить плату и пользоваться ею.

Здесь мы узнаем, как подключить интерфейс GPIO к Raspberry Pi и как использовать машинный код для доступа к этому подключению, что позволит вам читать и записывать данные на отдельные его контакты. Эта книга не посвящена именно интерфейсу GPIO, но знание того, как он работает с Raspberry Pi, имеет основополагающее значение для понимания принципов программирования платы.

Многие комплекты для работы с GPIO, продаваемые в магазинах, поставляются со специальными библиотеками кода, которые тоже можно скачать и использовать. Обычно эти библиотеки написаны на Python или Scratch и прекрасно позволяют проверять работоспособность любых аппаратных соединений, если в какой-либо момент возникнут трудности с отладкой. Код, приведенный в этой главе, был написан на Raspberry Pi и протестирован на нескольких версиях Raspberry Pi.

Платы расширения и интерфейсы GPIO претерпевали немало изменений с момента выпуска первой версии Raspberry Pi, однако я видел, что программисты проявляют немалую изобретательность при их подключении. Для целей этой книги я использовал комплект CamJam/EduKit. Но подойдет и любой другой аналогичный интерфейс. Существуют и иные комплекты, такие как RasPi.io Breadboard Pi Bridge (рис. 23.1), — о них вы можете почитать здесь: www.rasp.io/bbpi.

Отображение памяти

В Raspberry Pi используется ввод/вывод с отображением памяти. Адреса, используемые для ввода/вывода, прекрасно известны, и в частности их можно посмотреть в соответствующей таблице данных Broadcom. В ранних моделях Raspberry Pi для

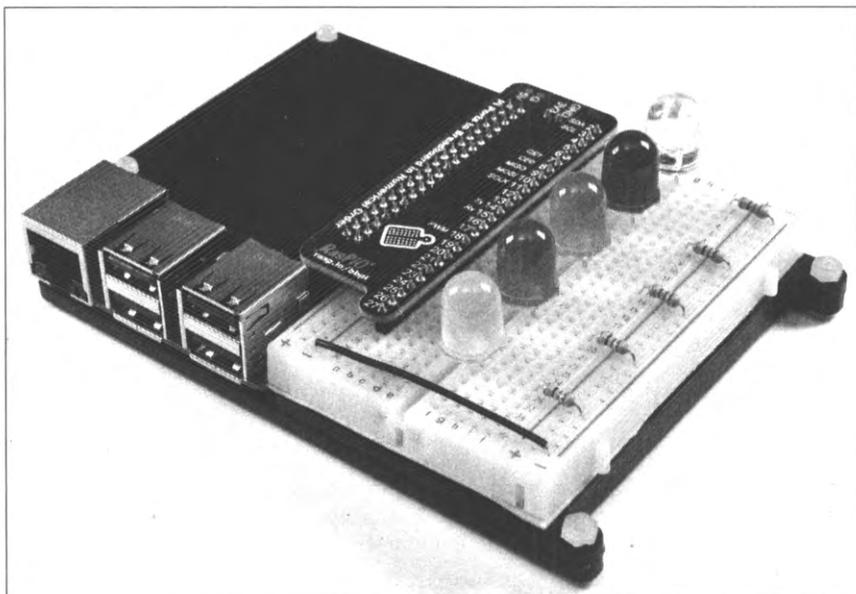


Рис. 23.1. Мост Breadboard Pi Bridge компании RasPi io

устройств ввода/вывода отводились адреса от $0x20000000$ до $0x20FFFFFF$. В более поздних версиях для устройств ввода/вывода зарезервированы адреса от $0x3F000000$ до $0x3FFFFFFF$. Имейте в виду, что это эти адреса зарезервированы не для одной лишь памяти GPIO, а для *всех* периферийных интерфейсов в целом.

Расположение этих блоков памяти, особенно в более поздних версиях платы, может измениться, как и пространство памяти, выделенное для порта GPIO. Очевидно, это может создать проблемы с совместимостью программ, напрямую обращающихся к порту GPIO. Учитывая также, что ОС Raspbian была переименована и фактически перезапущена под именем Raspberry Pi Operating System (ROS), возможно, хотя и маловероятно, что измениться могли некоторые из стандартных подпрограмм Linux.

Читая таблицы данных Broadcom, вы должны знать, что ядра ARM и графический процессор VideoCore (VC) на одном устройстве работают в *одном и том же пространстве памяти*, но по *разным адресам*. Вам необходимо четко различать адреса шины, адреса VC и адреса ARM. Эти адреса в каждой системе определены по-разному, что часто вносит путаницу в их работу. Адрес GPIO смещен от «базового адреса периферийных устройств» на $0x200000$. Именно столько нужно прибавлять к адресу для доступа. Информация, необходимая нам для примеров этой главы, будет приведена далее.

В ядре есть драйвер, расположенный в специальном символьном файле `/dev/mem`, который является зеркалом основной памяти. Как и в случае с любыми другими файлами, мы можем открыть его, прочитать из него байты, записать в него что-то, а затем закрыть. «Позиция» каждого байта в файле `/dev/mem` — это адрес байта в физической памяти. В принципе, запрограммировать устройство GPIO можно, открыв

файл `/dev/mem`, переместив указатель в желаемое место порта GPIO и записав соответствующее значение в выбранном месте.

Запись данных непосредственно в память ввода/вывода может быть рискованным делом, поэтому ROS не позволяет нам этого сделать. У системы есть встроенный механизм самозащиты, и только пользователи, обладающие правами `root`, могут получить доступ к памяти. Это позволяет гарантировать, что мы не сделаем ничего, что в противном случае могло бы вызвать блокировку или сбой операционной системы.

Существует и безопасный метод доступа к памяти GPIO. В нем используется аналогичная техника отображения памяти и драйвер в `/dev/gpiomem/`, который записывает изменения непосредственно в область управления GPIO через назначенный блок памяти, фактически являющийся отражением реальной памяти (эту память часто называют *виртуальной*). Таким образом, если мы внесем какие-либо изменения, которыми что-то испортим (например, система зависнет), достаточно будет выключить и снова включить питание, чтобы все сбросить. Этот драйвер гарантирует защиту от катастрофы, но, очевидно, работает он только с областью GPIO.

Поскольку в таблицах данных Broadcom можно найти базовый начальный адрес любой периферийной области ввода/вывода, мы можем позволить операционной системе самостоятельно вычислить адрес виртуальной памяти, и с этого момента все будет довольно просто.

В *главе 19* мы рассмотрели примеры вызовов файловой системы, необходимых для открытия и закрытия файлов с помощью системных вызовов. Чтение и запись в GPIO мало чем отличаются от работы с файловой системой, но на этот раз мы сделаем все немного проще, используя основные системные вызовы Linux.

Контроллер GPIO

У GPIO есть собственный контроллер, в котором расположен по меньшей мере 41 регистр. Первые пять из этих регистров предназначены для чтения и записи на 54 контактах GPIO. Эти контакты пронумерованы от GPIO 0 до GPIO 53, но лишь к некоторым из них можно подключиться через порт расширения GPIO. Дальнейшее зависит от размера разъема GPIO. Этот разъем имеет 26 контактов на ранних версиях Raspberry Pi и 40 контактов на более поздних (на рис. 23.3, приведенном в конце этой главы, схематически показана структура разъема GPIO).

Следует также отметить, что не все контакты контроллера GPIO работают одновременно на основном разъеме GPIO и их номера могут различаться у разных плат расширения. Важно предварительно ознакомиться с системой, которую вы используете, поскольку номера контактов тесно связаны с номерами, присвоенными BCM. При настройке используется несколько разных систем нумерации, и это может сбивать с толку. Я постараюсь помочь вам разобраться, что тут к чему. Важно понимать, что номер контакта разъема не соответствует номеру контакта GPIO.

Если у вас на рабочем столе открыто окно консоли, вы можете ввести команду `pinout` в приглашении, чтобы получить информацию о разъеме GPIO и других настройках.

Первые пять регистров контроллера GPIO, их имена и контакты, с которыми они связаны, приведены в табл. 23.1.

Таблица 23.1. Регистры GPIO и управление контактами

№	Имя	Смещение	Контакты
0	GPIO Function Select 0 (GPSEL0)	#0	0–9
1	GPIO Function Select 1 (GPSEL1)	#4	10–19
2	GPIO Function Select 2 (GPSEL2)	#8	20–29
3	GPIO Function Select 3 (GPSEL3)	#12	30–39
4	GPIO Function Select 4 (GPSEL4)	#16	40–49
5	GPIO Function Select 5 (GPSEL5)	#20	50–53

Каждый из этих регистров имеет ширину в 32 бита, и каждому контакту соответствуют три бита в каждом регистре. В столбце **Смещение** указано количество байтов, которые необходимо добавить к адресу GPIO. Подробнее об этом чуть позже. Обратите внимание, что под «контактом» здесь имеется в виду контакт GPIO, а не номер контакта разъема. В качестве примера мы рассмотрим вывод 22 GPIO, который будет использоваться в *программе 23.1*.

Судя по данным табл. 23.1, контакт 22 имеет смещение 8 байтов и, следовательно, находится в области GPSEL2. На рис. 23.2 показано, что с контактом 22 связаны биты 6, 7 и 8 в 32-битном адресе GPSEL2.

		Контакт 29			Контакт 28			Контакт 27			Контакт 26			Контакт 25			Контакт 24			Контакт 23			Контакт 22			Контакт 21			Контакт 20		
3	3	2	2	2	2	2	2	2	2	2	2	1	1	1	1	1	1	1	1	1	9	8	7	6	5	4	3	2	1	0	
1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0										

Рис. 23.2. Битовая ассоциация с GPSEL2

Возможно, изучая табл. 23.1, вы заметили, что используются не все биты в регистре 5. Это правильно. Заняты только первые 12 битов, а остальные отмечены как «зарезервированные». Биты 30 и 31 в каждом из других регистров тоже не используются.

Базовые адреса контроллера GPIO в операционной системе каждой модели Raspberry Pi свои (табл. 23.2).

Чтобы получить адрес любого регистра GPSEL, нам нужно добавить значение смещения (см. табл. 23.1) к сумме адресов колонок **Базовый адрес** и **Смещение GPIO** (см. табл. 23.2). Регистр GPSEL0 находится в начальной точке (поскольку смещение равно нулю). Чтобы получить доступ ко второму регистру GPSEL1, нужно добавить четыре к базовому адресу контроллера. Регистр GPSEL2 имеет смещение 8, поэтому его адрес будет равен либо

$$(0x3F000000 + 0x200000) + 8 = 0x3F200008$$

Таблица 23.2. Базовые адреса GPIO

Модель Raspberry Pi	SOC	Базовый адрес	Смещение GPIO
A+	BCM2835	0x20000000	0x200000
B	BCM2835	0x02000000	0x200000
Zero	BCM2835	0x20000000	0x200000
2	BCM2836	0x3F000000	0x200000
3	BCM2837	0x3F000000	0x200000
3+	BCM2837B0	0x3F000000	0x200000
4	BCM2711	0x3F000000	0x200000
400	BCM2711C0	0x3F000000	0x200000

либо

$$(0x20000000+0x200000) + 8 = 0x20200008$$

в зависимости от модели Raspberry Pi, с которой вы работаете.

Следовательно, чтобы получить доступ к контакту 22 разъема GPIO, нам нужно использовать биты 6–8 со смещением 0x08 относительно виртуального адреса, который мы получаем в нашей программе от вызова функции `pinmap`.

Вводы и выводы GPIO

Важно понимать, как назначаются эти биты, поскольку нам надо будет в тот или иной момент обращаться к ним, чтобы в связанных с ними регистрах что-то происходило. И еще нам нужно убедиться, что мы не искажаем и не изменяем значение других битов в регистре.

Работа с контактом — это двухэтапный процесс:

1. Задаем функцию конкретного контакта GPIO — на ввод или на вывод. Это достаточно сделать лишь раз, если ее не планируется менять по ходу программы.
2. Включить (Set) или выключить (Clear) контакты, как вам нужно. Для этого есть два отдельных набора регистров GPIO: GPSET и GPCLR.

ВНИМАНИЕ!

Не путайте регистры GPSEL и GPSET, т. к. они выполняют в контроллере GPIO совершенно разные функции.

Как мы уже говорили, сначала нужно выбрать функцию вывода GPIO, который мы будем использовать. В нашем примере это 22 (GPIO 22).

Чтобы назначить контакт в качестве ввода, мы должны поместить значение 0 в три связанных с ним бита: 000. Чтобы сделать этот контакт выводом, мы должны записать 1 в те же три бита: 111. Например, для настройки GPIO 22 на вывод нужно поместить 111 в биты 6, 7 и 8.

Для этого можно записать двоичное значение:

```
0b111
```

в нужные биты в GPSEL2. Разумеется, при этом мы должны беречь и не портить какие-либо другие биты, поэтому настройку лучше выполнять с помощью побитового оператора (другие битовые комбинации отвечают за другие функции, поэтому важно сделать все правильно).

Итак, мы рассмотрели настройку контакта на ввод или на вывод. Чтобы включить (поднять) или выключить (сбросить) контакт, нужно записать некоторые значения в другой регистр.

Существуют четыре регистра, работа которых связана с установкой и сбросом контактов (табл. 23.3).

Таблица 23.3. Регистры GPIO, предназначенные для включения и выключения контактов

№	Имя	Смещение	Контакты
7	GPIO Pin Set 0 (GPSET0)	#28	0–31
8	GPIO Pin Set 1 (GPSET1)	#32	32–53
10	GPIO Pin Clear 0 (GPCLR0)	#40	0–31
11	GPIO Pin Clear 1 (GPCLR1)	#44	32–63

С каждым контактом связан один бит установки и сброса. Чтобы включить контакт GPIO 22, мы должны записать 1 в бит 22 регистра GPSET0. Чтобы отключить тот же контакт, нам нужно будет записать 1 в бит 22 регистра GPCLR0.

Если записать 1 в бит 22 в GPSET0, а GPIO 22 настроен на ввод, ничего не произойдет. Если же настроить GPIO 22 на вывод, загорится подключенный к контакту светодиод. Последнее значение, записанное в GPSET0 или GPCLR0, в этом случае запоминается и используется при изменении состояния контакта.

Как видно табл. 23.3, смещение регистра GPSET0 составляет 28, а регистра GPCLR0 — 40, и это значение добавляется к базовому адресу контроллера GPIO для вычисления требуемого адреса.

В программе 23.1 приведен необходимый для этой настройки код, разбитый на несколько сегментов. В целом процесс выглядит следующим образом:

1. Открыть файл.
2. Отобразить память GPIO.
3. Установить контакт GPIO на вывод.
4. Включить вывод GPIO.
5. Выключить вывод GPIO.
6. Отменить отображение и закрыть файл.

К коду добавлены комментарии, поэтому отследить его работу легко, даже если со значениями смещений и регистров не все сразу понятно.

ПРОГРАММА 23.1. Файловый доступ к памяти GPIO

```

/* Доступ к GPIO с использованием отображения виртуальной памяти */
/* По предпочтительной методике используется вывод 22 GPIO */

@ Константы для создания карты памяти в ассемблере
.equ    gpiobase,0x3F000000    @ Периферийные устройства RPi 2, 3, 4, 400
.equ    offset, 0x200000      @ запуск устройства GPIO
.equ    prot_read, 0x1        @ возможно чтение
.equ    prot_write,0x2        @ возможна запись
.equ    readwrite,prot_read|prot_write
.equ    mapshare, 0x01        @ общие изменения
.equ    nopref, 0
.equ    pagesize, 4096        @ объем памяти

.equ    o_rdwr, 00000002      @ открытие для чтения/записи
.equ    o_dsync, 00010000     @ значения в восьмеричном формате
.equ    o_sync,04000000
.equ    o_allsync,o_sync|o_dsync
.equ    openflags,o_rdwr|o_sync @ флаги открытия файла

@ Константы для выбора функции
.equ    pinnumber,22          @ номер контакта (может различаться)
.equ    output, 1             @ контакт настроен на вывод
.equ    pinfield, 0b111       @ 3 бита
.equ    input, 0              @ контакт настроен на ввод

.equ    seconds,2             @ задержка

@ Константы для сброса и подъема контактов
.equ    pinbit, 1             @ 1 бит - 1 контакт
.equ    registerpins, 32
.equ    GPCLR0, 0x28          @ очистка сдвига регистров
.equ    GPSET0, 0x1C          @ установка сдвига регистров

@ адреса сообщений и значений
devicefile:    .word device
openMode:      .word openflags
gpio:          .word gpiobase+offset
openererror:   .word openstring1
memerror:      .word memstring2

@ Константы данных программы
.section .rodata
.align 2

```

```
device:      .asciz  "/dev/gpiomem"
openstring1: .asciz  "Didnt open /dev/gpiomem\n"
memstring2:  .asciz  "Didnt Map /dev/gpiomem \n"
```

@ Здесь начинается сама программа

```
.text
.align 2
.global main
```

main:

@ Открытие /dev/gpiomem для чтения/записи и синхронизации

```
LDR  R0, devicefile  @ адреса строк /dev/gpiomem
LDR  R1, openMode    @ флаги для доступа к устройству
BL   open            @ открытие
MOVS R4, R0          @ проверка ошибок
BPL  moveon1        @ при положительном значении
                                @ переход к moveon

LDR  R0, openererror @ ошибка
BL   printf
B    _exit          @ и окончание программы
```

moveon1:

@ Отображение GPIO в основную память

@ адрес скопированной памяти в R0

```
MOV  R4, R0          @ в R4 находится дескриптор файла
MOV  R8, R0          @ сохранение копии дескриптора файла
LDR  R9, gpio       @ адрес GPIO
PUSH {R9}           @ копирование стека для mmap
PUSH {R8}           @ дескриптор файла в стеке mmap
MOV  R0, #nopref    @ ядро само выбирает память
MOV  R1, #pagesize  @ получаем страницу 1 в памяти
MOV  R2, #readwrite @ чтение/запись памяти
MOV  R3, #mapshare  @ совместный доступ для всех процессов
BL   mmap           @ R0 - R3 + верхний адрес стека
MOV  R9,R0          @ сохранение отображенного адреса
CMP  R0, #-1        @ проверка ошибок
BNE  moveon2        @ если их нет, продолжение программы
LDR  R0, memerror   @ если ошибка, сообщение для пользователя
BL   printf
B    _exit
```

moveon2:

@ Выбор номера и назначения контакта.

```
MOV  R0, R9          @ программирование памяти
MOV  R1, #pinnumber  @ номер контакта (22)
MOV  R2, #output     @ назначение контакта (1)
MOV  R4, R0          @ сохранение указателя на GPIO
```

```

MOV R5, R1           @ сохранение номера контакта
MOV R6, R2           @ сохранение кода функции

@ вычисление адреса регистра GPFSEL и контакта
MOV R3, #10          @ делитель
UDIV R0, R5, R3      @ номер GPFSEL
MUL R1, R0, R3       @ вычисление остатка
SUB R1, R5, R1       @ для контакта GPFSEL

@ установка регистра контакта GPIO в памяти программ
LSL R0, R0, #2       @ 4 в регистре
ADD R0, R4, R0       @ адрес GPFSELn
LDR R2, [R0]         @ получаем значение из регистра
MOV R3, R1           @ умножаемый контакт
ADD R1, R1, R3, lsl #1 @ смещение на 3
MOV R3, #pinfield    @ контакт gpio (0b111)
LSL R3, R3, R1       @ сдвиг на позицию контакта
BIC R2, R2, R3       @ очистка поля контакта
LSL R6, R6, R1       @ сдвиг кода функции на позицию контакта
ORR R2, R2, R6       @ код функции
STR R2, [R0]         @ обновление регистра

@ setGPIOpin
@ все хорошо, включаем светодиод
@ нужен адрес пина и номер контакта
MOV R0, R9           @ получение адреса в памяти
MOV R1, #pinnumber   @ получение номера контакта (22)
ADD R4, R0, #GPSET0  @ указатель на GPSET в R4
MOV R5, R1           @ сохранение номера контакта

@ вычисление адреса регистра GPSET и поля контакта
MOV R3, #registerpins @ делитель
UDIV R0, R5, R3      @ номер GPSET
MUL R1, R0, R3       @ вычисление остатка
SUB R1, R5, R1       @ относительная позиция контакта
LSL R0, R0, #2       @ 4 байта в регистре
ADD R0, R0, R4       @ адреса GPSET

@ устанавливаем назначение контакта регистра GPIO в памяти
LDR R2, [R0]         @ получение регистра
MOV R3, #pinbit      @ один контакт (1)
LSL R3, R3, R1       @ сдвиг на позицию контакта
ORR R2, R2, R3       @ установка бита
STR R2, [R0]         @ обновление регистра

@ ожидание
MOV R0, #seconds     @ ожидание
BL sleep

```

```

@ сброс контакта GPIO
@ сброс контакта GPIO. Требуется адрес mmap и номер контакта
MOV R0, R9 @ получение адреса GPIO
MOV R1, #pinnumber
ADD R4, R0, #GPCLR0 @ указатель на регистры GPSET.
MOV R5, R1 @ сохранение номера контакта

@ вычисление адреса регистра GPSET и поля контакта
MOV R3, #registerpins @ делитель (32)
UDIV R0, R5, R3 @ номер GPSET
MUL R1, R0, R3 @ вычисление остатка
SUB R1, R5, R1 @ для относительной позиции контакта
LSL R0, R0, #2 @ 4 байта в регистре
ADD R0, R0, R4 @ адрес GPSET

@ установка назначения контакта GPIO в памяти
LDR R2, [R0] @ получение регистра целиком
MOV R3, #pinbit @ один контакт
LSL R3, R3, R1 @ сдвиг на позицию контакта
ORR R2, R2, R3 @ очистка бита
STR R2, [R0] @ обновление регистра

@ обратное отображение памяти GPIO
@ возврат отображенной памяти в исходное состояние
@ и закрытие файла
MOV R0, R5 @ память
MOV R1, #pagesize @ значение отображения
BL munmap @ обратное отображение

closeDev:
MOV R0, R8 @ дескриптор файла
BL close @ закрытие файла

@ и окончание программы
_exit:
POP {R8} @ восстановление исходного значения SP
POP {R9}
MOV R7, #1
SWI #0

```

Обратите внимание, что для сборки и генерации исполняемого файла вам нужно будет использовать следующую команду, иначе вы, скорее всего, получите ошибку:

```
gcc -march="armv8-a" -g -o gpio22 prog23a.s
```

В результате получится исполняемый файл с именем `gpio22`, если вы сохранили приведенный исходный файл как `prog23a.s`. Программа предполагает, что к `GPIO22`

подключен светодиод или нечто подобное, и она включит его (светодиод загорится), подождет две секунды, а затем выключит. Что бы ни было подключено, программа дополнительно создаст и откроет отображение виртуальной памяти, а затем вернет все как было по завершении (контакт GPIO 22 вроде как доступен на всех моделях Raspberry Pi, выпущенных к настоящему моменту).

Сборка кода

Программа получилась большой, поэтому давайте рассмотрим ее работу по частям.

Первые несколько блоков кода определяют константы и метки. Определения занимают приблизительно одну страницу файла или около того. Этот раздел заканчивается тремя определениями строк `asciz`. В комментариях и приведенном ранее описании подробно рассмотрена функция каждого из них.

Основная программа начинается с раздела `main`:

Чтобы открыть файл, который мы хотим отобразить, требуются два момента: адрес имени устройства, которое мы хотим открыть, и свойства, которые будут назначены файлу в момент его открытия. Их необходимо загрузить в `R0` и `R1` соответственно. Адрес строки

```
'/dev/gpiomem'
```

передается указателем адреса, указанным в `devicefile`. Далее выполняется быстрый переход к «открытию» в ядре Linux. При возврате к нашей программе в регистре `R0` будет находиться дескриптор файла или отрицательное число, указывающее, что произошла ошибка и файл не может быть открыт. В случае ошибки будет выведено соответствующее сообщение об ошибке и программа завершится. Этот механизм уже должен быть вам знаком исходя из знаний, полученных в предыдущих главах.

Если все в порядке, выполнение программы возобновляется на метке `moveon1`, где мы сохраняем копию файлового дескриптора в `R4` для использования программой и еще одну его резервную копию на будущее (`R8`). Нам также нужно сохранить копию адреса периферийного устройства, необходимого для доступа к GPIO (`0x3F200000`), и сохранить ее в `R9`. Оба эти элемента помещаются в верхнюю часть стека.

Функция `mmap` выполняет процесс отображения виртуальной памяти, которую мы будем использовать. Она принимает шесть следующих аргументов:

1. `R0` — адрес, по которому устройство должно быть отображено. Лучше всего позволить системе самостоятельно выбрать его, передав туда нулевое значение.
2. `R1` — объем необходимой для отображения памяти. Нам достаточно одной «страницы», размер которой равен 4096 байтов.
3. `R2` — значение, определяющее доступ к отображаемой памяти. Оно должно быть связано с защитой, настроенной в момент открытия файла. Должны совпадать возможности чтения и записи, а также возможности синхронизации отображаемой памяти с записанной в нее информацией.

4. R3 — значение, определяющее, может ли отображаемая память использоваться совместно с другими устройствами. Обычно лучше разрешить совместное использование, поскольку к GPIO могут обращаться и другие программы.
5. {R9} — физическая память устройства ввода/вывода.
6. {R8} — значение дескриптора файла для отображаемого устройства. Оно берется в момент открытия.

Последние два элемента в нашем списке помещаются в верхнюю часть стека. Вызову функции `mmar` требуется шесть аргументов, и, следовательно, функция обращается к стеку за теми двумя из них, которые не влезли в регистры R0–R3.

При возврате из вызова `mmar` адрес памяти, в которую выполняется отображение GPIO, находится в R0. Программа сохраняет его в R9. Если возвращается значение -1, оно указывает на ошибку, выдается сообщение об этой ошибке и программа завершается. В противном случае выполнение программы продолжается с метки `moveon2`, и она становится готова к выбору номера вывода GPIO и требуемой функции!

Следующий блок кода (следующие шесть строк сразу после метки `moveon2`) подготавливает регистры для программирования регистра GPFSEL и контактов. Эти значения изначально лежат в R0, R1 и R2, затем они копируются в R4, R5 и R6 соответственно, чтобы можно было использовать оригиналы.

Следующий раздел невероятно важен для вычисления адреса регистра и поля GPFSEL. Сюда входит команда `UDIV`, которую мы раньше не рассматривали, и это одна из причин, почему для сборки листинга необходимы специальные флаги. Команда `UDIV` делит одно беззнаковое 32-битное значение на другое беззнаковое 32-битное значение, возвращая 32-битный беззнаковый результат. Мы получаем частное, остаток отбрасывается, а результат округляется до следующего целого числа. Но именно остаток дает нам информацию о контакте GPFSEL, поэтому его нам также необходимо вычислить. Это делают три строки кода.

В нашем коде ассемблера частное вычисляется делением R5 на R3:

```
UDIV R0, R5, R3      @ беззнаковое деление R0 = R5/R3
R0=0x16/0x0A
R0=0x02
```

Константа 10, используемая в качестве делителя (0x0A) в R3, нужна для преобразования в десятичную систему (основание 10). Результирующее значение 2 представляется как деление 0x16 на 0x0A, или $22/10 = 2.2$, т. е. 0x02 округляется в меньшую сторону, а это и есть нужный нам номер GPSEL. Затем нам нужно вычислить остаток от частного, чтобы получить номер контакта GPSEL. Умножаем частное на делитель (теперь в R0):

```
MUL R1, R0, R3      @ R1 = R0 * R3
R0=0x02*0x0A
R1=0x14
```

Затем вычитаем результат в R1 из значения в R5:

```
SUB R1, R5, R1      @ R1 = R5 - R1
R1=0x16-0x14
R1=0x02
```

В результате получаем номер GPSEL и контакт GFSEL, после чего можно настроить регистр функции вывода GPIO. Важно учесть, что в регистре четыре байта (одно слово), поэтому значение нужно сдвинуть влево на две позиции:

```
LSL R0, R0, #2
R0=0x02,<<2
R0=0x08
```

Эта строка дает смещение 0x08, которое добавляется к базовому адресу, возвращаемому `mmap` (обратите внимание, что такой адрес получен на RPi 3B с 1 Гбайт памяти. Вероятно, у других конфигураций RPi он окажется иным, но вы можете не обращать внимания на этот адрес, поскольку система вычислила его сама). Вернитесь к табл. 23.1, и вы увидите, что смещение GPFSEL2 равняется 8. Вот его расчет:

```
ADD R0, R4, R0
R0=0x76FFF8000+0x08
R0=0x76FFF8008
```

Затем мы можем прочитать содержимое по адресу R0 в регистре R2 — в нашем случае это 0x40:

```
LDR R2, [R0]
R0=[0x76FFF8008]
R2=0x40
```

Теперь номер контакта, ранее сохраненный в R1, перемещается в R3:

```
MOV R3, R1      @ перемещение R1 на контакт R3
R3=0x02
```

А затем сдвигается на три позиции:

```
ADD R1, R1, R3, LSL #1
R1=0x02+0x2,<<1
R1=0x06
```

Взгляните на рис. 23.2, и вы увидите, что контакт 22 равен 6.

Значение контакта GPIO попадает в регистр R3 — в нашем случае это число 7. Затем значение сдвигается влево на шесть позиций, давая 0x1C0:

```
MOV R3, #pinfield @ поле контакта gpio
R3=0x07

LSL R3, R3, R1      @ сдвиг на позицию контакта
R3=0x07<<0x06
R3=0x1C0
```

Команда BIC выполняет побитовое логическое И между инвертированным значением битового шаблона и значением регистра. В результате мы перед перемещением кода функции очищаем в регистре биты, указанные битовой комбинацией:

```
BIC R2, R2, R3
    R2=0x040 BIC 0x1C0
    R2=0
LSL R6, R6, R1
    R6=0x01 << 0x06
    R6=0x40
ORR R2, R2, R6           @ код функции
    R2=0x0 ORR 0x40
    R2=0x40
```

Сохраняем содержимое R2 в месте, указанном адресом в R0, — 0x76FFF8008:

```
STR R2, [R0]
    0x76FFF8008 = 0x040
```

Теперь мы можем включить контакт, используя адрес `pinmap` и номер контакта. Они копируются в R0 и R1 с предварительным сохранением:

```
MOV R0, R9
    R0=0x76FF8000
MOV R1, #pinnumber      @ 0x16= GPIO 22
    R1=0x16
```

Добавляем номер контакта к исходному адресу `pinmap`, чтобы указать на регистры GPSET:

```
ADD R4, R0, #GPSET0
    R4=0x76FF8000+0x1C
    R4=0x76FF801C
```

Сохраняем номер для дальнейшего использования:

```
MOV R5, R1           @ сохранение номера контакта
    R5=0x16
```

Перемещаем номер контакта (32) в R3:

```
MOV R3, #registerpins
    R3=0x20
```

Как и раньше, используем команду UDIV, чтобы вычислить номер GPSET, а затем определить оставшееся значение относительного положения:

```
UDIV R0, R5, R3       @ номер GPSET
    R0=0x16/0x20
    R0=0
MUL R1, R0, R3        @ вычисление остатка
    R1=0x0 x 0x20
    R1=0
```

```

SUB R1, R5, R1      @ относительная позиция контакта
                    R1=0x16- 0x0
                    R1=0x16
LSL R0, R0, #2     @ 4 байта в регистре
                    R0=0 << 2
                    R0=0

```

Выполняем сложение, чтобы получить адрес GPSET в виртуальной памяти:

```

ADD R0, R0, R4     @ адреса GPSET
                    R0 = 0 + 0x76FF801c
                    R0=0x76FF801c

```

Обращаемся к содержимому всего регистра по адресу, указанному в R0:

```

LDR R2, [R0]      @ получение целого регистра
                    R2=[0x76ff801c]
                    R2=0x6770696f

```

Загружаем в R3 константу выводного контакта (1):

```

MOV R3, #pinbit   @ один контакт
                    R3=1

```

Логически сдвигаем влево значение на содержимое, содержащееся в R1 (относительное положение контакта):

```

LSL R3, R3, R1
                    R3 = 1 << 0x16
                    R3= 0x400000

```

Выполняем логическое ИЛИ с содержимым в R2, чтобы получить новое значение:

```

ORR R2, R2, R3     @ установка бита
                    R2= 0x6770696f ORR 0x400000
                    R2= 0x6770696f

```

Обновляем регистр, сохранив его в виртуальной памяти:

```

STR R2, [R0]      @ обновление регистра
                    R2=0x6770696f
                    R0=0x76ff801c
                    0x76ff801c=0x6770696f

```

Теперь контакт будет включен (как и подключенный к нему светодиод, если он есть).

Тот же метод используется для отключения контакта и, следовательно, любого подключенного светодиода. Проработайте этот код сами, используя описанную здесь методику.

Для работы с другими выводами потребуется минимум изменений в этой программе. Просто измените значение, присвоенное `pinnumber` в разделе Константы для выбора функции, на 17 и выполните сборку заново. Остальное программа сделает за вас.

Код программы приведен в здесь виде единого цельного файла. Для простого чтения он великоват. Как только вы поймете работу программы, можно будет разделить ее на серию макросов — в *главе 18* мы уже говорили о них.

Другие функции GPIO

Напомним, что существует еще несколько регистров, которые входят в состав контроллера GPIO и которые можно использовать и программировать по той же методике. Как уже упоминалось, вам нужно иметь под руками спецификацию периферийного устройства BCM, из которой вы можете получить конкретную информацию о других регистрах, их функциях и о том, как вообще начать с ними работать. Вся эта информация есть. Зайдите на официальный сайт Raspberry Pi для получения дополнительной информации, и там же вы найдете ссылки для ее загрузки.

И последнее, что касается таблицы данных. Если вы посмотрите на устройство памяти в начале документа, вы увидите, что периферийные устройства ARM начинаются с адреса `0x7E000000`, тогда как в ОС Raspbian или Raspberry Pi на Raspberry Pi Zero и 1 они начинаются с `0x2000000`, а на Raspberry Pi 2, 3, 4 и 400 — с `0x3F000000`. У всех ОС реализованы свои собственные системы адресации памяти, переопределяющие адреса, на которые изначально настроен ЦП. Это дает возможность ОС реализовывать отображение виртуальной памяти — метод, который позволяет программам и приложениям использовать больше памяти, чем им фактически доступно путем использования памяти SD-карты или подключенного жесткого диска. С практической точки зрения, работая с таблицей данных Broadcom, вы должны помнить, что все адреса периферийных устройств, указанные в тексте, являются адресами шины и должны быть преобразованы в физические адреса. Так, начальный адрес контроллера GPIO равен `0x7E200000`, но мы в наших программах в зависимости от модели реализуем `0x3F200000` или `0x20200000`.

И, наконец, небольшое предупреждение. Контакты GPIO управляют множеством функций вашей Raspberry Pi, и при отсутствии должной осторожности вы можете вывести операционную систему из строя. Всегда сохраняйте свою работу, прежде чем пытаться впервые выполнить свой код.

Описание контактов GPIO

На рис. 23.3 приведена схема 40-контактного разъема, установленного на более поздних модулях Raspberry Pi. Первое, на что следует обратить внимание, — это то, что в ней реально содержатся только три столбца информации: для нечетных контактов — слева от столбца **Разъем**, расположенного в центре, и для четных — справа от него.

Для версии 2.0 26-контактный разъем и контакты с 1-го по 26-й разъема по-прежнему актуальны и имеют те же назначения. Версия 1.0 от нее отличается. Если у вас Raspberry Pi с 26-контактным разъемом, вам следует зайти на веб-сайт Raspberry Pi и определить, какая у вас версия. Как правило, оригинальная модель Pi 1 B была оснащена версией 1.0.

Биты	Регистр	Контакт	Разъем		Контакт	Регистр	Биты
+3,3 Вольт			1	2	+5 Вольт		
6–8	GPSEL0	GPIO 2	3	4	+5 Вольт		
9–11	GPSEL0	GPIO 3	5	6	«Земля» (0 Вольт)		
12–14	GPSEL0	GPIO 4	7	8	GPIO 14	GPSEL1	12–14
«Земля» (0 Вольт)			9	10	GPIO 15	GPSEL1	15–17
21–23	GPSEL1	GPIO 17	11	12	GPIO 18	GPSEL1	24–26
21–23	GPSEL2	GPIO 27	13	14	«Земля» (0 Вольт)		
6–8	GPSEL2	GPIO 22	15	16	GPIO 23	GPSEL2	9–11
+3,3 Вольт			17	18	GPIO 24	GPSEL2	12–14
0–2	GPSEL1	GPIO 10	19	20	«Земля» (0 Вольт)		
27–29	GPSEL0	GPIO 9	21	22	GPIO 25	GPSEL2	15–17
3–5	GPSEL1	GPIO 11	23	24	GPIO 8	GPSEL0	24–26
«Земля» (0 Вольт)			25	26	GPIO 7	GPSEL0	21–23
Не подключается			27	28	Не подключается		
15–17	GPSEL0	GPIO 5	29	30	«Земля» (0 Вольт)		
18–20	GPSEL0	GPIO 6	31	32	GPIO 12	GPSEL1	6–8
9–11	GPSEL1	GPIO 13	33	34	«Земля» (0 Вольт)		
27–29	GPSEL1	GPIO 19	35	36	GPIO 16	GPSEL1	18–22
18–20	GPSEL2	GPIO 26	37	38	GPIO 20	GPSEL2	0–2
«Земля» (0 Вольт)			39	40	GPIO 21	GPSEL2	3–5

Внешний край платы

Рис. 23.3. Назначения контактов Raspberry Pi GPIO

Пояснения к схеме:

- ◆ Столбец **Разъем** — это физический разъем вашей Raspberry Pi, а цифры с 1 по 40 соответствуют его 40 контактам. Эти номера часто называют *номерами контактов*, каждый из них выполняет в Raspberry Pi определенную функцию, описанную в столбцах слева и справа от столбца **Разъем**.
- ◆ В столбцах **Контакт** в большинстве случаев указан номер вывода GPIO (FSEL). То есть вывод GPIO 22, например, подключается к контакту 15 разъема. В некоторых случаях такой вывод выполняет функцию питания или заземления. Так, на контакте 1 разъема имеется напряжение 3,3 В, а контакт 6 является заземлением. Два вывода с пометкой **Не подключается** на самом деле являются GPIO 0 и GPIO 1, но зарезервированы для других функций.

Вывод GPIO может быть установлен на высокий (3,3 В) или низкий (0 В) уровень. Если он обозначен как ввод, с него также считывается высокий (3,3 В) или низкий (0 В) уровень. Настройка осуществляется за счет использования внутренних подтягивающих или понижающих резисторов. У выводов GPIO 2 и GPIO 3 имеются встроенные подтягивающие резисторы, а у других выводов функция настраивается программно.

- ◆ В столбце **Регистр** указан регистр выбора функции (см. табл. 23.1). В нашем случае используются GPSEL 0, 1 и 2.
- ◆ В столбце **Биты** показаны три бита в соответствующем регистре GPSEL, которые управляют выводом GPIO. Взгляните, например, на выводы разъема 11 и 13. С ними связаны биты 21–23, которые находятся в регистрах GPSEL1 и GPSEL2.

Выводы GPIO не только являются устройствами ввода/вывода, но и могут использоваться для альтернативных и более сложных функций. Например, на всех выводах доступна программная широтно-импульсная модуляция (ШИМ), а вот аппаратная ШИМ доступна только на выводах GPIO 12, GPIO 13, GPIO 18 и GPIO 19. Последовательная передача (TX) осуществляется через GPIO 14, а прием (RX) — на GPIO 15. Существуют и другие функции, описание которых вы найдете на веб-сайте Raspberry Pi.

24. Числа с плавающей точкой

В своей повседневной работе с компьютерами мы воспринимаем само существование чисел с плавающей точкой как должное. Ну и правда, какая польза от компьютера банку, если на нем нельзя посчитать дробные числа? Когда мы используем электронные таблицы, калькуляторы и даже некоторые текстовые редакторы, сама возможность выполнять простые вычисления с точностью до нескольких знаков после запятой кажется нам вполне обычным делом. А как насчет языка ассемблера? Ведь все действия с числами, которые мы успели рассмотреть (и на самом деле их не так много), были связаны с целыми числами, не имеющими дробной части.

Работа с числами с плавающей точкой требует значительных вычислительных усилий, которые выполняются не микросхемой ARM, а так называемым *сопроцессором*. Как мы увидим в *главе 30*, чип ARM, используемый в Raspberry Pi, является частью более крупной инфраструктуры, известной как SOC, или System-on-Chip (Система на кристалле). Она гораздо больше, чем сам чип ARM, и в числе прочего в ее состав входит некоторая аппаратная схема, которая занимается именно числами с плавающей точкой. У этого сопроцессора, также известного как VFP, или Vector Floating Point (Вектор с плавающей точкой), есть своя архитектура, включающая регистры и команды, позволяющие добавлять в программы на языке ассемблера операции с числами с плавающей точкой.

В следующих нескольких главах мы рассмотрим, как реализована архитектура работы с числами с плавающей точкой, и команды, позволяющие использовать вещественные числа в наших собственных программах. В результате изучения этих глав вы получите достаточно информации, чтобы работать с этими числами на практике и в любых задачах.

Архитектура VFP

Современное программное обеспечение, и в частности медиакодеки и графические ускорители, работает с большими объемами данных, размер которых меньше слова. Например, в аудиоприложениях часто используются 16-битные данные, а 8-битные данные стандартно применяются в графике и видео. При выполнении этих операций на 32-битном микропроцессоре получается, что некоторые части микропроцессора не задействуются, но при этом тем не менее потребляют электроэнергию. Что-

бы эффективнее использовать эту расходуемую впустую мощность процессора, привлекается технология SIMD (Single Instruction Multiple Data, Одна инструкция, несколько данных), в рамках которой одна команда используется для выполнения одной и той же операции параллельно на нескольких элементах данных одного типа и размера. Таким образом, процессор складывает не два 32-битных значения, а вместо этого одновременно выполняет четыре сложения 8-битных значений.

Создатели Raspberry Pi стараются использовать все преимущества микропроцессора ARM, но если вы внимательно следите за новинками в мире технологий, то заметите, что микропроцессоров ARM существует великое множество. Такое разнообразие затрудняет обеспечение совместимости, поскольку у каждого из этих процессоров свои особенности. По этой причине на момент подготовки книги мы ориентировались лишь на три версии ARM, а именно: ARMv6, ARMv7 и ARMv8 (тому есть и другие причины, но совместимость аппаратного и программного обеспечения — самое важное для Raspberry Pi).

Архитектура VFP2 реализована на чипе ARM v6, установленном на Raspberry Pi 1 моделей A, B, A+ и B+. На Raspberry Pi 2 и Raspberry Pi 3 установлен чип ARMv7/8, который поддерживает VFP4. VFP4 включает в себя VFP2, но набор команд в нем побольше. В этой главе мы разберемся с основами, присущими обеим версиям.

VFP поддерживает числа с одинарной и двойной точностью. Как следует из названия, двойная точность точнее, чем одинарная. Если конкретнее, то число с одинарной точностью занимает одно слово памяти (32 бита), а число с двойной точностью — два слова памяти (64 бита). Мы знаем, что ARM может свободно работать с 32-битными числами, и вот вопрос: можно ли из чисел с плавающей точкой одинарной точности получить столь же большие значения, что и в целых числах? Ответ — да, и не просто столь же большие, а намного бóльшие, т. к. механизм представления у них отличается. Далее приведены примеры чисел с плавающей точкой, или вещественных чисел:

```
0.2345
546.6735268
1.001011010
4E7
```

В последнем случае это число 4, умноженное на 10 в степени 7, т. е. 40000000, или 4×10^7 , где 7 — показатель степени. В числах с одинарной и двойной точностью значения кодируются так, чтобы у них был знаковый разряд, показатель степени и дробная часть (мантисса). Такой метод представления позволяет хранить и большие, и малые числа.

На рис. 24.1 показано, как хранятся в памяти числа с одинарной и двойной точностью. В случае чисел с двойной точностью составляющие его слова должны занимать соседние ячейки памяти и быть выровненными по словам.

Здесь:

- ◆ **З** (Знак) — для знака используется значение 0 или 1 для представления положительных или отрицательных значений соответственно. Знак хранится в старшем разряде числа.



Рис. 24.1. Устройство чисел одинарной и двойной точности

- ◆ **Степень** (показатель степени) — это значение, которое используется для сдвига мантиссы влево, чтобы получить таким образом нужное значение. Степень хранится между знаковым разрядом и мантиссой.
- ◆ **Мантисса** (дробная часть) — это число после точки, которое, очевидно, может быть представлено в двоичном виде и соответствовать вещественному числу. Мантисса нормализуется, т. е. сдвигается вправо, пока слева от точки не останется одна цифра. В числах с двойной точностью она занимает все младшее слово и часть старшего слова.

В числах с плавающей точкой одинарной точности мантисса занимает 23 бита (+1 для нормализованного целого числа), а показатель степени — 8 битов, т. е. значение степени может лежать в диапазоне от -126 до 127 . При двойной точности мантисса занимает 53 бита (+1 для нормализованного целого числа), а показатель степени — 11 битов, поэтому степень может принимать значения от -1022 до 1023 .

Для полноты изложения, отметим, что существует также третий тип представления чисел. Он называется NaN, что является сокращением от «Not a Number» (не число). Такое представление используется в особых случаях, когда значение не может быть представлено одинарной или двойной точностью. Это довольно интересный момент, особенно учитывая, что существуют два разных типа NaN. Если вам интересны подобные тонкости, почитайте об этом дополнительно.

Регистровый файл

Та же архитектура загрузки и хранения, что используется в микросхеме ARM, применяется и для работы со значениями с плавающей точкой в VFP. В сопроцессоре есть набор регистров, предназначенных специально для этой цели. Всего их 32, у них есть префикс *s* и нумерация от *s0* по *s31*. Эти регистры служат для хранения значений с одинарной точностью, поскольку все они имеют ширину в одно слово.

Для работы с числами двойной точности эти регистры можно объединять в пары, получая таким образом до 16 регистров шириной в два слова. Для них используется префикс D и нумерация от D0 до D15. На рис. 24.2 показан пример регистрового файла.

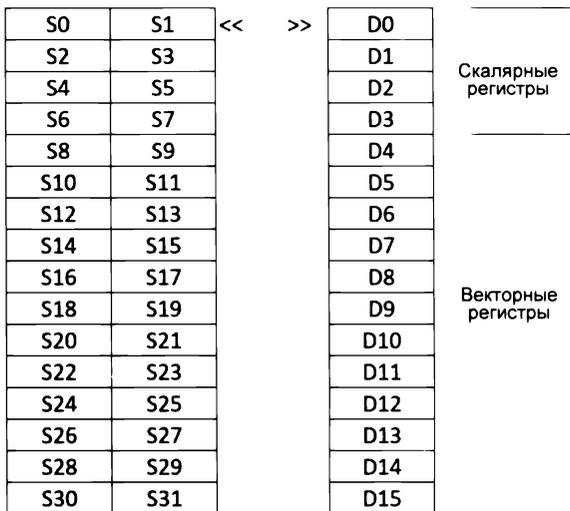


Рис. 24.2. Регистровый файл VFP. Регистры Sx можно использовать по отдельности или парами, создавая регистр двойной точности Dx

Следует понимать, что физически это одни и те же регистры, и, несмотря на возможность работы как с одиночными, так и с двойными регистрами, хранить можно всего одно значение за раз. Таким образом, регистры S0 и S1 можно использовать по отдельности для хранения двух значений с одинарной точностью или объединить в D0 для хранения значения с двойной точностью. При загрузке значения в D0 будет утрачено содержимое S0 и S1. Но вы можете хранить значения одинарной точности в S0 и S1 и значение двойной точности в D1, поскольку D1 состоит из S2 и S3.

Обратите внимание, что у сопроцессора нет никаких механизмов, которые сообщали бы вам, какое число в каком регистре находится. За этим придется следить самостоятельно — в конце концов, это машинный код. Вы тут босс. Имеющиеся регистры по назначению поделены на скалярные регистры (S0–S7 и D0–D3) и банки векторных регистров (S8–S31 и D4–D15). Принадлежность к одной из этих групп определяет способ доступа к ним, что мы рассмотрим в примерах позже.

Разумеется, у сопроцессора имеются команды, специально предназначенные для перемещения значений как с одинарной, так и с двойной точностью в память и регистры и из них, а также различные команды арифметических операций. Давайте сначала рассмотрим простой пример, в котором мы воспользуемся функцией `printf` для вывода числа с плавающей точкой. На этом же примере отметим несколько важных моментов.

Управление и вывод на экран

В программе 24.1 приведено несколько основных операций над числами с плавающей точкой, а также показана техника использования функции `printf` для вывода значения с плавающей точкой на экран.

Это весьма важный пример, т. к. использованная в нем методика позволит вам выводить результаты любых выполняемых операций. В первую очередь следует отметить, что вы не можете загрузить значение с плавающей точкой непосредственно в регистр. Это следует делать через косвенную адресацию. Эта концепция и способ структурирования кода должны быть вам уже знакомы, а пример как раз и приведен в программе 24.1.

ПРОГРАММА 24.1. Вывод значения с плавающей точкой с помощью функции `printf`

```

/* Вывод числа с плавающей точкой */
.global main
.func main

main:
    LDR R1, addr_value1      @ получение адреса value1
    VLDR S14, [R1]          @ перемещение value1 в S14
    VCVT.F64.F32 D5, S14    @ преобразование в 64-разрядный формат

    LDR R0, =string         @ в R0 помещается указатель на string
    VMOV R2, R3, D5        @ загрузка значения

bp:
    BL printf               @ вызов функции
    MOV R7, #1              @ выход через системный вызов
    SWI 0

addr_value1:
    .word value1

    .data
value1:    .float 108.65625
string:    .asciz "Floating point value is: %f\n"

```

В первой строке кода адрес значения `value1` загружается в регистр `R1`. В следующей строке он используется как косвенный адрес значения, загружаемого в `S14`. `VLDR` означает `Vector Load Register` — регистр векторной загрузки. В третьей строке это значение преобразуется в число двойной точности. Это связано с тем, что функция `printf` может выводить только значения с двойной точностью, а с одинарной — не может. Команда выглядит сложно, но ее на удивление легко читать и составлять, если вы знаете обозначения:

- ◆ `VCVT` — команда векторного преобразования;
- ◆ `.F64` — преобразуем в число двойной точности `Binary64`;

- ◆ .F32 — из Binary32 — числа одинарной точности;
- ◆ D5 — регистр назначения двойной точности;
- ◆ S14 — источник числа одинарной точности.

Здесь важно помнить порядок: сначала место назначения, потом источник.

В следующих трех строках мы готовим функцию `printf` к использованию. Как и раньше, регистр `R0` должен указывать на выводимую строку. Обычно для передачи дополнительных значений в `printf` задействуются регистры `R1`, `R2` и `R3`. Но мы можем хранить всего одно значение двойной точности в этих трех регистрах, поэтому обычно ограничиваются регистрами `R2` и `R3` (`R1` игнорируется). Команда `VMOV` перемещает `D5` в `R2` и `R3`. В выводимой строке используется директива `f%`, а функция `printf` уже знает, что эта директива указывает на значение с двойной точностью, и поэтому просто ищет его в `R2` и `R3` (другим функциям библиотеки `libc` можно передавать два значения с двойной точностью, поскольку при обычном использовании ARM для этой цели выделяются регистры `R0–R3`).

В разделе данных также видно, что для хранения значения с плавающей точкой используется директива `.float`.

Вывод нескольких значений с двойной точностью с помощью команды `printf` выполняется так же, как мы делали раньше. Другие подлежащие выводу элементы помещаются в стек. Но команды `PUSH` и `POP` или их эквиваленты становятся здесь уже лишними, поскольку мы работаем со значениями, состоящими из двух слов. Эта задача реализована в программе 24.2.

ПРОГРАММА 24.2. Вывод двух или более значений с плавающей точкой

```
/* Вывод двух чисел с плавающей точкой */
```

```
.global main
.func main
main:
    SUB SP, SP, #16           @ выделение пространства в стеке
    LDR R1, addr_value1      @ получение адреса value1
    VLDR S14, [R1]
    VCVT.F64.F32 D0, S14
    LDR R1, addr_value2      @ получение адреса value2
    VLDR S15, [R1]
    VCVT.F64.F32 D1, S15

    LDR R0, =string          @ в R0 помещается указатель на string
    VMOV R2, R3, D0          @ первое значение
    VSTR D1, [SP]            @ второе значение в стеке
    BL printf
    ADD SP, SP, #16          @ восстановление стека

    MOV R7, #1               @ выход через системный вызов
    SWI 0
```

```

addr_value1:      .word value1
addr_value2:      .word value2

        .data
value1:           .float 1.54321
value2:           .float 5.1
string:           .asciz "The FP values are %f and %f\n"

```

Процесс загрузки и преобразования остается прежним, но, очевидно, мы будем использовать другие регистры, а для помещения значения в стек служит команда `VPUSH`. Аналогичным образом можно добавить дополнительные значения.

Команды `VPUSH` и `VPOP` можно использовать с фигурными скобками, чтобы передать несколько элементов в стек и из стека:

```

VPUSH {S1-S4} @ помещение S1, S2, S3, S4 в стек
VPOP {S5-S8} @ вставка значений в S5, S6, S7 и S8

```

Обратите внимание, что большинство чисел с плавающей точкой, которые может представить компьютер, являются не точными, а приближенными. Одна из проблем в программировании при работе со значениями с плавающей точкой заключается в том, чтобы убедиться, что «приближенность» значений не мешает результатам. Если программист не будет осторожен, небольшие расхождения в приближениях могут, как снежный ком, превратиться в огромные ошибки.

Сборка и отладка на VFP с помощью GDB

Методология GCC идеально подходит для тестирования и отладки кода на низком уровне. Одним из преимуществ использования метода компиляции GCC является то, что GCC всегда выполняет всю «подкапотную» работу за вас. Анализируя код на ассемблере, компилятор понимает, что в программе используются коды операций с плавающей точкой, и сам подключает и собирает необходимую дополнительную информацию. Он же будет обрабатывать все необходимое для таких функций, как `printf`. Чтобы собрать *программу 24.1*, введите команду:

```
gcc -g -o prog24a prog24a.s
```

После этого вы можете использовать инструменты отладки GDB для изучения и пошагового выполнения кода. В GDB есть инструменты отладки для работы с VFP (об инструменте Neon мы поговорим в *главе 26*). Все параметры, описанные в *главе 14*, все еще подходят нам, и с ними GDB будет правильно дизассемблировать файлы машинного кода. Вы также можете обращаться к регистрам VFP, добавив расширение `all` к команде `info` следующим образом:

```
info r all
```

или так:

```
i r a
```

Вы увидите все регистры `D` и `S` (а также регистры `Q`, используемые Neon, но об этом позже). Вы заметите, что здесь также указаны регистры `D` вплоть до `D31`, хотя ранее мы отмечали, что есть только регистры до `D15`. Как мы увидим, набор регистров `D` расширен для соответствия регистрам `Q` в Neon.

Функция `all` позволяет получить очень много данных. Вы можете ограничить количество выводимых данных, указав в команде отдельные регистры. Сначала добавим точку останова, используя метку `bp:`, присутствующую в листинге программы. Для этого введите:

```
b bp
```

в приглашении GDB. Вы получите сообщение с подтверждением ввода. Не выходя из GDB, введите:

```
run
```

Код будет выполняться до точки останова, которая находится непосредственно перед вызовом функции `printf`. Весь остальной код до нее был выполнен. Теперь мы можем посмотреть содержимое соответствующих регистров, набрав:

```
info r s14 d5 r0 r1 r2 r3
```

Результат приведен в листинге 24.1. Для ясности я немного переформатировал вывод под регистр `D5`, чтобы его было легче читать.

Листинг 24.1. Содержимое регистров

```
s14 108.65625          (raw 0x42d95000)
d5   {u8 = {0x0, 0x0, 0x0, 0x0, 0x0, 0x2a, 0x5b, 0x40},
      u16 = {0x0, 0x0, 0x2a00, 0x405b},
      u32 = {0x0, 0x405b2a00},
      u64 = 0x405b2a0000000000,
      f32 = {0x0, 0x3}, f64 = 0x6c
r0   0x2102c           135212
r1   0x21028           135208
r2   0x0               0
r3   0x405b2a00        1079716352
```

Напомним, что шесть регистров в приведенном выводе указываются непосредственно перед вызовом `printf`, и поэтому в них передается вся информация.

- ◆ `S4` — хранит число одинарной точности 108,65625. Кроме того, показано его исходное двоичное значение: `0x42D95000`.
- ◆ `D5` — обычно нас здесь интересует представление числа в 32 разрядах. Вид числа следующий: `0x405b2a00` в первых 32 битах и 0 в остальных 32 битах. В 64-битном формате это выглядит так: `0x405b2a0000000000`.
- ◆ `R0` — содержит адрес строки, которая будет использоваться функцией `printf`;
- ◆ `R1` — хранит адрес с одинарной точностью, который следует вывести;

◆ R2, R3 — хранят значение, которое будет печатать функция `printf`. Аналогично регистру D5, рассмотренному ранее.

Следовательно, в регистрах R2 и R3 будет лежать значение двойной точности. Все дело в том, что функция `printf` печатает только числа с двойной точностью.

Значение в D5 — это число 108,656250, представленное в виде двоичного числа с одинарной точностью.

Отредактируйте программу 24.1 и измените число, которое нужно преобразовать, на следующее:

```
108.6000
```

Повторно соберите файл и снова запустите программу (не заходя в GDB). Мы получим результат:

```
Floating point value is: 108.599998
```

После обратного преобразования значение не окажется равным 108,60000. Все дело в ошибках округления. Несоответствие невелико, по факту всего 0,00000152587890625. Однако, если его усугубить, особенно путем умножения, оно может в конечном итоге стать весьма значительным.

Причина ошибки кроется в методике преобразования десятичного значения в двоичное. Иногда такие ошибки случаются, причем регулярно. В первом примере этого не произошло, поскольку начальное значение преобразуется правильно.

С помощью GDB вы можете узнать, что делают команды и как различные комбинации ваших действий влияют на флаги и результаты.

Вы можете собрать и связать ваш код для работы с числами с плавающей точкой в два этапа, используя `as` и `ld`, при условии, что будете соблюдать правила, изложенные в начальных главах этой книги. Вам также придется подключить библиотеки времени выполнения, которые потребуются для выполнения каких-либо вызываемых внешних функций, таких как `printf`, `scanf` и т. д. В этих случаях мы будем использовать компилятор GCC, который сделает все за нас.

Можно использовать отладчик, не забывая при этом добавить в последовательность сборки параметр `-g`.

Загрузка, хранение и перемещение

Как и в стандартном наборе команд ARM, в наборе VFP есть универсальные команды для перемещения информации. Команды `VLDR` и `VSTR` позволяют загружать и хранить величины в едином регистре с помощью косвенной адресации. Вот несколько примеров:

```
VLDR S1, [R5]      @ загрузка в S1 значения F32, хранящегося в R5
VLDR D2, [R5, #4] @ загрузка в S2 значения F64 addr + 4, хранящегося в R5
VSTR S3, [R6]      @ сохранение значения F32 из S3 по адресу в R6
```

Во втором примере используется предварительно индексированная адресация, в результате чего мы прибавляем 4 к адресу, хранящемуся в R5, до выполнения операции.

Мы также можем задействовать предварительно индексированную адресацию с обратной записью, чтобы обновить адрес в индексном регистре. Это полезно при использовании операций, работающих с наборами регистров, например:

```
VLDMIAS R5!, {S1-S4} @ копирование S1, S2, S3, S4 и обновление R5
```

В этом примере значения из четырех регистров S1, S2, S3 и S4 последовательно копируются в позицию слова, начиная с адреса, хранящегося в R5. По завершении операции длина занимаемого адресного пространства прибавляется к R5. Это означает, что R5 теперь будет указывать на следующий адрес — адрес после S4. Если бы команда выглядела так:

```
VLDMIAS R5!, {D1-D4} @ копирование D1, D2, D3, D4 и обновление R5
```

тогда была бы выполнена та же операция, но команда разрешила бы два слова (8 байтов) на регистр и добавила бы 32 байта к значению в R5. Если вы не хотите обновлять значение в R5, уберите из команды восклицательный знак.

Используемые режимы адресации аналогичны описанным в *главе 15*, но обратите внимание: регистры и информация организованы по-другому. Постиндексированная адресация не предусмотрена.

Команда `VMOV` позволяет свободно передавать значения между различными наборами регистров. При передаче значений между регистром VFP и регистром ARM передача выполняется побитно, а преобразование не выполняется:

```
VMOV S1, S2           @ копирование S2 в S1
VMOV S1, S2, R3, R5 @ копирование R3 в S1 и R5 в S2
VMOV R2, R4, D1      @ копирование младшей части D1 в R2,
                    @ а старшей - в R4
```

В последнем примере число двойной точности (длиной в 8 байтов) передается в два регистра. В таких случаях важно знать порядок старшей и младшей частей числа с плавающей точкой, поскольку иначе значение может получиться совершенно другим. Опять же, преобразование не выполняется, но выполняется побитовая передача. Значение сохраняется, если передача отменяется, поскольку измениться ничего не успеет.

Вы можете использовать инструкцию `VMOV` для копирования информации из регистров ARM в регистры VFP, и соответственно можно использовать команду, позволяющую сохранять содержимое регистра ARM в файле регистров, если вам нужно дополнительное пространство:

```
VMOV S1, R1           @ сохранение значения из R1 в S1
```


Векторная арифметика

В наборе команд VFP есть все нужные команды для выполнения всех арифметических операций, которые могут вам понадобиться. Их формат соответствует стандартной форме, уже знакомой вам, а суффиксы F32 и F64 указывают на значения с одинарной и двойной точностью. При выполнении операций должен применяться один формат точности в каждой строке, поскольку значения одинарной и двойной точности нельзя смешивать. В листинге 24.2 приведен пример всех команд с учетом возможных вариантов .F32 и .F64.

Листинг 24.2. Примеры команд векторной арифметики

VADD.F32 S0, S1, S2	@ сложение $S0 = S1 + S2$
VSUB.F64 D0, D2, D4	@ вычитание $D0 = D2 - D4$
VDIV.F64 D4, D5, D1	@ деление $D4 = D5/D1$
VMUL.F32 S2, S4, S1	@ умножение $S2 = S4 * S1$
VNMUL.F64 D4, D3, D2	@ умножение с отрицанием @ $D4 = -(D3 * D2)$
VMAL.F64 D4, D3, D2	@ умножение с накоплением @ $D4 = D4 + (D3 * D2)$
VSUB.F64 D0, D1, D2	@ умножение и вычитание @ $D0 = D0 - (D1 * D2)$
VABS.F32 S0, S1	@ модуль $S0 = ABS(S1)$
VNEG.F32 S2, S3	@ отрицание $S2 = -S3$
VSQRT.F64 D0, D1	@ квадратный корень $D0 = SQR(D1)$

25. Регистр управления VFP

У сопроцессора VFP есть три системных регистра. Наиболее важным из них для нас является *регистр состояния и управления чисел с плавающей точкой* (Floating-Point Status and Control Register, FPSCR). Он аналогичен регистру CPSR для обычного набора команд ARM, и в нем так же содержится информация о состоянии флагов. Знакомые нам флаги N, Z, C и V в нем присутствуют и работают так же. На рис. 25.1 показано, как регистр устроен, с точки зрения программиста, а в табл. 25.1 подробно описано назначение битов регистра, которые мы далее обсудим.

Формат FPSCR				Rd				Вектор			Исключение				CEB	
31	28	24	23	22	21	20	18	17	16	12	8	7	0			
N	Z	C	V				Rmode	Stride (Шаг)			Len (Длина)					

Рис. 25.1. Структура регистров состояния и управления с плавающей точкой

Принцип работы и назначение некоторых наборов флагов мы здесь рассмотрим. Но о принципах работы исключений, которым посвящена *глава 29*, сейчас подробно говорить не станем.

Таблица 25.1. Назначение битов регистра

Биты	Флаги	Назначение
31–28	Флаги состояния	Отрицательный, нулевой, перенос и переполнение
23–22	Режим округления	Управляет округлением значений
21–20	Шаг	Управляет размером шага в векторных банках
18–16	Длина	Задаёт длину вектора
12–8	Состояние исключения	Включает определенные типы исключений
7–0	Кумулятивное исключение	Захватывают кумулятивные исключения

Условное исполнение

Коды условий мы рассмотрели в *главе 10*. Точные значения флагов кода условия различаются в зависимости от того, были ли эти флаги установлены операцией с плавающей запятой или инструкцией обработки данных ARM. Все дело в том, что числа с плавающей точкой никогда не бывают беззнаковыми, и поэтому беззнаковые условия оказываются не нужны (есть и еще одна причина, связанная со значениями типа NaN, но, поскольку мы о них не говорили, это пока значения не имеет).

Единственная команда VFP, способная обновлять флаги состояния, — это команда VCMR, которая устанавливает относительные биты в FPSCR. А вот флаги условий и команды контролируются регистром APSR (Application Program Status Register, регистр состояния прикладной программы), поэтому для работы нужно скопировать флаги FPSCR в APSR. Для этого имеется специальная команда:

```
VMRS APSR_nzcv, FPSCR
```

У команды VCMR есть варианты .F32 и .F64, которые используются следующим образом:

```

VCMR.F32 S0, S1      @ выполнение вычитания S0 - S1
                     @ и установка флагов условий
VCMR.F64 D2, D3      @ выполнение вычитания D2 - D3
                     @ и установка флагов условий

```

Все содержимое регистра FPSCR можно скопировать в регистр ARM следующим образом:

```
VMRS R4, FPSCR      @ копирование FPSCR в R4
```

Аналогично, в FPSCR можно скопировать содержимое регистра ARM, что позволяет заранее определять и устанавливать биты:

```
VMSR FPSCR, R4      @ копирование R4 в FPSCR
```

Используя побитовые операторы (AND, ORR, EOR) в сочетании с этой командой, мы можем маскировать отдельные биты и проверять отдельно взятые флаги состояния. Особенно удачно этот механизм используется при работе с битами, отвечающими за параметры «длина» и «шаг», которые мы вскоре рассмотрим. В табл. 25.2 приведены расшифровки мнемоник кодов состояния ARM и VFP, которые для удобства расположены в соседних столбцах.

Таблица 25.2. Сравнение кодов условий ARM и VFP

Суффикс	После команды ARM	После команды VCMR
EQ	Равенство	Равенство
NE	Неравенство	Неравенство или неупорядоченность
CS	Установлен флаг переноса	Равно, больше или неупорядочено
HS	Больше или равно (без знака)	Равно, больше или неупорядочено

Таблица 25.2 (окончание)

Суффикс	После команды ARM	После команды VCMR
CC	Флаг переноса снят	Меньше
LO	Меньше (без знака)	Меньше
MI	Отрицательный результат	Меньше
PL	Положительный или ноль	Равно, больше или неупорядочено
VS	Переполнение	Неупорядоченность
VC	Нет переполнения	Не неупорядоченный
HI	Больше (без знака)	Больше или неупорядочено
LS	Меньше или равно (без знака)	Меньше или неупорядочено
GE	Больше или равно (со знаком)	Больше или равно
LT	Меньше (со знаком)	Меньше или неупорядочено
GT	Больше (со знаком)	Больше
LE	Меньше или равно (со знаком)	Меньше, равно или неупорядочено
AL	Всегда	Всегда

Одним из огромных преимуществ использования условного выполнения является сокращение числа применяемых команд ветвления и соответственно уменьшение общего объема кода. Кроме того, команды ветвления весьма затратны с точки зрения времени выполнения, т. к. заполнение конвейера процессора занимает три цикла. Например:

```

VADDEQ.F32 S0, S1, S2 @ Выполнить только при C = 1
VSUBNE.F64 D0, D2, D4 @ Выполнить, только при отрицательном результате

```

Из программы 25.1 видно, насколько легко использовать эти команды. Приведенный в ней код загружает значения в S14 и S15, а затем сравнивает их с помощью команды VCMR. В результате выполнения устанавливаются флаги в FPSCR. После чего мы копируем флаги NZCV в регистр состояния ARM с помощью команды VMRS. Затем, в зависимости от состояния флага C, в регистр R0 загружается либо 0, либо 255. После запуска программы выполните команду

```
echo $?
```

для вывода результата.

Вы можете поэкспериментировать со значениями констант, загружаемых в регистры одинарной точности, а с помощью GDB последить за значениями в регистрах. Вы можете попробовать дополнить эту программу и создать цикл, который будет вести обратный отсчет с шагом 0,1 и выводить значения на экран, пока дело не дойдет до нуля.

ПРОГРАММА 25.1. Условное выполнение в VFP

```

/* Условное выполнение в коде VFP */

.global main
.func main

main:
    LDR R1, addr_value1      @ получение адреса value1
    VLDR S14, [R1]
    VCVT.F64.F32 D1, S14

    LDR R1, addr_value2      @ получение адреса value2
    VLDR S15, [R1]
    VCVT.F64.F32 D2, S15

    VCMPEQ F32 S14, S15      @ сравнение S14 и S15
    VMRS APSR_nzcv, FPSCR    @ копирование флагов

    MOVEQ R0, #0              @ если C = 1, R0 = 0
    MOVNE R0, #255           @ если C = 0, R0 = 255

    MOV PC, LR

addr_value1:      .word value1
addr_value2:      .word value2

.data
value1:           .float 1.54321
value2:           .float 5.1

```

Скалярные и векторные операции

В предыдущей главе, когда мы говорили о файле регистров, я упоминал, что регистры по признаку доступа делятся на скалярные и векторные. На рис. 25.2 показано, как устроена эта архитектура.

Банк 0				Банк 1				Банк 2				Банк 3			
S0			S7	S8			S15	S16			S23	S24			S31
D0	D1	D2	D3	D4	D5	D6	D7	D8		D11		D12			D15

Рис. 25.2. Четыре банка VFP и связанные с ними регистры

В предыдущих примерах всегда подразумевалось, что при выполнении операций мы работаем с отдельными регистрами. Но VFP позволяет группировать регистры в векторы или наборы регистров. При выполнении векторных операций файл регистров VFP можно рассматривать как набор банков поменьше. Каждый из этих банков — это либо банк из восьми регистров одинарной точности, либо банк из четырех регистров двойной точности. Количество регистров, используемых вектором, определяется битом LEN в FPSCR. Банки регистров можно настраивать следующим образом:

- ◆ четыре банка регистров одинарной точности: от S0 до S7, от S8 до S15, от S16 до S23 и от S24 до S31;
- ◆ четыре банка регистров двойной точности: от D0 до D3, от D4 до D7, от D8 до D11 и от D12 до D15;
- ◆ любая комбинация банков одинарной и двойной точности.

Обычно значение флага LEN в VFP равно 1, поэтому команда будет работать только с регистрами, указанными в теле команды. Однако, увеличивая значение LEN, мы можем заставить команду работать и с другими регистрами в соответствующем банке регистров. Таким образом, вектор может начинаться с любого регистра и возвращаться в начало банка. Другими словами, если вектор выходит за пределы своего банка, он возвращается к началу того же банка (табл. 25.3).

Таблица 25.3. Бит LEN и его влияние на выбор банков

Бит LEN	Начальный регистр	Используемые регистры
2	D11	D11, D8
3	D7	D7, D4, D5
4	S5	S5, S6, S7, S0
5	S22	S22, S23, S16, S17, S18

Важно отметить, что в вектор не могут входить регистры из разных банков, поэтому, если вектор возвращается к началу, операция остановится сразу же при заполнении банка.

Итак, согласно табл. 25.3, в первой записи LEN равен 2. Это означает, что количество регистров, с которыми будет работать команда, равно двум. Стартовый регистр — D11. На рис. 25.2 мы видим, что D11 находится в последнем регистре в банке 2. После возврата к началу следующим регистром в банке станет D8 (D12 находится в банке 3).

Первый регистр, используемый вектором операнда, — это регистр, указанный как операнд в отдельных командах VFP. Первый регистр, используемый вектором назначения, — это регистр, который указывается как адресат в отдельных командах VFP.

В табл. 25.3 регистры, к которым осуществлялся доступ, были расположены по их номерам, т. е. по порядку, допускающему циклический переход. Но это могут быть

и не стоящие по порядку регистры, а это определяется установкой битов STRIDE в регистре FPSCR. В примерах, приведенных в табл. 25.3, параметр STRIDE был бы равен 1, поскольку используются следующие друг за другом регистры. А вот если STRIDE равен 2, регистры использовались бы через один (табл. 25.4).

Таблица 25.4. Влияние битов LEN и STRIDE на перенос вектора

Бит LEN	Бит STRIDE	Начальный регистр	Используемые регистры
2	2	D1	D1, D3
3	2	S1	S1, S3, S5
4	2	S6	S6, S0, S2, S4
5	1	S22	S22, S23, S16, S17, S18

Как мы уже отмечали, один и тот же регистр не может входить в вектор дважды, поэтому возможные сочетания значений LEN и STRIDE ограничены.

Обратите внимание на следующую команду:

```
VADD.F32 S8, S16, S24 @ S8 = S16 + S24
```

По умолчанию LEN = 1 и STRIDE = 1, поэтому содержимое S16 и S24 складывается, а результат помещается в S8. Однако, если мы установим LEN = 2 и STRIDE = 2 и выполним ту же самую команду, результат будет аналогичен выполнению следующих двух команд:

```
VADD.F32 S8, S16, S24 @ S8=S16+S24
VADD.F32 S10, S18, S26 @ S10=S18+S26
```

Или если мы зададим LEN = 4 и STRIDE = 2, то выполнение той же команды будет эквивалентно четырем командам:

```
VADD.F32 S8, S16, S24 @ S8=S16+S24
VADD.F32 S10, S18, S26 @ S10=S18+S26
VADD.F32 S12, S20, S28 @ S12=S20+S28
VADD.F32 S14, S22, S30 @ S14=S22+S30
```

Как видите, это мощный метод программирования, особенно полезный, когда дело доходит до матричных операций с наборами чисел.

Какой тип оператора?

По сути, арифметические операции VFP можно выполнять над скалярными числами, векторами или и теми и другими вместе. При значении LEN = 1 (по умолчанию) все операции VFP становятся скалярными. Если у LEN установлено любое другое значение, операнды могут быть скалярными, векторными или смешанными. Управлять тем, какие конкретно операторы будут использоваться, можно только путем выбора используемых банков регистров или регистров источника и назначения.

В большинстве задач Банк 0 (S0–S7 и D0–D3) является скалярным банком, а остальные три банка являются векторными. Если регистр назначения находится в одном из векторных банков, выполняется смешанная операция (между скаляром и вектором). В табл. 25.5 приведено несколько примеров с кратким описанием типа выполняемого действия. В этих примерах используется операция `VADD`, но примеры актуальны и для остальных арифметических команд VFP.

Таблица 25.5. Примеры скалярных, векторных и смешанных операций

Бит STRIDE	Бит LEN	Команда	Результат
1	1	VADD.F64 D0, D1, D2	Операция является скалярной, т. к. регистр назначения находится в Банке 0
1	1	VADD.F32 S4, S8, S20	Операция является скалярной, т. к. регистр назначения находится в Банке 0
2	4	VADD.F32 S10, S16, S24	Операция является векторной. В последней операции выполняется возврат в начало банка
2	2	VADD.F64 D4, D8, D0	Операция является смешанной, т. к. второй регистр-источник находится в Банке 0

Параметры *LEN* и *STRIDE*

Биты, определяющие параметры `LEN` и `STRIDE` в `FPSCR`, можно устанавливать с помощью команд `VMRS` и `VMSR`. При этом требуемая битовая комбинация передается в `FPSCR`. Это действия выполняются через регистр ARM в два этапа, т. к. сначала необходимо скопировать `FPSCR`, чтобы можно было сохранить его флаги, а затем с помощью маски изменить конкретно нужные биты `LEN` и `STRIDE`. Можно использовать любой регистр ARM, и порядок команд будет выглядеть так:

```
VMRS R4, FPSCR           @ копирование FPSCR в R4
                          @ установка нужных битов
VMSR FPSCR, R4           @ копирование R4 в FPSCR
```

Поле `LEN` занимает три бита (b16–b18), а поле `STRIDE` — два бита (b20 и b21). В табл. 25.6 показаны различные битовые комбинации значений `LEN` и `STRIDE` и результаты их работы, а также надежность результатов при работе с числами с одинарной и двойной точностью.

Таблица 25.6. Комбинации параметров *STRIDE* и *LEN* и их влияние на числа с одинарной и двойной точностью

№	STR	Биты	LEN	Биты	Одинарная точность (Sx)	Двойная точность (Dx)
1		b00	1	b000	Скалярный результат	Скалярный результат
2		b11	1	b000	Надежный результат	Надежный результат
3	1	b00	2	b001	Надежный результат	Надежный результат

Таблица 25.6 (окончание)

№	STR	Биты	LEN	Биты	Одинарная точность (Sx)	Двойная точность (Dx)
4	2	b11	2	b001	Надежный результат	Надежный результат
5	1	b00	3	b010	Надежный результат	Результат непредсказуем
6	2	b11	3	b010	Надежный результат	Надежный результат
7	1	b00	4	b011	Надежный результат	Результат непредсказуем
8	2	b11	4	b011	Надежный результат	Результат непредсказуем
9	1	b00	5	b100	Надежный результат	Результат непредсказуем
10	2	b11	5	b100	Результат непредсказуем	Результат непредсказуем
11	1	b00	6	b101	Надежный результат	Результат непредсказуем
12	2	b11	6	b101	Результат непредсказуем	Результат непредсказуем
13	1	b00	7	b110	Надежный результат	Результат непредсказуем
14	2	b11	7	b110	Результат непредсказуем	Результат непредсказуем
15	1	b00	8	b111	Надежный результат	Результат непредсказуем
16	2	b11	8	b111	Результат непредсказуем	Результат непредсказуем

Не все комбинации дают предсказуемые результаты, и таких комбинаций следует избегать. С помощью данных табл. 25.6 вы можете выбрать нужную комбинацию для типа значений, с которыми работаете. В частности, вы увидите, что биты STRIDE всегда равны 00 или 11, что соответствует значениям 1 или 2. У параметра LEN значение 1 представлено как 000 (т. е. фактически выставляемое битами двоичное значение на единицу меньше реального).

В программе 25.2 показано, как векторная адресация позволяет получить третий вариант, приведенный в табл. 25.5. Немалая часть кода посвящена заполнению значений, а затем их выводу с помощью команды printf. Как организовать вывод, вы уже знаете, и поскольку в конечном итоге мы будем выводить на экран четыре значения с двойной точностью, три из них нужно будет поместить в стек, поэтому в функции main: первым делом для этой цели резервируется 24 байта, т. е. три слова.

Фактически выполняется следующая команда:

```
VADD.F32 S10, S16, S24
```

Здесь задействованы три банка векторов: Банк 1, Банк 2 и Банк 3, поскольку мы в настройках векторного управления задали параметры STRIDE = 2, LEN = 4 (см. табл. 25.5). Банк 1 используется для хранения результатов (S10, S12, S14, S8), Банк 2 будет содержать первый набор значений (S16, S18, S20, S22), а Банк 3 — второй набор значений (S24, S26, S28, S30).

Мы определили пять используемых значений. Чтобы упростить проверку, регистры Банка 2 присваиваются значения одинарной точности, а затем четыре отдельных значения назначаются каждому из регистров в Банке 3.

Метка `lenstride`: показывает, где регистр `FPSCR` заполняется настройками значений `STRIDE` и `LEN`. Нам требуются значения 2 и 4 соответственно. В строке 8 табл. 25.6 показаны настройки битов для достижения одинарной точности: 11 и 011 («нормальная» работа для одинарной точности).

Поскольку параметры `STRIDE` и `LEN` разделены одним битом, нам нужно задать начальное значение 110011. Его нужно будет сдвинуть, чтобы крайний левый бит в `FPSCR` начинался с `b21`. Это делает команда `LSL #16`.

В коде программы 25.2 процедура `convert`: преобразует регистры `Sx` Банка 1 в значения двойной точности Банка 0. В программе Банк 0 не использовался, поэтому эти регистры свободны, но убедитесь, что у вас есть хотя бы один доступный для использования регистр двойной точности, если планируете использовать функцию `printf`, иначе вам придется много перемещать и восстанавливать значения регистров.

ПРОГРАММА 25.2. Использование битов `LEN` и `STRIDE` для сложения векторов

```

/**/ Использование битов LEN и STRIDE для сложения векторов ***/
.global main
.func main

main:
SUB SP, SP, #24           @ место для printf
LDR R1, addr_value1      @ получение адресов значений
LDR R2, addr_value2
LDR R3, addr_value3
LDR R4, addr_value4
LDR R5, addr_value5

VLDR S16, [R1]           @ загрузка значений
VLDR S18, [R1]           @ в регистры
VLDR S20, [R1]
VLDR S22, [R1]
VLDR S24, [R2]
VLDR S26, [R3]
VLDR S28, [R4]
VLDR S30, [R5]

lenstride:
/* Установка LEN=4 0b101 и STRIDE=2 0b11 */
VMSR R3, FPSCR           @ получение FPSCR
MOV R4, #0b110011        @ битовая маска
MOV R4, R4, LSL #16      @ перемещение к b21
ORR R3, R3, R4           @ везде должны быть единицы
VMSR FPSCR, R3           @ передача в FPSCR

```

```
VADD.F32 S10, S16, S24 @ векторное сложение
VADD.F32 S12, S18, S26
VADD.F32 S14, S20, S28
VADD.F32 S8, S22, S30
```

```
convert:
```

```
/* Выполнение преобразования для печати, убедившись, что */
/* не будут перезаписаны регистры Sx */
VCVT.F64.F32 D0, S10
VCVT.F64.F32 D1, S12
VCVT.F64.F32 D2, S14
VCVT.F64.F32 D3, S8
LDR R0, =string @ настройка для printf
VMOV R2, R3, D0
VSTR D1, [SP] @ помещение данных в стек
VSTR D2, [SP, #8]
VSTR D3, [SP, #16]
BL printf
ADD SP, SP, #24 @ восстановление стека
```

```
_exit:
```

```
MOV R0, #0
MOV R7, #1
SWI 0
```

```
addr_value1: .word value1
addr_value2: .word value2
addr_value3: .word value3
addr_value4: .word value4
addr_value5: .word value5
```

```
.data
```

```
value1: .float 1.0
value2: .float 1.25
value3: .float 1.50
value4: .float 1.75
value5: .float 2.0
```

```
string:
```

```
.asciz " S10 is %f\n S12 is %f\n S14 is %f\n S8 is %f\n"
```

Эта программа до смешного проста, но зато отлично позволит вам разобраться с FPSCR и битами STRIDE и LEN.

26. Сопроцессор Neon

На момент подготовки книги эта глава не актуальна для Raspberry Pi серии 1 и для платы Raspberry Pi Zero. Это связано с тем, что процессор Neon появился лишь в серии Raspberry Pi 2, или, если говорить точнее, в определенной версии чипа ARM, который туда поставили. Neon тесно связан с архитектурой VFP, о которой мы говорили в *главе 24*, посвященной числам с плавающей точкой. Однако это не блок с плавающей точкой (FPU) процессора ARM (в табл. 1.1 мы уже показывали эволюцию микросхем, используемых в Raspberry Pi).

Neon — это усовершенствованный процессор SIMD (Single Instruction, Multiple Data, Одна инструкция, несколько данных), который может за одно действие обрабатывать много элементов данных, и при этом данные остаются в своих регистрах! В процессе работы даже кажется, что вы просто загружаете, а затем сохраняете данные. Таким образом, из Neon можно выжать больше «скорости» (или выполненных операций), чем из стандартного процессора SISD (Single Instruction, Single Data) с той же тактовой частотой. Важность Neon заключается в том, что он может быстро сортировать повторяющуюся сложную информацию с помощью так называемого чередования.

Рис. 26.1 иллюстрирует философию работы Neon. Когда обычный процессор выполняет команды $A0 + B0 = C0$, $A1 + B1 = C1$ (*слева*), система Neon делает это одной командой, а результат получается таким же (*справа*)

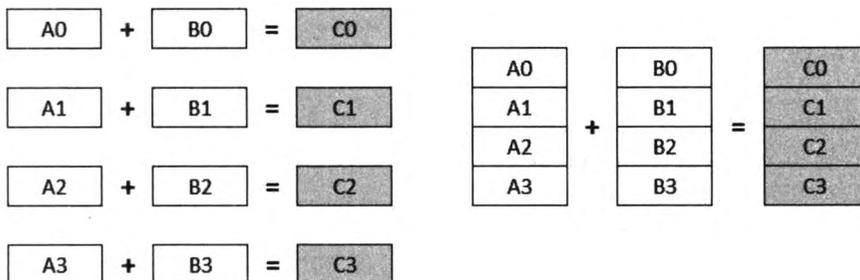


Рис. 26.1. Операции в Neon можно проводить параллельно

Neon поддерживает несколько типов данных:

- ◆ 32-битные числа с плавающей точкой одинарной точности;
- ◆ 8-, 16-, 32- и 64-битные целые числа со знаком и без знака;
- ◆ 8- и 16-битные полиномы.

Для различения типов перед размером принято ставить первую букву типа. Например, 32-битное целое число без знака обозначается как U32, 32-битное число с плавающей точкой — F32 и т. д.

Между системами Neon и VFP существует несколько различий:

- ◆ Neon не поддерживает работу с числами с плавающей точкой двойной точности;
- ◆ Neon работает только с векторами и не поддерживает сложные операции, такие как извлечение квадратного корня и деление;
- ◆ у VFP есть специализированные команды, не поддерживаемые устройством Neon (например, SQRT).

Процессор Neon используется для быстрого выполнения операций над большим количеством чисел. Если вам нужна точность при работе с числами с плавающей точкой, обращайтесь к VFP. Но следует помнить, что Neon использует одни и те же регистры с плавающей точкой, что и VFP, поэтому, если вы работаете с обоими форматами одновременно, нужно очень внимательно следить за управлением регистрами.

Как показано на рис. 26.2, система Neon использует банк из 32 64-битных регистров, который также можно перенастроить на 16 128-битных регистров:

- ◆ 32 64-битных (по два слова) регистра: D0–D31;
- ◆ 16 128-битных (по четыре слова) регистра: Q0–Q15.

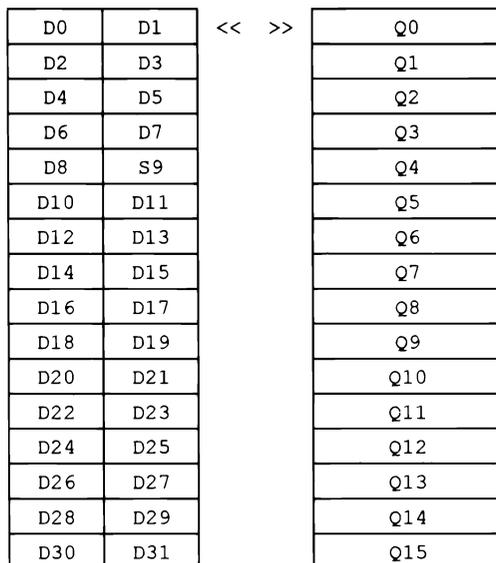


Рис. 26.2. Структура регистра Neon. Сравните эту схему с рис. 24.2, чтобы вспомнить, как выглядит структура регистра VFP

Уже известные нам регистры D из VFP удваиваются, составляя регистры Q (Quad, четверные). У этих регистров используются псевдонимы имен, поэтому данные в регистре Q совпадают с данными в двух соответствующих регистрах D . Например, $Q0$ — это псевдоним пары $D0$ и $D1$, и по обоим вариантам имен хранятся одни и те же данные.

Чтобы повысить производительность Neon и уменьшить плотность кода, в набор команд Neon добавлены структурированные команды загрузки и сохранения, которые позволяют загружать одно или несколько значений из одной или нескольких «дорожек» в векторном регистре или сохранять их туда. Эти операции загрузки и сохранения невероятно универсальны и позволяют управлять данными во время операций, извлекая данные из памяти и одновременно раскладывая значения в нужные регистры.

Ассемблер Neon

В программе 26.1 приведен простой пример, попробовав который вы убедитесь в том, что процессор Neon на вашей Raspberry Pi работает. Вы также проверите правильность выполнения операций сборки и связывания. Предполагая, что файл с кодом называется:

```
prog26a.s
```

вы можете собрать и связать код этой программы:

```
as -mfpu=neon-vfpv4 -g -o prog26a.o prog26a.s
ld -o prog26a prog26a.o
```

Опция

```
-mfpu=neon-vfpv4
```

в командной строке ассемблера указывает, что выполнение команд Neon разрешено. Если вы пропустите эту опцию, то почти наверняка получите несколько сообщений об ошибках. Добавьте опцию `-g`, если вы хотите просмотреть работу программы в GDB.

ПРОГРАММА 26.1. Простая проверка Neon

```
/* Проверка работоспособности Neon */
```

```
.global _start
_start:
    LDR R0, =number1
    LDR R1, =number2

    VLD1.32 {Q1}, [R0]
    VLD1.32 {Q2}, [R1]
    VADD.I32 Q0, Q1, Q2
```

```
MOV R7, #1
SWI 0
```

```
.data
```

```
number1:      .word 1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
number2:      .word 2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
```

В этом коде есть всего три строки, которые вы увидите впервые, и вы наверняка уже догадались, что делает по меньшей мере одна из них! Программа помещает значения 1 и 2 в Q1 и Q2 соответственно, а затем складывает их, сохраняя результат в Q0. Числа в регистры помещает команда VLD, а команда VADD складывает их, давая нужный результат.

Как уже отмечалось ранее, вы можете использовать GDB для подробного анализа Neon, если во время сборки добавите опцию `-g`. Доступ к регистрам Neon осуществляется по номеру. Например, точка останова после команды VADD в программе 26.1 позволяет вам увидеть результат операции с помощью команды

```
info r q0 q1 q2
```

Поскольку регистры Q можно использовать для хранения данных во множестве форматов и большая часть информации, которую вы передаете в командах Neon, определяет именно формат, вы можете изучить все возможные выходные данные с помощью GDB. Результат, показанный в листинге 26.1 (я отформатировал его для простоты восприятия), должен быть вам понятен.

Листинг 26.1. Результат выполнения операции: `info r q0 q1 q2`

```
q0 {u8 = {0x3, 0x0 <повторяется 15 раз>},
    u16 = {0x3, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0},
    u32 = {0x3, 0x0, 0x0, 0x0},
    u64 = {0x3, 0x0},
    f32 = {0x0, 0x0, 0x0, 0x0},
    f64 = {0x0, 0x0}}

q1
  {u8 = {0x1, 0x0 < повторяется 15 раз >},
   u16 = {0x1, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0},
   u32 = {0x1, 0x0, 0x0, 0x0},
   u64 = {0x1, 0x0},
   f32 = {0x0, 0x0, 0x0, 0x0},
   f64 = {0x0, 0x0}}

q2
  {u8 = {0x2, 0x0 < повторяется 15 раз >},
   u16 = {0x2, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0},
   u32 = {0x2, 0x0, 0x0, 0x0},
   u64 = {0x2, 0x0},
   f32 = {0x0, 0x0, 0x0, 0x0},
   f64 = {0x0, 0x0}}
```

Команды и типы данных Neon

В табл. 26.1 приведены типы данных, которые можно использовать в командах Neon. Имя типа данных состоит из размера значения в битах и буквенного обозначения формата. Как можно видеть, не все типы данных доступны во всех размерах.

В программе 26.1 в команде `VADD` указан тип `I32`. Из табл. 26.1 видно, что это 32-битное целочисленное (или неопределенного типа) сложение. Важно понимать, что регистры могут содержать один или несколько элементов одного и того же типа данных. Таким образом, мы можем собрать не общее значение в регистре, а отдельные значения элементов в векторе.

Таблица 26.1. Обозначения типов данных Neon

Тип	8 битов	16 битов	32 бита	64 бита
Беззнаковое целое	U8	U16	U32	U64
Целое со знаком	S8	S16	S32	S64
Целое неопределенного типа	I8	I16	I32	I64
Число с плавающей точкой	Не используется	F16	F32	Не используется
Многочлен по {0,1}	P8	P16	Не используется	Не используется

Количество элементов, участвующих в операции, указывается через размер регистра:

```
VADD.I16 Q0, Q1, Q2
```

Эта команда говорит, что мы работаем с 16-битными целыми элементами, хранящимися в 128-битных `Q`-регистрах. Вычисления производятся путем деления вектора на несколько дорожек, после чего операция выполняется параллельно на восьми 16-битных дорожках. Схематично этот процесс проиллюстрирован на рис. 26.3, где также наглядно видно, как организованы эти самые «дорожки». Команда выполняет параллельное сложение восьми дорожек из 16-битных элементов, которые берутся из векторов `Q1` и `Q2`, а результат сохраняется в `Q0`.

При работе с большими регистрами они разделяются на равные по размеру элементы заданного типа, после чего операция выполняется над соответствующими элементами каждого регистра.

Далее приведен пример, в котором беззнаковое 16-битное содержимое регистров `D0` и `D1` складывается в четырех 16-битных параллельных дорожках. При этом элементы `D0` и `D1` складываются, образуя четыре числа-результата, которые помещаются в `D2`:

```
VADD.U16 D2, D1, D0
```

Если предположить, что в `D0` и `D1` лежат значения, показанные на рис. 26.4 в верхних двух строках, то результатом сложения будут значения, показанные на рис. 26.4 в нижней строке.

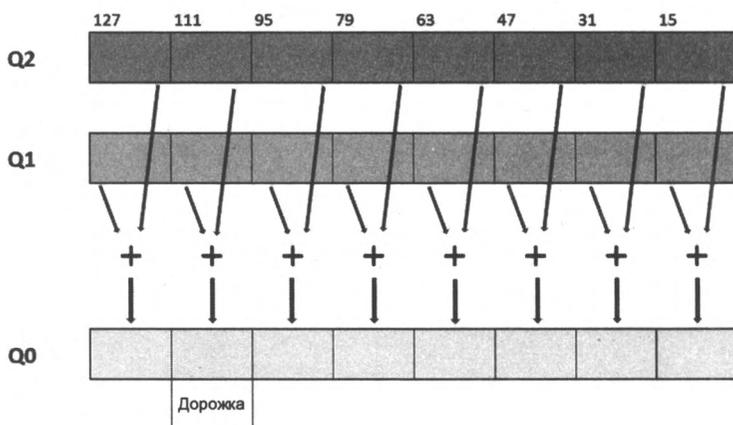


Рис. 26.3. Управление дорожками данных в Neon

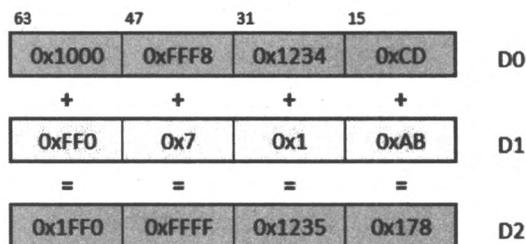


Рис. 26.4. Сложение дорожек в Neon

Не забываем, что регистры D0 и D1 вместе называются Q0, а D2 является «нижней половиной» Q1. Вы можете использовать эту информацию в своих интересах, если хотите манипулировать отдельными дорожками данных. Несколько примеров этого будет приведено в этой главе позже.

У некоторых команд используются регистры ввода и вывода разного размера. Например, этот код:

```
VMULL.S16 Q0, D2, D3
```

параллельно умножает четыре 16-битные дорожки, создавая четыре 32-битных произведения в 128-битном векторе назначения! В теле команды указано, что хранится в этих векторах.

У Neon нет флагов состояния для каждой дорожки в отдельности. Если требуется получить некоторый общий результат и есть вероятность того, что образуется большой перенос, то его необходимо обработать, используя более общий результат. Например так, как показано в примере, приведенном ранее. В противном случае регистры Neon будут использовать флаги в FPSCR процессора VFP и команды, работающие в зависимости от их значений.

Тип данных всегда соответствует регистру источника. Нельзя получить размер больше 64-битного или меньше 8-битного. Некоторые команды могут повышать или понижать разрядность в рамках своей нормальной работы:

VADDL.S32 Q0, D0, D1 @ 2 знаковых 32-битных числа превращаются в
 @ 64-битные, а затем складываются
 VADDW.S32 Q0, Q0, D2 @ повышение уровня D2 до S64 и выполнение
 @ 2x64-бит сложения с Q0

Режимы адресации

Возможности Neon в установке режима адресации ограничены. Например, следующая команда загружает в D0 содержимое адреса, хранящегося в R0:

```
VLD1.64 {D0}, [R0]
```

Этот режим адресации использовался в сочетании с регистром Q для загрузки вектора в *программе 26.1*.

Следующий код делает то же самое, но добавляет размер передачи к R0 после выполнения передачи, что удобно, когда вы сохраняете данные в последовательно расположенных блоках памяти.

```
VLD1.64 {D0}, [R0]!
```

Наконец, следующая команда складывает содержимое R1 и R0, но перед этим в D0 загружаются данные, лежащие по адресу, хранящемуся в R0:

```
VLD1.64 {D0}, [R0], R1
```

Параметр *Stride* команд *VLD* и *VST*

Команды загрузки и сохранения в Neon невероятно хороши. Синтаксис команды состоит из пяти частей:

- ◆ сама команда: *vld* — для загрузки или *vst* — для хранения;
- ◆ число, задающее промежуток между соответствующими элементами в каждой структуре (шаблон чередования);
- ◆ тип элемента, определяющий количество битов в элементах, к которым осуществляется доступ;
- ◆ набор 64-битных регистров Neon для чтения или записи (в зависимости от шаблона чередования можно указать до четырех регистров);
- ◆ регистр ARM, содержащий адрес в памяти, к которому необходимо получить доступ. Адрес можно изменить после доступа к нему в зависимости от используемого режима адресации.

В *программе 26.1* использовалась команда *vld*, которая загружала в Q1 и Q2 числа из в памяти, адрес которых лежит в R0:

```
VLD1.32 {Q1}, [R0]
```

Это простейшая команда загрузки. Шаблон чередования здесь — 1. В этом случае доступ к данным осуществляется как есть и элементы передаются прямо: один за

другим, последовательно и по порядку. Обычно в качестве значения шаблона чередования используются значения 1, 2, 3 или 4, т. е. берется от одного до четырех элементов одинакового размера, где элементы представляют собой обычные подерживаемые Neon числа шириной 8, 16 или 32 бита:

- ◆ VLD1 загружает от одного до четырех регистров данных из памяти без нарушения чередования. Используется при обработке массива данных без чередования;
- ◆ VLD2 загружает два или четыре регистра данных, чередуя четные и нечетные элементы в этих регистрах. Используется для разделения стереофонических аудиоданных на левый и правый каналы;
- ◆ VLD3 загружает три регистра и выполняет чередование. Полезно для разделения пикселей RGB на разные каналы;
- ◆ VLD4 загружает четыре регистра и выполняет чередование. Применяется для обработки данных изображений ARGB.

Эти команды широко используются в среде обработки аудиовизуальных материалов. Вы можете задать значение 2 — для разделения стереофонических аудиоданных на левый и правый каналы, 3 — для разделения пикселей RGB на отдельные каналы и 4 — для обработки изображений ARGB. И это лишь несколько примеров. Во всех этих примерах команда VST также может делать то же самое перед сохранением данных в памяти.

Рассмотрим следующий пример:

```
VLD 2.8 {D14, D15}, [R0]
```

Здесь:

- ◆ 2 — шаблон чередования (шаг). Может быть равен 1, 2, 3 или 4;
- ◆ 8 — тип данных: 8, 16 или 32 бита;
- ◆ D14, D15 — список используемых регистров Neon. Можно использовать до четырех регистров;
- ◆ [R0] — регистр ARM, содержащий адрес данных.

На рис. 26.5 показана часть работы команды:

```
VLD2.16 {D0, D1}, [R0]
```

Команда загружает в регистры D0 и D1 (Q0) четыре 16-битных элемента в первом регистре (D0) и четыре 16-битных элемента во втором (D1), при этом смежные пары (x и y) каждого регистра разделены.

Изменение размера на 32 бита позволит загрузить тот же объем данных, но теперь только два элемента составляют каждый вектор, снова разделенный на элементы x и y. Работа команды

```
VLD2.32 {D0, D1}, [R0]
```

показана на рис. 26.6.

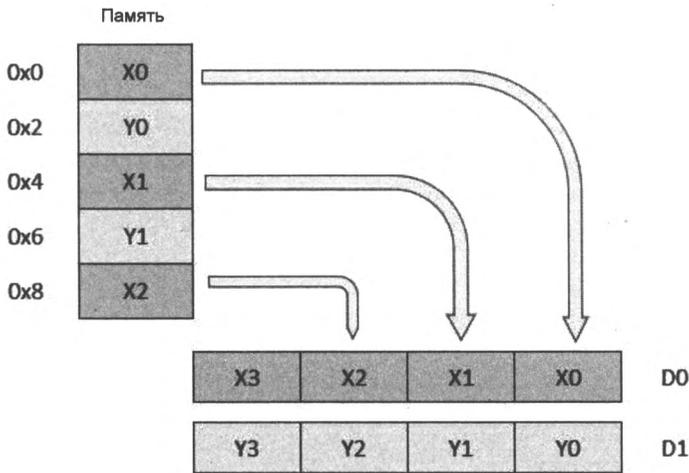


Рис. 26.5. Выборочная загрузка данных из памяти в регистры с помощью команды `VLD2.16 {D0, D1}, [R0]`

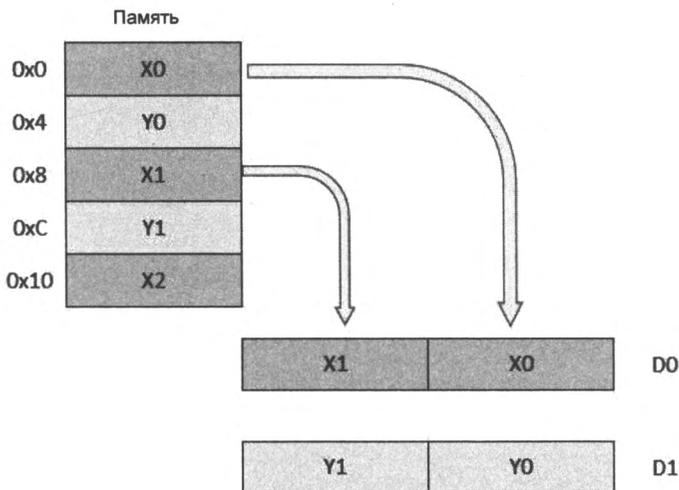


Рис. 26.6. Выборочная загрузка данных из памяти в регистры с помощью команды `VLD2.32 {D0, D1}, [R0]`

Размер элемента может повлиять на объект данных, и, как правило, если вы укажете в командах загрузки и сохранения правильный размер элемента, байты будут считываться из памяти в правильном порядке. Если нет, то в какой-то момент вам может потребоваться выполнить пару настроек вручную. Сохранение данных напрямую в памяти — всегда хорошо. Например, при загрузке 32-битных элементов нужно выровнять адрес первого по крайней мере на 32 бита.

Вернемся к *программе 26.1*. Она берет некоторые простые визуальные данные и выполняет на них команду `vld`. Если вы скомпилируете ее, добавив параметр `-g`, то можете перейти в GDB, просмотреть регистры, и вам станет понятно, что случи-

лось с вашими данными. Вы можете модифицировать программу, просто используя другие значения и дополнительные регистры, что даст вам более глубокое понимание происходящего.

Одним из наиболее распространенных применений этого вида команд Neon является чтение и сортировка данных в формате RGB. Если мы предположим, что информация была сохранена в виде последовательности R,G,B, R,G,B, R,G,B в 24-битном формате, вы можете отсортировать данные на каналы R, G и B, используя команду:

```
VLD3.8 {D0, D1, D2}, [R0]
```

При этом информация о красном цвете (R) окажется в D0, о зеленом (G) — в D1 и о синем (B) — в D2.

Некоторые команды могут ссылаться на отдельные скалярные элементы, для которых используется синтаксис массива $V_n[x]$ (порядок элементов в массиве всегда начинается с младшего бита).

Рассмотрим такую команду:

```
VLD4.8 {D0[2], D1[2], D2[D2], D3[2]}, [R0]
```

Она позволяет взять элемент в третьей дорожке из четырех векторов и сохранить их по адресу в R0, оставив остальные дорожки нетронутыми.

Загрузка в прочих форматах

Помимо структурной загрузки и сохранения у Neon также есть два других формата команд:

- ◆ VLDR и VSTR — для загрузки или сохранения одного регистра в виде 64-битного значения;
- ◆ VLDM и VSTM — для загрузки нескольких регистров в виде 64-битных значений.

Последний способ может быть полезен для загрузки регистров в стек и извлечения из него.

Neon Intrinsic

Из-за сложности приложений на основе Neon (например, в задачах декодирования видео или звука на Raspberry Pi) для написания исходного кода и компиляции в машинный код ARM используется язык C или его производные (например, C++). Но наиболее прямой способ задействовать Neon — это писать код на ассемблере, и не в последнюю очередь потому, что код, производимый из файла C, часто бывает расточителен по отношению к ресурсам. Кроме того, C не всегда наилучшим образом использует регистры при компиляции.

Термин «Neon Intrinsic» (внутренний Neon) часто применяется для обозначения компиляции в Neon из C. Внутренняя функция — это функция языка программирования, реализация которой специально обрабатывается компилятором. Обычно внутренний механизм заменяет последовательность автоматически сгенерирован-

ных команд исходного вызова функции на другую встроенную функцию. Однако, в отличие от встроенной функции, компилятор должен хорошо знать внутреннюю функцию, а следовательно, может интегрировать и оптимизировать ее для конкретной ситуации.

Внутренние функции часто используются для явной реализации векторизации и распараллеливания на языках, в синтаксисе которых такие конструкции не предусмотрены. Компилятор анализирует встроенные функции и преобразует их в векторный математический или многопроцессорный код, подходящий для целевой платформы.

Neon Intrinsics — это набор определений, которые побуждают использовать Neon при компиляции программы на C. Некоторые программисты их любят, другие — нет. Я не фанат программ, которые нужно оптимизировать по производительности. Компилятор постоянно добавляет между внутренними операциями дополнительные шаги выгрузки/загрузки регистров. Трудно заставить его упростить эту работу — проще просто написать код на чистом ассемблере Neon. На этом уровне хорошо знать, что происходит, и управлять происходящим самостоятельно. Особенно если важна скорость выполнения.

Массивы Neon

Поскольку Neon позволяет управлять большими объемами данных с помощью одной команды, он часто используется для написания ПО для работы с графикой. Например, если вы поворачиваете изображение на своем смартфоне или планшете, этот процесс, вероятно, выполняется путем управления блоками данных и команд Neon.

В следующем примере код поворачивает содержимое блока из четырех регистров Q на 90 градусов. На рис. 26.7 показана матрица до и после поворота. Цифры слева — это регистры Q, а числа в матрице приведены для того, чтобы было понятнее, как все выглядит до и после. Данные загружаются в регистры Q, и поскольку порядок массивов всегда начинается с младшего значащего бита, регистры D, связанные с ними, также будут иметь соответствующие значения.

	До поворота		После поворота	
Q0	0	1	2	3
Q1	4	5	6	7
Q2	8	9	10	11
Q3	12	13	14	15
	12	8	4	0
	13	9	5	1
	14	10	6	2
	15	11	7	3

Рис. 26.7. Поворот данных на 90 градусов

В программе 26.2 приведен код ассемблера, который выполняет этот поворот. Для наглядности мы используем простые числа, чтобы вы могли визуально оценить начальный и конечный результаты с помощью дампа регистров в GDB. Давайте проработаем каждый блок кода и рассмотрим все команды.

ПРОГРАММА 26.2. Поворот 2D-матрицы на 90 градусов (по часовой стрелке)

```

/* Поворот матрицы 4x4 на 90 градусов */

.global _start
_start:

    @ получение указателей данных
    LDR R0,=matrix0
    LDR R1,=matrix1
    LDR R2,=matrix2
    LDR R3,=matrix3

    @ загрузка данных в Q0 - Q3
    VLD1.32 {Q0}, [R0]
    VLD1.32 {Q1}, [R1]
    VLD1.32 {Q2}, [R2]
    VLD1.32 {Q3}, [R3]

    @ транспонирование матрицы, а затем чередование внутренних пар
bp1:
    VTRN.32 Q0, Q1
    VTRN.32 Q2, Q3
    VSWP D1, D4
    VSWP D3, D6

    @ зеркально отраженная матрица
    VREV64.32 Q0, Q0
    VREV64.32 Q1, Q1
    VREV64.32 Q2, Q2
    VREV64.32 Q3, Q3

    @ обмен местами старшей и младшей половин
    VSWP D0, D1
    VSWP D2, D3
    VSWP D4, D5
    VSWP D6, D7

    @ сохранение результата
bp2:
    VST1.32 {Q0}, [R0]
    VST1.32 {Q1}, [R1]

```

```
VST1.32 {Q2}, [R2]
VST1.32 {Q3}, [R3]

MOV R7, #1
SWI 0
```

```
.data
matrix0:    .word 0,1,2,3
matrix1:    .word 4,5,6,7
matrix2:    .word 8,9,10,11
matrix3:    .word 12,13,14,15
```

Некоторые приведенные в этом коде команды мы использовали впервые, и если вы хотите копнуть глубже, то можете поработать с ними подробнее, заменяя числа в матрице на цвета, например.

Данные для нашего массива хранятся в конце листинга программы в блоке `.data`. В матрице используются цифры от 0 до 15. После запуска кода регистры `R0`, `R1`, `R2` и `R3` указывают на свои соответствующие строки и загружаются как 32-битные значения в `Q0`, `Q1`, `Q2` и `Q3` соответственно. Если вы посмотрите на значения в регистрах до выполнения двух команд транспонирования, все будет видно, особенно, если посмотреть на вывод регистра `u32`. В регистрах `Q0`, `Q1`, `Q2` и `Q3` изначально будут находиться следующие значения (рис. 26.8).

Q0	0	1	2	3
Q1	4	5	6	7
Q2	8	9	A	B
Q3	C	D	E	F

Рис. 26.8. Исходные значения в регистрах `Q0`, `Q1`, `Q2` и `Q3`

Первая операция — это транспонирование самой матрицы. Команда `VTRN` (Vector Transpose, транспонирование вектора) обрабатывает элементы своих векторно-операндов как элементы матриц 2×2 и транспонирует их:

```
@ Транспонирование матрицы
VTRN.32 Q0, Q1
VTRN.32 Q2, Q3
```

Первая строка меняет местами `Q0` и `Q1`, поэтому 1 и 4 меняются местами, как и 3 и 6. Вторая строка работает с регистрами `Q2` и `Q3`, где меняются местами 9, B и E (рис. 26.9).

Затем выполняется команда `VSWP` (Vector Swap, векторный обмен) для обмена содержимого двух векторов. Векторы могут быть как двойными, так и четверными. Тип данных значения не имеет. Фактически эта команда реализует чередование:

Q0	0	4	2	6	0	4	2	6
Q1	1	5	3	7	1	5	3	7
Q2	8	9	A	B	8	C	A	E
Q3	C	D	E	F	9	D	B	F

Рис. 26.9. Значения в регистрах Q0, Q1, Q2 и Q3 после выполнения команд VTRN

VSWP D1, D4
VSWP D3, D6

D1 — это два «старших» элемента Q0, а D4 — два «младших» элемента Q2. На рис. 26.10 видно, что в левой матрице 8 и C поменялись местами с 2 и 6, а в правой — 9 и D поменялись местами с 3 и 7.

Q0	0	4	8	C	0	4	8	C
Q1	1	5	3	7	1	5	9	D
Q2	2	6	A	E	2	6	A	E
Q3	9	D	B	F	3	7	B	F

Рис. 26.10. Значения в регистрах Q0, Q1, Q2 и Q3 после выполнения команд VSWP

Команда VREV64 используется для изменения порядка 32-битных элементов в каждом двойном слове вектора. Это делается для каждого из четырех регистров Q:

@ Зеркально перевернутая матрица

VREV64.32 Q0, Q0

VREV64.32 Q1, Q1

VREV64.32 Q2, Q2

VREV64.32 Q3, Q3

Матрица теперь выглядит, как показано на рис. 26.11.

Q0	4	0	C	8
Q1	5	1	D	9
Q2	6	2	E	A
Q3	7	3	F	B

Рис. 26.11. Значения в регистрах Q0, Q1, Q2 и Q3 после выполнения команд VREV64

Теперь достаточно лишь подогнать некоторые элементы, поменяв местами нижнюю и верхнюю половины каждого полного регистра следующим образом:

@ Меняем местами старшую и младшую половины

VSWP D0, D1

VSWP D2, D3

VSWP D4, D5
VSWP D6, D7

Матрица теперь выглядит, как показано на рис. 26.12.

Q0	C	8	4	0
Q1	D	9	5	1
Q2	E	A	6	2
Q3	F	B	7	3

Рис. 26.12. Значения в регистрах Q0, Q1, Q2 и Q3 после выполнения подгонки командами VSWP

В результате поворот завершен, и теперь можно сохранять данные обратно в память.

Если вы скомпилируете *программу 26.2*, то сможете использовать метки bp1 и bp2 в качестве меток точек останова и увидеть содержимое регистров Q0, Q1, Q2 и Q3 (листинг 26.2) в каждой из этих точек (здесь показаны только данные U32).

Листинг 26.2. Содержимое регистров Q0, Q1, Q2 и Q3 в точках останова (данные U32)

```
Breakpoint 1, bp1 () at Prog26b.s:21
21      VTRN.32 Q0, Q1
(gdb) info r q0 q1 q2 q3
q0      u32 = {0x0, 0x1, 0x2, 0x3}
q1      u32 = {0x4, 0x5, 0x6, 0x7}
q2      u32 = {0x8, 0x9, 0xa, 0xb}
q3      u32 = {0xc, 0xd, 0xe, 0xf}

Breakpoint 2, bp2 () at Prog26b.s:42
42      VST1.32 {Q0}, [R0]
(gdb) info r q0 q1 q2 q3
q0      u32 = {0xc, 0x8, 0x4, 0x0}
q1      u32 = {0xd, 0x9, 0x5, 0x1}
q2      u32 = {0xe, 0xa, 0x6, 0x2}
q3      u32 = {0xf, 0xb, 0x7, 0x3}
```

Правильный порядок

При выполнении каких-либо матричных вычислений важно, чтобы используемые данные были правильными и единообразными. Это необходимо для того, чтобы вычисления выполнялись для тех же элементов, что и в обрабатываемой матрице. В какой-то момент вам нужно будет переместить информацию из источника куда-то, где можно будет с ней поработать. Вектор можно представить как одномерный массив. Память занимает линейное пространство, поэтому это также одномерная

область. Матрица — это двумерный объект. Матрица 4×4 содержит 16 ячеек, расположенных в виде массива, состоящего из четырех строк и четырех столбцов. Как преобразовать ее в вектор разумным и удобным для обработки способом?

Наиболее популярными методами хранения многомерных массивов в линейной памяти являются строковый порядок и столбцовый порядок (рис. 26.13).

- ◆ При хранении в столбцовом порядке столбцы располагаются один за другим. На рис. 26.13 показано, что столбец 0 с числами 1, 4, 7 располагается в векторе на первых трех местах. За ним следует столбец 1 (2, 5, 8), а затем столбец 2 (3, 6, 9).
- ◆ В строковом порядке строки хранятся последовательно. То есть сначала идет первая строка (1, 2, 3), затем вторая (4, 5, 6) и, наконец, третья (7, 8, 9).



Рис. 26.13. Варианты хранения данных в столбцовом (вверху) и строковом (внизу) порядке

Этот процесс одинаков для матрицы любого размера. Если вам известен размер массива и способ хранения данных, его можно деконструировать и восстановить безопасно и без ошибок. Этот же метод позволяет быстро и эффективно сортировать и перемещать данные. Посмотрите еще раз на рис. 26.13. Если вы взяли данные в столбцовом порядке, а затем деконструировали его в строковый, то вы выполнили поворот матрицы.

Вы сами можете решить, на каком принципе хранения данных остановиться. Выбор может быть обусловлен процедурой, которую вы хотите использовать, или языковой средой, в которой вы работаете (например, C или Python). Зная Neon, вы можете независимо управлять данными обоими способами (но лучше все же выбрать какой-то один).

Матричная математика

Матричная математика при наличии сопроцессора VFP и/или Neon выполняется достаточно просто. В следующих двух примерах показано, как складывать и умножать матрицы. Вы можете выполнять и другие операции. Используя правильные принципы работы и знания матричной математики вы легко получите требуемый результат.

Чтобы сложить две матрицы, нужно сложить соответствующие элементы в каждом столбце, содав тем самым третью матрицу, содержащую результат (рис. 26.14). Этот пример реализован в программе 26.3.

Матрица 1				Матрица 2					
	Столб. 0	Столб. 1	Столб. 2	Столб. 3					
Строка 0	106.6	105	104	103	+	2	3	4	5
Строка 0	2.4	4	6	8		3	4	5	6
Строка 0	4.5	6	8	10		4	5	6	7
Строка 0	6.2	8	10	12		5	6	7	8

Результат = M1 + M2			
108.6	108	108	108
5.4	8	11	14
8.5	11	14	17
11.2	14	17	20

Рис. 26.14. Сложение двух матриц

ПРОГРАММА 26.3. Сложение двух матриц размером 4x4

```

/* Сложение двух матриц 4x4*/
/* хранение в столбцовом порядке*/

@ Указатели:
@ R10 = указатель, куда будет сохранена результирующая матрица 4x4
@ R11 = указатель на первую матрицу 4x4, числа с плавающей точкой
@ одинарной точности
@ R12 = указатель на вторую матрицу 4x4, числа с плавающей точкой
@ одинарной точности
@ d16-d19 и d20-d23 (Q8, Q9, Q10, Q11) для матрицы 1
@ d8-d11 и d12-d15 (Q4, Q5, Q6, Q7) для матрицы 2
@ d24-d27 и d28-d31 (Q12-Q15) на выходе содержат результат

.global main
.func main

main:
    LDR R10, =result
    LDR R11, =matrix1
    LDR R12, =matrix2

    VLD1.32 {D16-D19}, [r11]!    @ Q8-Q9 загрузка матрицы 1, 2 строки
    VLD1.32 {D20-D23}, [r11]!    @ Q10-Q11 загрузка матрицы 1, 2 строки
    VLD1.32 {D8-D11}, [r12]!     @ Q4-Q5 загрузка матрицы 2, 2 строки
    VLD1.32 {D12-D15}, [r12]!    @ Q6-Q7 загрузка матрицы 2, 2 строки

    VADD.F32 Q12, Q8, Q4         @ Q12=Q8+Q4
    VADD.F32 Q13, Q9, Q5         @ Q13=Q9+Q5
    VADD.F32 Q14, Q10, Q6        @ Q14=Q10+Q6
    VADD.F32 Q15, Q11, Q7        @ Q15=Q11+Q7

```

```
VST1.32 {D24-D27}, [r10]!    @ D24-D27 первая восьмерка чисел
VST1.32 {D28-D31}, [r10]!    @ D28-D31 вторая восьмерка чисел
```

@ следующий код выводит матрицу результатов:

```
matrixprint:
```

```
.equ Num, 4                @ количество байтов
.equ Cells, 16             @ число ячеек матрицы
LDR R10, =result          @ R10 хранит адрес результирующей матрицы
MOV R7, #Cells            @ R7 содержит счетчик ячеек матрицы
```

```
loop:
```

```
LDR R0, [R10]             @ получение элемента данных в R0
VMOV S2, R0               @ перемещение в регистр FPU
VCVT.F64.F32 D0, S2      @ преобразование в формат одинарной точности
VMOV r2, r3, D0          @ помещение R2 и R3 для функции fprint
LDR R0, =string          @ в R0 помещается указатель на string
BL printf                 @ вызов функции fprint
ADD R10, #Num             @ увеличение адреса следующего результата
SUBS R7, #1              @ уменьшение счетчика ячеек
BNE loop                 @ если матрица не заполнена, обработка
                          @ следующей ячейки
```

```
MOV R7, #1                @ иначе - выход
SWI 0
```

```
string:                   .asciz "Result is: %f\n"
```

```
.data
```

```
matrix1:                 .single 106.6,2.4,4.5,6.2
                          .single 105,4,6,8
                          .single 104,6,8,10
                          .single 103,8,10,12
```

```
matrix2: .single 2,3,4,5
          .single 3,4,5,6
          .single 4,5,6,7
          .single 5,6,7,8
```

```
result: .word 0,0,0,0
         .word 0,0,0,0
         .word 0,0,0,0
         .word 0,0,0,0
```

Соберите и свяжите программу:

```
gcc -mfpu=neon-vfpv4 -g -o prog26c prog26c.s
```

Выполнив команду

```
./prog26c
```

вы увидите результат (листинг 26.3).

Листинг 26.3. Результат выполнения программы 26.3

```

Result is: 108.599998
Result is: 5.400000
Result is: 8.500000
Result is: 11.200000
    Result is: 108.000000
Result is: 8.000000
Result is: 11.000000
Result is: 14.000000
Result is: 108.000000
Result is: 11.000000
Result is: 14.000000
Result is: 17.000000
Result is: 108.000000
Result is: 14.000000
Result is: 17.000000
Result is: 20.000000

```

Опять же, если вы посмотрите на выполнение программы с помощью GDB, результаты можно будет увидеть наглядно. Обратите внимание, что в результатах, выведенных с помощью программы, также присутствует ошибка округления, которую мы обсуждали ранее.

В программе 26.3 содержится процедура, называемая *матричным выводом*, которую можно использовать для вывода на экран матрицы 4×4 . Она же включена и в программу 26.4 (см. далее), и ее можно адаптировать для простого вывода результатов или значений во время выполнения программы. Всегда приятно вывести данные из памяти или содержимое регистров, чтобы проверить, что у нас получилось. Так мы и делали раньше.

Матричное умножение

Существуют два типа матричного умножения: умножение на скаляр и умножение на матрицу. Со скалярным умножением все просто. Вы берете обычное число (называемое *скаляром*) и умножаете его на каждый элемент матрицы (рис. 26.15).

$$\begin{array}{|c|c|c|} \hline 0 & 1 & 2 \\ \hline 3 & 4 & 5 \\ \hline \end{array} \times \begin{array}{|c|} \hline 2 \\ \hline \end{array} = \begin{array}{|c|c|c|} \hline 0 & 2 & 4 \\ \hline 6 & 8 & 10 \\ \hline \end{array}$$

Рис. 26.15. Умножение матрицы на скаляр

При умножении матрицы на матрицу мы умножаем каждый из элементов строки в одной матрице на соответствующие элементы столбца в другой матрице, а затем суммируем полученные n произведений, что дает один элемент в результирующей матрице.

В программе 26.4 эта задача несколько упрощена за счет использования двух матриц равного размера, и в ней предполагается, что матрицы хранятся в памяти в столбцовом порядке. Процесс умножения показан на рис. 26.16, где приведены математические вычисления первого столбца. Прочие столбцы вычисляются так же.

	Столб. 0	Столб. 1	Столб. 2	Столб. 3		Столб. 0	Столб. 1	Столб. 2	Столб. 3
Строка 0	X0	X4	X8	X12	X	Y0	Y4	Y8	Y12
Строка 1	X1	X5	X9	X13		Y1	Y5	Y9	Y13
Строка 2	X2	X6	X10	X14		Y2	Y6	Y10	Y14
Строка 3	X3	X7	X11	X15		Y3	Y7	Y11	Y15

	Столбец 0
Строка 0	$=(X0*Y0)+(X4*Y1)+(X8*Y2)+(X12*Y3)$
Строка 1	$=(X1*Y0)+(X5*Y1)+(X9*Y2)+(X13*Y3)$
Строка 2	$=(X2*Y0)+(X6*Y1)+(X10*Y2)+(X14*Y3)$
Строка 3	$=(X3*Y0)+(X7*Y1)+(X11*Y2)+(X15*Y3)$

Рис. 26.16. Вычисление первого столбца (столбец 0) при умножении матриц

ПРОГРАММА 26.4. Матричное умножение чисел с одинарной точностью

```
/* Умножение матриц 4 x 4 одинарной точности в Neon*/
```

```
/* Порядок по столбцам */
```

```
@ Указатели:
```

```
@ R10 = указатель на место хранения результирующей матрицы 4x4
```

```
@ R11 = указатель на матрицу 1, в ней числа с плавающей точкой
```

```
@ одинарной точности
```

```
@ R12 = указатель на матрицу 2, в ней числа с плавающей точкой
```

```
@ одинарной точности
```

```
@ d16-d19 и d20-d23 (Q8, Q9, Q10, Q11) - матрица 1
```

```
@ d8-d11 и d12-d15 (Q4, Q5, Q6, Q7) - матрица 2
```

```
@ d24-d27 и d28-d31 (Q12-Q15) - результат
```

```
.global main
```

```
.func main
```

```
main:
```

```
LDR R10, =result
```

```
LDR R11, =matrix1
```

```
LDR R12, =matrix2
```

```
VLD1.32 {D16-D19}, [R11]! @ Q8-Q9 загрузка матрицы 1, 2 строки
```

```
VLD1.32 {D20-D23}, [R11]! @ Q10-Q11 загрузка матрицы 1, 2 строки
```

```

VLD1.32 {D8-D11}, [R12]!    @ Q4-Q5 загрузка матрицы 2, 2 строки
VLD1.32 {D12-D15}, [R12]!  @ Q6-Q7 загрузка матрицы 2, 2 строки

VMUL.f32 Q12, Q8, D8[0]     @ RC0 = (M1 C0) * (M2 C0 E0)
VMUL.f32 Q13, Q8, D10[0]    @ RC1 = (M1 C0) * (M2 C1 E0)
VMUL.f32 Q14, Q8, D12[0]    @ RC2 = (M1 C0) * (M2 C2 E0)
VMUL.f32 Q15, Q8, D14[0]    @ RC3 = (M1 C0) * (M2 C3 E0)

VMLA.f32 Q12, Q9, D8[1]     @ RC0 += (M1 C1) * (M2 C0 E1)
VMLA.f32 Q13, Q9, D10[1]    @ RC1 += (M1 C1) * (M2 C1 E1)
VMLA.f32 Q14, Q9, D12[1]    @ RC2 += (M1 C1) * (M2 C2 E1)
VMLA.f32 Q15, Q9, D14[1]    @ RC3 += (M1 C1) * (M2 C3 E1)

VMLA.f32 Q12, Q10, D9[0]    @ RC0 += (M1 C2) * (M2 C0 E2)
VMLA.f32 Q13, Q10, D11[0]   @ RC1 += (M1 C2) * (M2 C1 E2)
VMLA.f32 Q14, Q10, D13[0]   @ RC2 += (M1 C2) * (M2 C2 E2)
VMLA.f32 Q15, Q10, D15[0]   @ RC3 += (M1 C2) * (M2 C3 E2)

VMLA.f32 Q12, Q11, D9[1]    @ RC0 += (M1 C3) * (M2 C0 E3)
VMLA.f32 Q13, Q11, D11[1]   @ RC1 += (M1 C3) * (M2 C1 E3)
VMLA.f32 Q14, Q11, D13[1]   @ RC2 += (M1 C3) * (M2 C2 E3)
VMLA.f32 Q15, Q11, D15[1]   @ RC3 += (M1 C3) * (M2 C3 E3)

VST1.32 {D24-D27}, [R10]!   @ d24-d27 первая восьмерка чисел
VST1.32 {D28-D31}, [R10]!   @ d28-d31 вторая восьмерка чисел

```

@ следующий код выводит матрицу результатов, если требуется
matrixprint:

```

.equ Num, 4                @ количество байтов
.equ Cells, 16             @ число ячеек матрицы

LDR R10, =result           @ R10 хранит адрес
                           @ результирующей матрицы
MOV R7, #Cells             @ R7 содержит счетчик ячеек матрицы

```

loop:

```

LDR R0, [R10]              @ получение элемента данных в R0
VMOV S2, R0                @ перемещение в регистр FPU
VCVT.F64.F32 D0, S2       @ преобразование в формат
                           @ одинарной точности
VMOV R2, R3, D0           @ помещение R2 и R3 для функции fprintf
LDR R0, =string            @ в R0 помещается указатель на string
BL printf                  @ вызов функции fprintf
ADD R10, #Num              @ увеличение адреса следующего результата
SUBS R7, #1                @ уменьшение счетчика ячеек
BNE loop                   @ если матрица не заполнена, обработка
                           @ следующей ячейки
                           @ иначе — завершение

```

```
MOV R7, #1           @ ВЫХОД
SWI 0
```

```
string:      .asciz "Result is: %f\n"
.data
Matrix1:     .single 106.6,2.4,4.5,6.2
             .single 105,4,6,8
             .single 104,6,8,10
             .single 103,8,10,12

Matrix2:     .single 2,3,4,5
             .single 3,4,5,6
             .single 4,5,6,7
             .single 5,6,7,8

result:      .word 0,0,0,0
             .word 0,0,0,0
             .word 0,0,0,0
             .word 0,0,0,0
```

В этой программе каждый столбец матрицы загружается в регистр Neon. Для вычисления результата для каждого столбца мы можем применить инструкцию векторно-скалярного умножения (`VMLA`). Мы также должны сложить результаты каждого элемента столбца, а для этого используем накопительную версию той же команды.

Помните, что регистры `D` связаны с регистрами `Q`, так что мы в любом случае можем получить доступ к содержимому этих регистров, зная, что `Q0` — это комбинация `D0` и `D1`, и т. д.

Сначала выполняется загрузка первых восьми элементов матрицы 1 в `D16–D19` и следующих восьми ее элементов в `D20–D23`, которые соответствуют `Q12` и `Q13`. После чего мы загружаем содержимое матрицы 2 в `D0–D7` (`Q0–Q1`).

Затем выполняется остальная часть программы, в которой вычисляется каждый один столбец с использованием всего четырех команд Neon:

```
VMUL.F32 Q12, Q8, D8[0]
VMLA.F32 Q12, Q9, D8[1]
VMLA.F32 Q12, Q10, D9[0]
VMLA.F32 Q12, Q11, D9[1]
```

Первая команда (`VMUL.F32`) принимает значения `x0`, `x1`, `x2` и `x3` (в регистре `Q8`), каждое из них умножается на `y0` (элемент 0 в `D0`), и результат сохраняется в `Q12`. Три последующие команды `VMLA.F32` работают с тремя другими столбцами первой матрицы, умножая их на соответствующие элементы первого столбца второй матрицы. Результаты накапливаются в `Q12`, давая тем самым первый столбец значений для матрицы результатов.

Этот набор команд необходимо выполнить еще трижды, чтобы вычислить второй, третий и четвертый столбцы. Здесь мы используем значения от `Y4` до `Y15` из второй матрицы в регистрах с `Q1` по `Q3`.

Повторяющиеся команды идеально подходят для реализации в качестве макроса, однако, хотя код и проще некуда, возможны проблемы синхронизации из-за так называемого *эффекта планирования*. Часто, когда смежные команды умножения, которые записывают что-то в один и тот же регистр, находятся рядом друг с другом, процессор Neon вынужден ждать полного завершения первой операции, прежде чем он сможет переместить на конвейер следующую.

Если вы разделите команды таким образом, чтобы их работа не перекрывала доступ к регистрам Neon, их можно продолжать подавать в конвейер Neon, не теряя при этом скорости и реализуя по-настоящему параллельные операции.

Как и в предыдущей программе, для отображения результатов матрицы здесь используется процедура `matrixprint`.

Соберите и свяжите программу:

```
gcc -mfpu=neon-vfpv4 -g -o prog26d prog26d.s
```

Выполнив команду

```
./prog26d
```

вы увидите результат (листинг 26.4).

Листинг 26.4. Результат выполнения программы 26.4

```
Result is: 1459.199951
Result is: 80.800003
Result is: 109.000000
Result is: 136.399994
Result is: 1877.800049
Result is: 101.199997
Result is: 137.500000
Result is: 172.600006
Result is: 2296.399902
Result is: 121.599998
Result is: 166.000000
Result is: 208.800003
Result is: 2715.000000
Result is: 142.000000
Result is: 194.500000
Result is: 245.000000
```

На рис. 26.17 показаны обе матрицы, умножение которых производится в программе 26.4, а также третья матрица результатов: $(M1 * M2)$. Обратившись к рис. 26.16, мы увидим, что верхняя левая ячейка рассчитывается следующим образом:

$$\begin{aligned}
 &= (X3*Y0) + (X7*Y1) + (X11*Y2) + (X15*Y3) \\
 &= (106.6*2) + (105*3) + (104*4) + (103*5) \\
 &= 1459.2
 \end{aligned}$$

Матрица 1				Матрица 2					
	Столб. 0	Столб. 1	Столб. 2	Столб. 3	x	Столб. 0	Столб. 1	Столб. 2	Столб. 3
Строка 0	106.6	105	104	103		2	3	4	5
Строка 0	2.4	4	6	8		3	4	5	6
Строка 0	4.5	6	8	10		4	5	6	7
Строка 0	6.2	8	10	12		5	6	7	8

Результат (M1 x M2)			
1459.2	1877.8	2296	2715
80.8	101.2	121.6	142
109	137.5	166	194.5
136.4	172.6	208.8	245

26.17. Умножение матриц

Пример использования макроса

Если вы удалите из кода программы 26.4 процедуру `matrixprint`, в коде вычислений будет много повторений. А это означает, что здесь удобно использовать макрос. Приведенный в этой программе пример нам важен, т. к. позволяет проиллюстрировать использование Neon для выполнения процесса умножения. Теперь, когда мы с этим разобрались, давайте посмотрим на версию того же кода, но уже с макросами. Код этот приведен в программе 26.5, и большинство комментариев из него для простоты убрано. Сам макрос включает всего шесть строк кода, включая директивы.

ПРОГРАММА 26.5. Умножение матриц, но с макросами

```

/* Умножение матриц 4x4 */
/* Используем макросы для сокращения кода*/

.global main
.func main
main:
    LDR R10, =result
    LDR R11, =matrix1
    LDR R12, =matrix2

    .macro matrixf32 resultQ, col0_d, coll_d
    VMUL.f32    \resultQ, Q8,    \col0_d[0]    @ элемент 0 матрицы C0
    VMLA.f32    \resultQ, Q9,    \col0_d[1]    @ элемент 1 матрицы C1
    VMLA.f32    \resultQ, Q10,   \coll_d[0]    @ элемент 2 матрицы C2
    VMLA.f32    \resultQ, Q11,   \coll_d[1]    @ элемент 3 матрицы C3
    .endm

```

```
VLD1.32 {D16-D19}, [R11]! @ первые восемь элементов M1
VLD1.32 {D20-D23}, [R11]! @ вторые восемь элементов M1
VLD1.32 {D0-D3}, [R12]!   @ первые восемь элементов M2
VLD1.32 {D4-D7}, [R12]!   @ вторые восемь элементов M2
```

@ Вызов макроса

```
matrixf32 Q12, D0, D1      @ матрица 1 * матрица 2 - столбец 0
matrixf32 Q13, D2, D3      @ матрица 1 * матрица 2 - столбец 1
matrixf32 Q14, D4, D5      @ матрица 1 * матрица 2 - столбец 2
matrixf32 Q15, D6, D7      @ матрица 1 * матрица 2 - столбец 3
```

```
VST1.32 {D24-D27}, [R10]! @ сохранение первых 8 элементов результата
VST1.32 {D28-D31}, [R10]! @ сохранение вторых 8 элементов результата
```

@ сюда вставляется код вывода матрицы, если нужно вывести результат

@ полный код программы доступен на сайте www.brucesmith.info

```
MOV R7, #
SWI 0
```

```
string:      .asciz "Result is: %f\n"
```

.data

```
matrix1:     .single 106.6,2.4,4.5,6.2
             .single 105,4,6,8
             .single 104,6,8,10
             .single 103,8,10,12
```

```
matrix2:     .single 2,3,4,5
             .single 3,4,5,6
             .single 4,5,6,7
             .single 5,6,7,8
```

```
result:      .word 0,0,0,0
             .word 0,0,0,0
             .word 0,0,0,0
             .word 0,0,0,0
```

27. Код Thumb

Thumb — это название некоторого подмножества набора команд ARM. Что еще более важно, это 16-битная (двухбайтовая) реализация, поэтому для кодирования инструкций теоретически требуется вдвое меньше места, чем у эквивалентной программы ARM. Но на самом деле экономия составляет примерно треть. Более высокая плотность кода делает код Thumb популярным в задачах, где существуют жесткие ограничения памяти. На форумах вы, вероятно, редко встретите программы на Thumb, главным образом потому, что большая часть кода Thumb пишется и компилируется из C. Но это не значит, что мы не можем писать его вручную.

С точки зрения аппаратного обеспечения никакой разницы между тем, как работают наборы команд ARM и Thumb, нет. Они суть одно и то же. Разумеется, Thumb — это 16-битная реализация, но размеры регистров остаются прежними. R0, как и другие регистры, — это всего лишь слово данных. Разница лишь в том, как эти данные извлекаются и интерпретируются перед выполнением. Аппаратное обеспечение само расширяет команды Thumb до 32-битных вариантов, поэтому выполнение кода не замедляется и ARM работает все так же быстро. Поэтому нам ничто не мешает комбинировать разделы обычного кода ARM с Thumb и переходить от одного к другому. Фактически вводить код Thumb лучше всего именно путем перехода от ARM к Thumb.

Вернемся к *главе 5* и посмотрим на рис. 5.3, где показана схема устройства регистра состояния. Бит 5 — это бит T, и обычно он сброшен, обозначая тем самым режим ARM. Когда бит T установлен ($T = 1$), микросхема находится в режиме Thumb. Мы рассмотрим, как можно переключаться между состояниями, и напишем простую программу, которую можно использовать в качестве оболочки для любого кода Thumb, который вам понадобится в будущем. Но сначала...

Различия

Мы подробно познакомимся с набором команд Thumb, но между ним и ARM есть различия, о которых необходимо знать. Понимая их и вооружившись знаниями ARM, вы избежите множества трудностей с реализацией и написанием программы Thumb (а знания ARM у вас к настоящему моменту уже весьма обширны).

И, конечно же, GCC и GDB тоже поддерживают Thumb, поэтому с инструментами также полный порядок.

Основное архитектурное различие этих наборов команд состоит в том, что код на Thumb не имеет прямого доступа ко всем регистрам ARM. Вам доступны лишь регистры с R0 по R7 включительно. Регистры с R8 по R12 включительно могут использоваться только с командами MOV, ADD, SUB и CMP. Доступ к R13 (SP), R14 (LR) и R15 (PC) ограничен, доступ к CPSR осуществляется только косвенно. Доступа к SPSR нет, а команды VFP недоступны.

Обращаться к недоступным регистрам можно будет лишь после того, как программа вернется в режим ARM. Другими словами, вы должны сначала выйти из режима Thumb, выполнить нужную операцию, а затем снова переключиться в режим Thumb, чтобы продолжить работу. На рис. 27.1 схематически показаны эти ограничения регистра. Преимущество всего этого заключается в том, что при переходе между режимами ARM и Thumb содержимое регистров сохраняется!

Еще одно существенное различие состоит в том, что мнемонические представления команд Thumb короче, и зачастую в них на один операнд меньше. Сравните команды ADD у ARM и Thumb:

ADDS R2, R2, #16	@ сложение в ARM
ADD R0, #3	@ та же команда в Thumb, при этом регистр назначения задан неявно

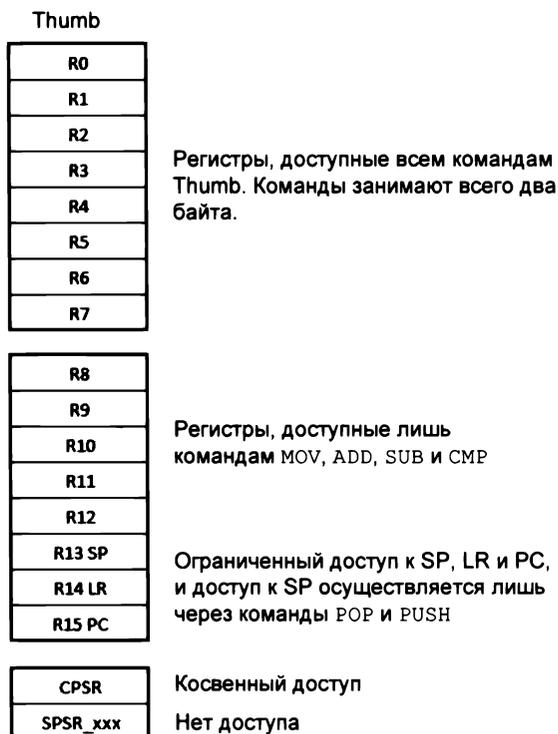


Рис. 27.1. Доступность регистров Thumb

Модификаторы условного выполнения команд использовать нельзя, условно выполняются только команды ветвления. Это значит, что в режиме Thumb вы не можете выполнять такие команды, как

```
ADD CC R0, #3
```

Операторы сдвига и поворота: ASR, LSL, LSR и ROR — реализованы как отдельные команды и уже не являются модифицирующими операндами. В следующем фрагменте кода показан формат их использования:

```
LSL R2, R3 @ смещение R2 влево на указанное в R3 количество позиций
```

В приведенном примере, если $R2 = 4$ и $R3 = 1$, то $R2$ станет равен 8.

Команды ветвления в Thumb несколько ограничены в пространстве. Вариант, используемый в приведенной в конце следующего раздела *программе 27.1*, — это единственный условный код, а диапазон работы здесь ограничен меткой, которая должна находиться в пределах однобайтового значения со знаком, т. е. от -256 до 254 . Команда безусловного перехода работает в пределах 11-битового значения со знаком: от -2048 до $+2046$ байтов.

Команда BL не является условной, но, поскольку ее можно использовать косвенным образом, возможен диапазон адресов до 4 Мбайт в любом направлении.

Имеются также значительные изменения в некоторых командах загрузки/хранения и доступа к стеку, и о них мы поговорим отдельно.

Пишем на Thumb

Для переключения между состояниями ARM и Thumb используются директивы GCC:

```
.arm
```

или

```
.thumb
```

Эти директивы заменяют популярные ранее директивы `.code32` и `.code16`, которые в текущей версии GCC на Raspberry Pi все еще работают, но теперь считаются устаревшими. При необходимости директивы `.arm` и `.thumb` автоматически вставляют до трех заполняющих байтов для выравнивания по границе следующего слова для ARM или до одного байта для выравнивания по границе следующего полуслова для Thumb. Таким образом, директива `.align` уже не нужна.

Директивы `.arm` и `.thumb` говорят ассемблеру, что нужно компилировать. Сами по себе они не выполняют никаких команд, а лишь сообщают ассемблеру, что будет дальше. Вместо прямого изменения бита 5 непосредственно в CPSR, изменение состояния будет обрабатываться непосредственно для вас, если вы будете следовать для этого правильному протоколу, который включает использование инструкции `WX` (см. далее).

По сути, если вы загрузите адрес ссылки начала кода Thumb в $R0$ и установите младший бит $R0$ (т. е. $b0 = 1$), то состояние Thumb будет вызываться автоматически

по достижении кода Thumb с помощью команды `bx`. То есть код всегда сначала работает в режиме ARM, и без этого не обойтись, поскольку именно в этом состоянии запускается микросхема. В *программе 27.1* показано, как это работает на практике (обратите внимание на использование символа `@` перед комментариями. Код GCC Thumb не любит включение комментариев в стиле `/* */`, и иногда это ведет к ошибкам).

ПРОГРАММА 27.1. Вызов режима Thumb и запуск кода Thumb

```
@ Использование кода Thumb на Raspberry Pi
@ Процедура деления R0/R1
@ в R2 помещается MOD, а в R3 - DIV

.global main
.func main
.arm
main:
    ADR R0, thumbcode+1
    MOV LR, PC
    BX R0

exit:
    MOV R0, #0
    MOV R7, #1
    SWI 0

    @ Здесь начинается код Thumb
    .thumb

thumbcode:
    MOV R3, #0

loop:
    ADD R3, #1
    SUB R0, R1
    BGE loop
    SUB R3, #1
    ADD R2, R0, R1
    BX LR                                @ Возврат к коду ARM
```

Команды Thumb начинаются с метки `thumbcode:`. Ее адрес загружается в `R0` в начале блока `main:`, и к адресу добавляется 1, чтобы установить младший бит `R0`. Выполняется команда `bx` (Branch with eXchange, переход с обменом), и адрес из `R0` попадает в `PC`. Поскольку команды ARM и Thumb выровнены по слову или полуслову соответственно, биты 0 и 1 адреса игнорируются, т. к. они лежат в третьем байте адреса.

Команды в блоке `thumbcode` выполняют процедуру деления (хотя мы не давали им для этого никаких данных для работы), после чего `BX LR` возвращает код обратно вызывающему коду ARM. Мы снова входим в режим ARM, т. к. нужно выполнить инструкцию ARM. Обратите внимание, что только из режима ARM можно выполнить команду `SWI` или функцию вроде `printf`.

В конце подпрограммы ARM нужно использовать инструкцию `BX LR`, чтобы вернуться к вызывающей стороне. Использовать команду `MOV PC, LR` для возврата нельзя, поскольку она не будет обновлять бит `T` и, следовательно, не изменит состояние.

Стоит взглянуть на то, как GDB видит собранную программу, поскольку именно так легко увидеть различия, описанные ранее. В листинге 27.1 показан вывод GDB с использованием команды

```
x /13i main
```

Как и обычно, адреса, указанные в первом столбце, у вашей Raspberry Pi могут быть другими (вы можете использовать GCC для компиляции, ничего особенного не требуется).

Листинг 27.1. Результат дизассемблирования программы 27.1

```
(gdb) x /13i main
0x103d0 <main>:      add    r0, pc, #17
0x103d4 <main+4>:     mov    lr, pc
0x103d8 <main+8>:     bx     r0
0x103dc <exit>:      mov    r0, #0
0x103e0 <exit+4>:    mov    r7, #1
0x103e4 <exit+8>:    svc   0x00000000
0x103e8 <thumbcode>: movs  r3, #0
0x103ea <loop>:      adds  r3, #1
0x103ec <loop+2>:    subs  r0, r0, r1
0x103ee <loop+4>:    bge.n 0x103ea <loop>
0x103f0 <loop+6>:    subs  r3, #1
0x103f2 <loop+8>:    adds  r2, r0, r1
0x103f4 <loop+10>: bx     lr
```

Обратите внимание, что к командам работы с данными в разделе Thumb `thumbcode` применяется суффикс `s`. При работе с регистрами `R0–R7` все команды обработки данных обновляют флаги состояния, поэтому суффикс `s` применяется автоматически.

Также заметьте, что к команде `BGE` добавлен суффикс `.n`. В Thumb он инструктирует ассемблер генерировать 16-битную кодировку, и если команда не может быть закодирована 16 битами, ассемблер выдает ошибку.

Процесс объединения кода ARM и Thumb называется *взаимодействием*, и, если нужно, вы можете писать код, работающий на обоих наборах команд попеременно.

Всегда можно использовать одну и ту же связку команд: ADR/BX. Например, если подпрограмма ARM начинается с метки `armroutine`, ее можно было бы вызвать следующим образом:

```
ADR R0, armroutine
BX R0                @ переход на ветку armroutine
```

Перед выполнением любых команд, действующих на регистр ссылок, нужно сохранять его содержимое в стеке, чтобы можно было вернуться к исходной точке входа, а затем вызвать основную программу, для правильного завершения своего потока.

В приведенных здесь примерах используется команда `BX`, но есть и команда `BLX`, которая также позволяет переключаться на код Thumb. Эта команда автоматически сохраняет содержимое регистра `PC` в `LR`, поэтому команда `MOV LR, PC` уже не нужна. Но нам по-прежнему надо прибавлять 1 к адресу входа для переключения состояния:

```
main:
ADR R0, thumbcode+1
BLX R0
```

Впрочем, команду `BX` по-прежнему можно использовать для возврата из любой вызванной подпрограммы.

В *программе 27.2* показано, как наборы команд ARM и Thumb объединяются на практике и как с помощью функции `printf` мы выводим на экран результат деления, выполненный в этой расширенной программе. Обратите внимание, что за кодом Thumb снова следует код ARM. Это необходимо, иначе компилятор выдаст ошибку при попытке создать относительный адрес ветвления для кода ARM. Также заметьте, что перед меткой, обозначающей раздел соответствующего кода, должны стоять директивы `.arm` и `.thumb`.

ПРОГРАММА 27.2. Использование внешних функций путем взаимодействия кода

@ Взаимодействие кода ARM и Thumb для вызова `printf`

```
.global main
.func main
.arm
main:
ADR R5, thumbstart+1
BX R5

.thumb

thumbstart:
MOV R0, #9                @ деление 9 на 3
MOV R1, #3
MOV R3, #0
```

```

loop:
    ADD R3, #1
    SUB R0, R1
    BGE loop
    SUB R3, #1           @ R2=MOD
    ADD R2, R0, R1     @ R3=DIV
    ADR R5, divprint
    BX R5

thumbreturn:
    @ Здесь добавляется весь нужный код
    @ и вызываются функции ARM, если нужны
    ADR R5, exit
    BX R5               @ возврат на метку exit

    .arm
divprint:
    LDR R0, =string
    MOV R1, R3          @ DIV и MOD уже лежат в R2
    BL printf
    ADR R5, thumbreturn+1
    BX R5

exit:
    MOV R7, #1
    SWI 0

.data
string:    .asciz "Result of 9/3 is: %d MOD %d\n"

```

Доступ к старшим регистрам

Получить доступ к полному набору регистров ARM позволяет всего лишь несколько команд. Как уже говорилось, большинство команд Thumb ограничены регистрами R0–R7 и при этом автоматически обновляют CPSR. Далее приведены используемые для доступа к старшим регистрам (R8–R14 и PC) команды и их формат. Не считая команды CMP, эти команды не обновляют CPSR.

- ◆ MOV <Регистр_назначения> <Операнд1>;
- ◆ ADD <Регистр_назначения> <Операнд1>;
- ◆ CMP <Операнд1>, <Операнд2>;
- ◆ ADD <Регистр_назначения> <Операнд1> | <#константа>;
- ◆ ADD <Регистр_назначения> <Операнд1>, <Операнд2> | <#константа>;
- ◆ SUB <Регистр_назначения> <Операнд1> | <#константа>;
- ◆ SUB <Регистр_назначения> <Операнд1>, <Операнд2> | <#константа>.

Операторы стека

Команды стека Thumb сильнее прочих не похожи на команды из набора ARM, поэтому фактически используются более традиционные PUSH и POP. Мы видели это раньше, поскольку они предоставляются компилятором GCC как псевдоинструкции. В Thumb они работают так же, поэтому вам будет несложно с ними разобраться. Но есть и существенная разница — у этих команд нет доступа к указателю стека (SP). Это связано с тем, что в операциях Thumb указателем стека является R13, и именно он автоматически обновляется командами:

```
PUSH {R1-R4}      @ отправление R1, R2, R3 и R4 в стек
POP {R2-R3}       @ 2 верхних элемента из R2 и R3
```

В команде PUSH можно использовать LR, а в POP — PC, а иначе используются только R0–R7. В первом случае адрес SP настраивается четырьмя словами, во втором — двумя. В терминах ARM PUSH выполняет следующее:

```
STMDB SP!, <REGLIST>
```

а POP:

```
LDMIA SP!, <REGLIST>
```

Одно- и многорегистровые команды

Thumb поддерживает команды LDR и STR, но работают не все режимы адресации. Фактически с этими и связанными с ними командами используется только три режима. Они приведены в табл. 27.1 — это режим предварительно индексированной адресации, смещение по регистрам или с помощью заданного константой операнда.

Доступ с несколькими регистрами осуществляется только за счет использования приращения после режимов адресации с помощью команд LDMIA и STMIA. Также обратите внимание, что знак обновления оператора ! уже обязателен к применению, в отличие от режима ARM:

```
STMIA R1!, {R2, R3, R4}
```

Таблица 27.1. Примеры адресации в режиме Thumb

Режим адресации	Пример
Загрузка/сохранение регистра	LDR R0, R1
Базовое значение и смещение	LDR R0, [R1, #5] LDR R0, [R1, R2]
Косвенная адресация	LDR R0, [PC, #8]

Функции в Thumb

Пример, приведенный в программе 27.2, показывает, как из кода Thumb можно вызвать функцию ARM или, если точнее, функцию libc. Как мы уже говорили ранее, для этого нужно снова вернуться в режим ARM. Ничто не мешает вам создавать

в коде Thumb свои собственные функции, полностью написанные на Thumb и выполняемые исключительно в этом режиме. Однако при вызове функции у нее должен быть установлен младший значащий бит указателя на нее. Поскольку компоновщик в компиляторе сам этого сделать не может, вы вручную должны сделать это в вызывающем коде, особенно если вы используете абсолютный адрес.

Таким образом, когда вы вызываете любой автономный код Thumb из другого раздела кода Thumb, переход выполняется так же, как если бы вы переходили в код Thumb из режима ARM.

Кстати, за счет этого у вас может быть две функции с одним и тем же именем: одна для ARM, а другая для Thumb. Компоновщик позволяет так делать при условии, что эти функции работают с разными наборами команд. Но, несмотря на возможность, это плохая практика, и ее следует избегать.

Команды ARMv7 Thumb

В рамках выпуска архитектуры ARMv7 (версии Raspberry Pi 2B и новее) в набор команд Thumb было добавлено семь новых команд, наиболее значимой из которых является команда `CBNZ` (Compare and Branch on Zero, сравнить и перейти на ноль) или Non-Zero. Эта команда сравнивает значение в регистре с нулем и условно переходит к постоянному значению. Она не влияет на флаги условий:

```
CBNZ R0, newdest    @ R0 <> 0, переход к 'newdest'
CBZ  R0, next       @ R0 = 0, переход к 'next'
```

Обе команды поддерживают суффиксы `.N` и `.W`, измененные следующим образом:

```
CBZ.N R0, next
```

Суффикс `.N` (Narrow, узкий) сообщает ассемблеру, что нужно сгенерировать 16-битную кодировку. Суффикс `.W` (Wide, широкий) означает 32-битную кодировку. По умолчанию выбрано `.N` (16 битов), тогда как сборка в режиме A32 дает 32-битную кодировку.

Команда `NOP` означает No Operation и не делает ничего, но может выступать в роли заполнителя, необходимого, чтобы следующая команда правильно расположилась на 64-битной границе. Кроме того, она задерживает программу на один цикл.

Также появились команды `YIELD`, `SEV`, `WFE` и `WFI`, которые связаны с возникающими событиями и прерываниями.

Еще одна новая команда `IT` (If/Then) позволяет выполнять до четырех последовательных команд в зависимости от выполнения или невыполнения переданного условия. Команда `IT` игнорируется при компиляции в код ARM, а при компиляции в Thumb генерирует фактическую инструкцию. Например:

```
CMP R0, R1    @ если (R0 == R1)
ITE EQ       @ то R0 = R2;
MOVEQ R0, R2 @ Thumb: условие через ITE 'T' (то)
              @ иначе - R0 = R3;
MOVNE R0, R3 @ Thumb: условие через ITE 'E' (иначе)
```

28. Единый язык

В *главе 1* мы рассмотрели различия между 32-битными и 64-битными версиями микросхемы ARM и ввели термины A32 и A64, чтобы различать 32- и 64-битные ARM.

Набор команд Thumb-2 появился в версии ARMv6T2 (и был включен в последующие выпуски). Он не только добавил новые команды в базовый набор, но также предоставил возможность условно выполнять большинство новых команд Thumb. Thumb-2 — это своего рода компромисс, где взято «лучшее из обоих миров», поскольку в нем есть доступ как к 16-битным, так и к 32-битным командам, что позволяет программистам, которым не хватает памяти, извлекать максимум из наличного ограниченного числа байтов. Таким образом, набор Thumb-2 стал существенной вехой на пути к развитию унифицированного языка ассемблера (Unified Assembler Language, UAL).

UAL — это стандартная синтаксическая модель для ARM, реализованная, начиная с ARMv7, для команд A32 и T32. Она заменяет более ранние версии языков ассемблера A32 и T32. Код, написанный на UAL, можно собрать в режиме A32 или T32 для любого процессора ARM. Однако не каждый ассемблер «знает» этот код полностью. Впрочем, GCC обычно правильно выполняет и собирает код, написанный в формате UAL.

В версиях от ARMv4T до ARMv7-A используются два набора команд: ARM и Thumb. Оба они являются «32-битными» в том смысле, что работают с данными шириной до 32 битов в 32-битных регистрах с 32-битными адресами. Фактически оба набора используют одни и те же команды, а различается только кодировка команд. У ЦП фактически есть два разных интерфейса декодирования для своего конвейера, между которыми он может переключаться.

Комбинируя наборы команд A32 и T32, UAL формирует согласованную модель программирования. В результате вы пишете максимально экономичный и производительный код. UAL вносит некоторые изменения как в код ARM, так и в код Thumb, приводя их к общему стандарту, а также вводит новые команды. Использование нового синтаксиса может повлиять на обратную совместимость кода, если ваш ассемблер недостаточно умен, чтобы понять, как с ним работать.

Компилятор GCC (и, я думаю, это касается и других совместимых компиляторов) сможет собирать код, написанный на синтаксисе pre-UAL и UAL. Если вы хотите

использовать UAL, в вашем коде — в числе прочих определений, приведенных в его начале, — должна присутствовать директива

```
.syntax unified
```

Вам также может потребоваться указать архитектуру, которую вы используете (об этом чуть позже).

T32 расширяет набор команд Thumb возможностью работы с битовыми полями, переходами в таблицах и операциями условного выполнения. Набор команд ARM также был расширен, чтобы в обоих наборах команд был одинаковый набор функций.

В результате и получился язык UAL, который поддерживает сборку команд Thumb и ARM из одного и того же исходного кода. Версии Thumb, впервые представленные на процессорах ARMV7, по сути, имеют ту же функциональность, что и код ARM (включая возможность писать обработчики прерываний).

С учетом сказанного нужно переосмыслить концепцию Thumb-2. Как уже отмечалось ранее, в ARMv6 есть два набора команд: ARM (A32) и Thumb-2 (T32). Оба они являются «32-битными» в том смысле, что работают с данными шириной до 32 битов в 32-битных регистрах с 32-битными адресами. В функциональном смысле у них одни и те же команды, отличается только кодирование команд, а у ЦП заведено два разных интерфейса конвейера, между которыми он может переключаться при декодировании команды.

T32 не только включает в себя дополнительные команды (в основном с 4-байтовым кодированием, хотя есть и несколько двухбайтовых кодировок), что дает почти полную эквивалентность ARM, но также допускает условное выполнение большинства команд Thumb. Смешанный 16/32-битный поток команд позволяет одновременно получить экономию места от Thumb и большую скорость чистого кода от ARM. Изначально цель T32 заключалась в достижении плотности кода, соизмеримого с Thumb, а также производительности, подобной набору команд ARM.

Если в вашей задаче крайне важна производительность, то для достижения максимальной скорости важно, чтобы хотя бы половина команд была закодирована в 16-битном виде.

Общие правила генерации 16-битной формы команд следующие:

- ◆ нужно использовать регистры R0–R7;
- ◆ по возможности устанавливать флаги условий, если команда не является условной;
- ◆ использовать значения-константы в диапазоне 0–255.

Изменения Thumb

В исходный синтаксис Thumb были внесены изменения, чтобы подогнать T32 под A32. В исходном коде Thumb, где первый операнд и операнд назначения совпадали, достаточно указать его лишь один раз. Теперь же указываются оба операнда:

```
ADD R0, R1    @ старый формат Thumb
ADD R0, R0, R1 @ Формат UAL
```

Аналогичным образом, когда команда устанавливает флаги условия, это делается стандартным суффиксом *S*:

```
ADD R0, R1, R2 @ старый формат Thumb
ADDS R0, R1, R2 @ Формат UAL
```

Теперь команда *MOV* используется как операция сложения с нулем в виде константы:

```
MOV R0, R1    @ старый формат Thumb
ADD R0, R1, #0 @ Формат UAL
```

И чтобы жизнь медом не казалась, команда *CPY* превращается в *MOV*:

```
CPY R0, R1    @ старый формат Thumb
MOV R0, R1    @ Формат UAL
```

Постинкрементный режим адресации задан для команды *LDM* по умолчанию:

```
LDMIA R0!, {R0, R1} @ старый формат Thumb
LDM R0!, {R1, R2}   @ Формат UAL
```

В *LDM* явно не указывается обратная запись, если работа идет с базовым регистром:

```
LDMIA R0!, {R0, R1} @ старый формат Thumb
LDM R0, {R0, R1}   @ Формат UAL
```

Новые команды A32

В табл. 28.1 показаны новые команды, добавленные к набору команд A32 в UAL. У набора T32 команды *VL* и *VLX* являются 32-битными операциями. Далее приведено несколько примеров команд, которые также можно использовать с дополнительным условием, если требуется. Разрешено и использование директив *.N* и *.W*.

Таблица 28.1. Новые команды A32

Команда	Действие
BFC	Очистка бита
BFI	Вставка бита
MLS	Умножить и вычесть
MOV	Широкий вариант команды, которая загружает 16-битную константу в биты 0–15 регистра
MOVT	Загружает 16-битную константу в биты 16–31 регистра
RBIT	Инверсия битов в слове
SBFX	Побитовое извлечение со знаком
UBFX	Побитовое извлечение без знака

Команда `BFC` очищает любое количество соседних битов в любой позиции регистра, не затрагивая другие биты:

```
BFC R0, #5, #3
```

Здесь эта команда очищает биты 5, 6 и 7 значения, хранящегося в `R0`.

Команда `BFI` копирует любое количество младших битов из регистра в те же биты указанного регистра назначения:

```
BFI R0, R1, #5, #3
```

Здесь эта команда копирует биты 5, 6 и 7 из `R1` в биты 5, 6 и 7 из `R0`.

Команда `MLS` перемножает значения двух регистров, а затем вычитает 32-разрядные младшие разряды результата из третьего регистра и записывает результат в регистр назначения:

```
MLS R0, R1, R2, R3
```

Здесь эта команда перемножает содержимое `R1` и `R2`, вычитает младшие 32 бита из `R3` и помещает результат в `R0`.

Сравнение по нулю

Весьма часто при написании программ используется сравнение по нулю. В табл. 28.2 показаны различия в использовании этой команды в наборе ARM, T32 и T16.

Таблица 28.2. Сравнение по нулю: варианты кода

Режим	Мнемоника	Длина
ARM	<code>CMP R0, #0 ; BEQ<метка></code>	8 байтов
T32	<code>CMP R0, #0 ; BEQ<метка></code>	4 байта
T16	<code>CBZ R0, <метка></code>	2 байта

Как можно видеть, в строке T16 сочетание команд `CMP + BEQ` из ARM и Thumb заменено одной командой `CBZ` длиной в 2 байта.

Сборка UAL

Процессор ARM может работать только в режимах ARM или Thumb. Это касается даже процессоров с поддержкой pre-UAL. Смешивать код A32 и T32 нельзя. Чтобы использовать код A32 и T32 в одной программе, код пишется в разных блоках, а затем вы должны переключаться между состояниями в нужных местах. Используемая для этого методика под названием *взаимодействие* была рассмотрена в предыдущей главе. Разница в том, что для вызова UAL нужно использовать директиву

```
.syntax unified
```

в верхней части файла с кодом в разделе директив. Когда вы компилируете исходный код, вы также должны указать архитектуру, для которой компилируете, например:

```
march=armv8-a
```

Код, написанный на UAL, может быть собран для A32 или T32 на любом процессоре ARM, где эта техника поддерживается.

Как мы видели, некоторые команды T32 могут иметь как 16-битную, так и 32-битную кодировку. Если вы не укажете размер команды, по умолчанию принимается следующее:

- ◆ для прямых ссылок в командах LDR, ADR и в компилятор генерирует 16-разрядную команду, даже если это приведет к сбою в задаче, которая может быть достигнута с помощью 32-разрядной команды;
- ◆ для внешних ссылок LDR и в компилятор должен сгенерировать 32-битную команду;
- ◆ во всех остальных случаях компилятор должен генерировать кодировку наименьшего возможного размера.

Если вы хотите переопределить эти значения по умолчанию, можно использовать суффиксы `.w` или `.N` — чтобы указать конкретную нужную ширину команды (широкую или узкую). Суффикс `.w` игнорируется при сборке кода A32, поэтому вы можете безопасно использовать его в программах, которые подразумевают сборку в код A32 или T32.

Поскольку команда может быть как 16-битной, так и 32-битной, важно следить за выравниванием адресов. Начиная с архитектуры ARMv7, бит A в регистре управления системой (SCTLR) управляет включением или отключением проверки выравнивания. Исключение составляет процессор ARMv7-M, где этим занимается UNALIGN_TRP — бит 3 в регистре конфигурации и управления (CCR).

Если включена проверка выравнивания, все невыровненные слова и полуслова вызывают исключение. Если проверка отключена, в невыровненном варианте могут выполняться команды LDR, LDRH, STR, STRH, LDRSH, LDRT, STRT, LDRSHT, LDRHT, STRHT и TBH. Другие команды доступа к данным всегда вызывают исключение выравнивания. У команд STRD и LDRD указанный адрес должен быть выровнен по словам.

Пример использования единого языка ассемблера приведен в *программе 28.1*.

ПРОГРАММА 28.1. Единый язык ассемблера

- @ Использование кода UAL на Raspberry Pi
- @ Добавление двух значений с помощью короткой процедуры
- @ можно использовать параметр gcc

```
.syntax unified
.global _start
```

```

_start:
    MOV    r0, #10      @ настройка параметра
    MOV    r1, #5       @ настройка параметра
    MOV    r2, #5
    MOV    r3, #20
    BL     doadd        @ вызов процедуры

    MOV    R4, #0xFF00
    MOVT   R4, #0xFFFF
    MLA    R0, R1, R2, R3

stop:
    MOV    R7, #1
    SWI    0

doadd:
    ADD    r0, r0, r1   @ код процедуры в формате UAL
    BX     lr           @ возврат из процедуры

```

Соберите и запустите этот код:

```

as -g -o prog28a.o prog28a.s
ld -o prog28a prog28a.o
./prog28a
echo $?
45

```

Как можно видеть, команда `echo $?` выведет значение: **45**.

Итак, UAL — это общий синтаксис команд A32 и T32, который заменяет собой более ранние версии языков ассемблера A32 и T32. Код, написанный с использованием UAL, можно собрать для A32 или T32 для любого процессора ARM. Изучите некоторые исходные файлы на веб-сайте ARM, и вы заметите, что директива `.syntax unified` используется весьма часто.

29. Обработка исключений

В этой главе мы поговорим об обработке исключений, различных режимах работы ARM, векторах и прерываниях. Это невероятно важная часть работы чипа ARM, позволяющая гибко и универсально настраивать работу выбранной вами операционной системы. Тема обработки исключений весьма сложна, и многие ее подробности выходят за рамки этой книги. Тем не менее это весьма увлекательно, как и вся идея прерываний, т. к. они имеют фундаментальное значение для повседневной работы Raspberry Pi. Поэтому в этой главе представлен краткий обзор, который, безусловно, будет вам полезен, если вы собираетесь глубже заняться программированием на «голом железе» или создать собственную ОС для работы с Raspberry Pi. Именно эти задачи стоит рассмотреть на следующем этапе обучения.

Поскольку операционная система Raspberry Pi OS с точки зрения настройки и ядра дает вам довольно-таки мало свободы, управление памятью ядра не позволяет вам получить доступ к разделам, которые не отображаются в карте памяти процесса. Система не позволит вам читать и писать что угодно и куда угодно в памяти. Именно по этой причине к контактам GPIO и другим аппаратным компонентам Raspberry Pi нельзя получить доступ из стандартной программы машинного кода, работающей под ROS. Этот принцип отличается от, например, RISC OS, где вся система развернута таким образом, чтобы упростить настройку и перенастройку под нужды программиста. Более того, именно для этой цели в ОС RISC есть множество вызовов SYS. Поэтому эту ОС стоит попробовать, если вам нужно вручную управлять прерываниями и событиями.

Прямой доступ к памяти возможен только при работе от пользователя root или написании автономной ОС, которая заменяет Raspberry Pi OS, — по сути, это программирование на «голом железе».

Режимы работы

В *главе 5* мы рассмотрели регистр состояния программы и увидели, как отдельные его биты используются в качестве флагов для обозначения определенных условий. Воспроизведем здесь ту схему снова (рис. 29.1).

31	30	29	28	27...8	7	6	5	4	3	2	1	0
N	Z	C	V		I	F	T	Режим				

Рис. 29.1. Конфигурация регистра состояния

Флаги N, Z, C и V вам уже знакомы. Но те, которые нас интересуют сейчас, хранятся в младшем байте регистра, а именно в битах от 0 до 7.

Биты режима — это биты от 0 до 4 (всего их пять), и их установка определяет, в каком из шести возможных режимов работает ARM. Эти режимы приведены в табл. 29.1. Любой из них можно задать, изменив значение CPSR. За исключением режимов User и Supervisor, при возникновении исключения можно перейти во все эти режимы.

Таблица 29.1. Шесть режимов работы ARM

Режим	Описание
FIQ	Активируется при возникновении прерывания высокого приоритета (быстрого)
IRQ	Активируется при возникновении прерывания низкого приоритета (медленного)
Supervisor и Reset	Активируется при перезагрузке и прерывании SWI
Abort	Обрабатывает ошибки доступа к памяти
Under	Используется для обработки неопределенных команд
User	Стандартный режим без привилегий, в котором работает большинство задач

Режим User — это режим, который используется программами и приложениями по умолчанию. Это обычная среда, в которой мы работаем, и не стоит нам, программистам, ее покидать, если, конечно, нас не манит жажда приключений и мы контролируем работу чипа ARM целиком. Подобные вещи не для новичков, и нужно проявлять осторожность, пересекая эту черту.

Вернемся к рис. 29.1: биты 7 и 6 используются для включения и отключения прерываний IRQ и FIQ. Если бит установлен, соответствующее прерывание отключено. Если бит сброшен, прерывание разрешено. Бит 5 — это бит режима Thumb, о котором мы говорили в главе 27. Для работы прерываний этот бит сброшен и процессор работает в состоянии ARM.

Векторы

Векторы играют огромную роль в работе чипа ARM. Вектор — это ячейка памяти, длина которой составляет ровно одно слово или 32 бита (не путать с векторами в VFP!).

Существуют два типа векторов: аппаратные и программные векторы. Аппаратные векторы жестко привязаны к самому чипу ARM и не меняются, и, как показано в табл. 29.2, они расположены в самом начале карты памяти.

Таблица 29.2. Аппаратные векторы ARM

Адрес	Старшая часть адреса	Вектор
0x00000000	0xFFFF0000	Сброс ARM
0x00000004	0xFFFF0004	Неопределенная команда
0x00000008	0xFFFF0008	Программное прерывание (SWI)
0x0000000C	0xFFFF000C	Отмена (предзахват)
0x00000010	0xFFFF0010	Отмена (данные)
0x00000014	0xFFFF0014	Исключение адреса
0x00000018	0xFFFF0018	IRQ
0x0000001C	0xFFFF001C	FIRQ (или FIQ)

Аппаратные векторы управляют конечным потоком информации и представляют собой набор адресов памяти, которые «известны» микросхеме ARM. Понятие «известны» означает, что они физически «защиты», потому и называются аппаратными векторами. Аппаратные векторы обычно управляют потоком аномальных событий, с которыми сам чип справиться не может. Их часто называют *векторами исключений*, и, как уже отмечалось, они располагаются прямо в начале карты памяти на адресах от 0x00000000 до 0x0000001C.

Однако в ROS таблица векторов может располагаться по более высокому адресу в памяти — в нашем случае, начиная с 0xFFFF0000.

Часто аппаратными векторами приходится управлять для изменения реакции машины на ошибки доступа к памяти. Если происходит попытка получить доступ к несуществующей области памяти, вызывается один из векторов ошибки памяти: от 0x0000000C до 0x00000014. Обычно это приводит к тому, что операционная система сообщает о фатальной ошибке и прекращает выполнение текущей задачи. Иногда это не очень желательно — например, при написании редактора памяти. Было бы лучше просто предупредить пользователя о том, что заданного местоположения не существует, и разрешить продолжение редактирования остальной памяти.

Векторы полезны программистам, поскольку они позволяют программам обращаться к стандартным процедурам без прямого вызова физического адреса, где хранится машинный код процедуры.

На заре компьютерной эры операционные системы были небольшими, и все в них было «жестко закодировано», т. е. адрес использовался в абсолютном выражении. Проблема с использованием абсолютных адресов, а не смещений, заключается в том, что вы всегда привязаны к этому адресу. Если ОС в какой-то момент обновится, этот адрес изменится. Из-за чего любой внешний или сторонний код, ис-

пользующий этот абсолютный адрес, сломается. А при работе с векторами — если обновился код и изменилась его точка выполнения — достаточно лишь изменить адрес в векторе.

Второе преимущество использования векторов состоит в том, что мы, как программисты, также можем изменить адрес в векторе и переопределить его. Таким путем мы можем модифицировать и изменить способ работы Raspberry Pi. В ROS сделать это непросто, но если вы планируете писать код на «голом железе», вам придется самостоятельно управлять векторной таблицей и ее требованиями.

Когда происходит прерывание-исключение, процессор перестает делать то, что делал, и переходит в соответствующее место в векторной таблице. А там уже содержится инструкция, указывающая на начало определенной процедуры. Эти команды обычно принимают одну из трех форм, приведенных в табл. 29.3.

Таблица 29.3. Команды, которые можно использовать в векторе

Команда	Описание
<code>B <адрес></code>	Переход к адресу, заданному в виде смещения от значения в PC
<code>LDR PC, [PC, #смещение]</code>	Загрузка адреса из памяти в PC. Адрес — это 32-битное значение, хранящееся недалеко от векторной таблицы. Эта команда работает чуть медленнее предыдущего метода из-за лишних процессов доступа к памяти. В качестве бонуса — вы можете прыгнуть на любой адрес в карте памяти
<code>LDR PC, [PC, #-0xFFF0]</code>	Загрузка адреса определенной процедуры прерывания из адреса <code>0xFFFFF030</code> в PC
<code>MOV PC #значение</code>	Загрузка константы напрямую в PC. Обычно это однобайтовое значение, прокрученное вправо на четное число битов. Дает доступ ко всей карте памяти, но с пробелами

Например, когда происходит прерывание IRQ, оно в конечном итоге проходит через вектор IRQ. Он имеет ширину 32 бита и достаточно велик, чтобы вместить инструкцию, которая упрощает переход в другое место.

Настройка регистров

У каждого из режимов есть свой набор доступных регистров. Регистры, доступные программисту, зависят от текущего режима работы ЦП.

При работе в режиме User можно использовать полный набор регистров: от R0 до R15. Однако когда CPU переключается в другой режим работы, все меняется. На рис. 29.2 показано расположение регистров в зависимости от режима работы. Все режимы имеют выделенные указатели стека и связанные с ними регистры ссылок. У всех режимов, кроме пользовательского, есть доступный им регистр состояния сохраненной программы (Saved Program Status Register, SPRS), а у режима FIQ доступны регистры R8–R12. В противном случае регистры остаются без изменений.

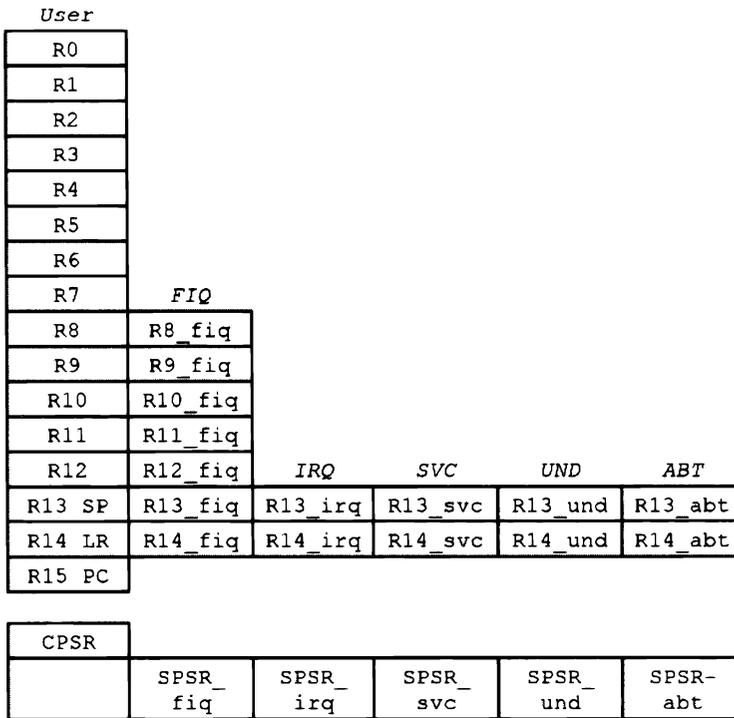


Рис. 29.2. Модель программиста ARM

SPSR используется для хранения копии регистра состояния пользовательского режима при входе в один из других режимов. В режиме *User* SPSR нет, и он не используется. Важно отметить, что CPSR (Current Program Status Register) копируется в SPSR только при возникновении исключения или прерывания, и не изменяется, если вы физически пишете в CPSR для изменения режима.

Идея банковской системы регистров заключается в том, что у каждого режима процессора есть несколько частных регистров, которые он может использовать, не трогая обычные регистры и тем самым гарантируя, что программисту не нужно думать о сохранении содержимого состояния режима *User* при входе в альтернативный режим.

На рис. 29.3 показано, как выглядит младший байт CPSR при вызове одного из режимов. За исключением режима *User*, все режимы являются привилегированными. Когда на чип ARM впервые подается питание, он запускается в режиме *Supervisor*.

Прерывания в ARM очень легко включать и отключать с помощью масок маскирования. Биты 7 и 6 включают или отключают прерывания *IRQ* и *FIQ* соответственно. Если установлен какой-либо бит, соответствующее прерывание отключено и не будет обрабатываться. Когда происходит исключение или прерывание, бит маски прерывания обычно устанавливается микросхемой. Для ряда режимов бит *FIQ* остается неизменным (*unchanged*, *uc*).

	I	F	T	Режим				
	7	6	5	4	3	2	1	0
Abort	1	1	0	1	0	1	1	1
FIQ	1	1	0	1	0	0	0	1
IRQ	1	uc	0	1	0	0	1	0
Supervisor	1	uc	0	1	0	0	1	1
System	1	1	0	1	1	1	1	1
Undefined	1	uc	0	1	1	0	1	1
User	0	0	0	1	0	0	0	0

Рис. 29.3. Битовые настройки для изменения режима CPSR

Обработка исключений

Обработка исключений — это ситуация, требующая временной или иной остановки выполняемого кода. После остановки вызывается сегмент кода, называемый *обработчиком исключений*. Он идентифицирует условие и передает управление соответствующему маршруту для обработки исключения. Когда исключение вызывает изменение режима, происходит следующая последовательность событий:

1. Адрес следующей команды копируется в соответствующий LR.
2. CPSR копируется в SPSR нового режима.
3. Соответствующий режим устанавливается изменением битов в CPSR.
4. Следующая команда берется из таблицы векторов.
5. После обработки исключения управление может быть возвращено коду, который выполнялся, когда возникло исключение.

Управление можно вернуть следующим образом:

1. LR (без смещения) перемещается в PC.
2. SPSR копируется обратно в CPSR, и по умолчанию это автоматически позволяет вернуть предыдущий режим.
3. Флаги запрета прерывания сбрасываются, чтобы снова включить прерывания.

На рис. 29.4 показано, что происходит на уровне регистров при выполнении вызова прерывания SWI. Концепция одинакова для любых исключений привилегированного режима. Значения SP и LR режима доступа переключаются, разрешая обработку исключения, но сохраняя при этом статус прерванной программы. Адрес возврата копируется с PC в пользовательском режиме и сохраняется в LR вызываемого привилегированного режима. В приведенном примере это R14_svc.

Сохраняя состояние трех показанных регистров режима User, можно сохранить выполнение программы после обработки режима SVC, просто вернув значения обратно. Как мы говорили ранее, CPSR сохраняется в SPSR запрашивающего режима



Рис. 29.4. Замена регистров при привилегированном исключении

только при возникновении исключения. Этого не происходит, когда режим меняется путем изменения битов режима.

Команды *MRS* и *MSR*

Существуют две команды, которые можно использовать для прямого управления содержимым CPSR и SPSR, и работают они как для цельных значений, так и на уровне битов.

Команда *MRS* передает содержимое CPSR или SPSR в регистр. Команда *MSR*, наоборот, передает содержимое регистра либо в CPSR, либо в SPSR. Синтаксис команд следующий:

```
MRS (<Синтаксис>) <Операнд1>, <CSPR|SPSR>
MSR (<Синтаксис>) <CSPR|SPSR|Flags>, <Операнд1>
MSR (<Синтаксис>) <CSPR|SPSR|Flags>, #immediate
```

Приведем пару примеров, чтобы стало понятнее. Следующие три строки кода можно использовать для включения режима IRQ:

```
MRS R1, CPSR
BIC R1, R1, #0x80
MSR CPSR_C, R1
```

Сначала регистр CPSR копируется в R1, где затем маскируется с помощью 10000000, чтобы очистить бит b7 — в позиции флага I (см. рис. 5.3). Затем R1 записывается обратно в CPSR. Обратите внимание на суффикс *_C* в конце операнда CPSR. С точки зрения программирования CPSR и SPSR разделены на четыре разных сегмента (рис. 29.5).

Используя правильные суффиксы к командам, мы можем гарантировать, что обновлены будут только правильные биты соответствующего регистра. Суффиксы *F*, *S*, *X* и *C* работают одинаково.

Флаги (F)					Состояние (S)					Расширение (X)					Управление (C)			
[24:31]					[16:25]					[8:15]					[0:7]			
N	Z	C	V	Q											I	F	T	Режим

Рис. 29.5. Сегменты CPSR/SPSR для управления обновлением

Чтобы отключить IRQ, достаточно будет следующего фрагмента кода:

```
MRS R1, CPSR
ORR R1, R1, #x080
MSR CPSR_C, R1
```

Те же команды можно использовать для изменения режима таким образом:

```
MRS R0, CPSR           @ копирование CPSR
BIC R0, R0, #0x1F      @ очистка битов режима
ORR R0, R0, #new_mode  @ выбор нового режима
MSR CPSR, R0           @ запись CPSR обратно
```

В режиме User вы можете читать все биты CPSR, но обновлять можно только состояние флага поля, т. е. CPSR_F.

Что происходит при возникновении прерывания?

Режим запроса прерывания (IRQ) и режим быстрого прерывания (FIQ) вызываются, когда внешнее устройство осуществляет поисковый вызов микросхемы ARM и запрашивает ее внимание. Например, клавиатура генерирует прерывание при каждом нажатии клавиши. При этом возникает сигнал для ЦП о том, что нужно считать матрицу клавиатуры и передать значение ASCII клавиши в буфер клавиатуры. Если значение ASCII — 0x0D (RETURN), запускается интерпретация буфера клавиатуры.

Другой пример: вы сидите за своей собственной клавиатурой и изучаете язык ассемблера Raspberry Pi. В какой-то момент ваш телефон начинает звонить. Вас прервали. Итак, вы прекращаете то, что делали (возможно, сделав небольшую заметку, чтобы позже вспомнить, что к чему), и отвечаете на телефонный звонок (т. е. обрабатываете прерывание). После завершения телефонного разговора вы кладете трубку и возвращаетесь к тому, что делали до прерывания.

На чипе ARM все работает так же. Он получает сигнал прерывания, после чего сохраняет то, что делал до этого, а затем передает управление вызывающей программе, включая соответствующий режим работы. Когда прерывание завершает свою работу (в примере с клавиатурой это будет считывание нажатия клавиши и помещение кода ASCII в буфер клавиатуры), оно передает управление обратно ARM, который восстанавливает всю ранее сохраненную информацию и возвращается в режим User, продолжая с того места, где он остановился.

Итак, прерывание — это функция самого чипа ARM, но то, как оно обрабатывается и что происходит при его возникновении, зависит от обрабатывающего его программного обеспечения — в нашем случае это Raspberry Pi OS.

Без эффективной системы прерываний Raspberry Pi пришлось бы тратить много времени на проверку всех подключенных компонентов, а это отнимает время и ресурсы. Ведь к Raspberry Pi подключено много всего: клавиатура, мышь, USB-порты, дисковые накопители. Все это нужно обслуживать, причем часто.

Считается, что быстрые прерывания (FIQ) имеют наивысший приоритет и должны обслуживаться в первую очередь. Например, при подключении дисков — иначе данные могут быть потеряны. Единственный случай, когда FIQ не обслуживается первым, — это когда уже обрабатывается другое FIQ. Считается, что линия запроса прерывания (IRQ) имеет более низкий приоритет и небольшая задержка не вызовет никаких проблем.

Необходимость в скорости обработки прерывания FIQ обозначена его положением в таблице векторов оборудования. Оно последнее в списке. Все дело в том, что оно начинает выполняться моментально, а все остальные векторы выполняют команды перехода следом за ним. Код FIQ находится в пространстве после вектора по адресу $0 \times 1C$. Задача кодирования прерывания заключается в том, чтобы определить, какое устройство вызвало прерывание, и обработать его правильно и как можно быстрее.

Решения о прерываниях

При работе с прерываниями вам необходимо принять решение о том, что делать с другими прерываниями. Что произойдет, если новое прерывание возникнет в момент, пока вы обрабатываете имеющееся прерывание? Самый простой способ — вызвать так называемую *схему обработки невложенных прерываний*. В этом случае все прерывания отключаются до тех пор, пока управление не будет возвращено прерванной задаче. Обратной стороной такого подхода является то, что одновременно может обслуживаться только одно прерывание, и если происходит последовательность прерываний, есть шанс потерять некоторые запросы. Последствия могут оказаться негативными.

Схема вложенных прерываний позволяет обрабатывать более одного прерывания одновременно, и в этом случае вы должны повторно включить прерывания, прежде чем полностью обслужить текущее прерывание. Это чуть более сложный механизм, но он решает возможные проблемы с задержкой прерывания, которая представляет собой интервал времени от подъема внешнего сигнала прерывания до первой выборки команды этого прерывания.

При обработке FIQ прерывания IRQ отключены, т. к. FIQ имеет высший приоритет.

Для переключения контекста между режимами и сохранения информации всегда важна реализация стека для обработки прерываний (стека прерываний). Если одновременно происходит несколько прерываний, необходимо где-то хранить информацию.

Возврат из прерываний

После выполнения процедуры обработки прерывания операционная система должна вернуться к основной программе, которая была прервана FIQ или IRQ. Это делается с помощью следующей команды:

Она восстанавливает счетчик программ, поэтому прерванную программу можно возобновить точно с того момента, на котором она прервалась. Вычитание значения 4 требуется для компенсации эффектов конвейерной обработки. При условии, что процедура обработки прерывания не повредила какие-либо общие регистры или рабочее пространство, программа продолжит выполнение, как если бы прерывание вовсе не возникло.

Пишем процедуры прерывания

Обычно создавать собственные процедуры обслуживания обработчиков прерываний не требуется, потому что у ОС есть для этого своя система. Если вы собираетесь писать процедуры обработки прерываний, нужно соблюдать несколько правил, которые спасут вас от катастроф:

- ◆ не включайте прерывания в процедуру обработки. Если так сделать, второе прерывание IRQ/FIQ может прервать работу процессора до того, как он обработает уже имеющееся прерывание. Иногда это допустимо, но это весьма рискованный путь. Будьте очень внимательны!
- ◆ процедура прерывания должна быть написана как можно более экономично. Обработка прерывания должна выполняться как можно быстрее. Если она будет продолжаться слишком долго, обычная фоновая работа Raspberry Pi попросту встанет. Клавиатура заблокируется, программные часы будут терять время и т. д.;
- ◆ все регистры процессора нужно сохранить. При выходе из процедуры обработки прерывания они должны содержать те же значения, что и при входе в прерывание. Это очень важно для правильного возобновления прерванной задачи;
- ◆ процедура обработки прерывания не должна вызывать процедуры из ОС. Если процедура будет прервана IRQ/FIQ, она выполнится только наполовину. При повторном входе в процедуру прерывания рабочее пространство может быть нарушено, что потенциально способно привести к повреждению процедуры при возобновлении.

30. System-on-Chip

Квадратная микросхема в центре платы Raspberry Pi (рис. 30.1) — это то самое устройство, которому посвящена эта книга. Все, что мы делали, читая ее и прорабатывая ее примеры, происходило в этой микросхеме. Фактически этот кусок кремния — это гораздо больше, чем просто чип ARM. Это устройство System-on-Chip.

В целом SoC — это микросхема, в которой есть все компоненты, необходимые для управления компьютером. Состав SoC в разных версиях Raspberry Pi менялся, но основными его компонентами являются ARM-совместимый ЦП и встроенный графический процессор GPU (VideoCore IV). Частоты процессора варьируются от 700 МГц (Raspberry Pi 1) до 1,2 ГГц у Raspberry Pi 3. Объем ОЗУ составляет от 256 МБ до 1 Гбайт.

Созданный специально для Raspberry Pi 4, серебристый элемент BCM2711B0 (см. рис. 30.1) представляет собой 64-битную четырехъядерную систему на кри-

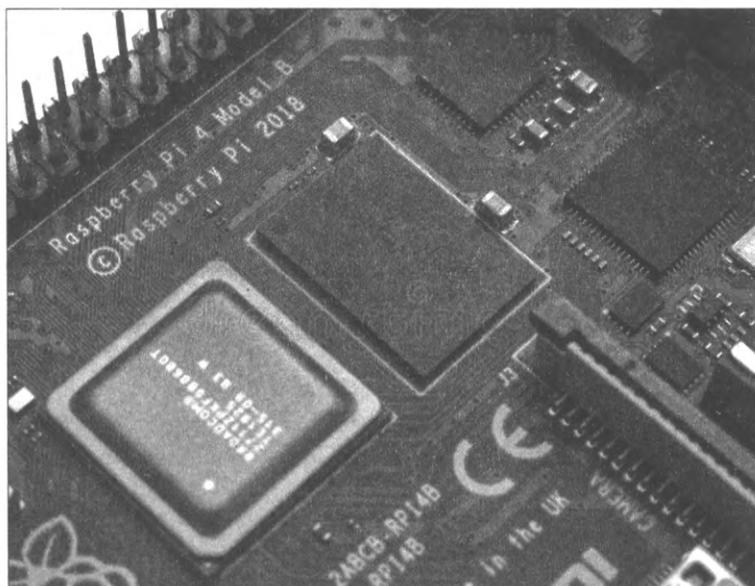


Рис. 30.1. Raspberry Pi с SoC

сталле Cortex-A72 (ARMv8-A), работающую на частоте 1,5 ГГц. В его состав входит Broadcom VideoCore VI — первое обновление графического процессора для Raspberry Pi, а также различные сопроцессоры, о которых мы говорили в этой книге.

Большой черный чип рядом с ним — это память, доступная в вариантах 1, 2, 4 или 8 Гбайт.

Технология SoC становится все более популярной, а упаковка множества компонентов в единую интегральную схему (ИС) означает, что устройства, в которые они встроены, становятся все меньше. Если вы внимательно посмотрите на плату Raspberry Pi, то увидите, что большая часть места на ней занята компонентами, которые позволяют нам подключаться к основной схеме. Непосредственно выполняющая вычисления часть, вероятно, занимает всего около 10% печатной платы! И с каждой новой версией на плату Raspberry Pi помещается все больше и больше.

Можете не сомневаться, технология SoC, используемая в Raspberry Pi, стремительно развивается, и скоро она станет преобладающей во множестве сфер в области технологий, поскольку физический размер устройства SoC становится все меньше. Благодаря огромным цифровым возможностям этой технологии скоро могут появиться аналогичные кремниевые устройства, восстанавливающие зрение слепым и слух глухим.

Еще одно важное преимущество конструкции SoC заключается в том, что она не требует большого количества энергии и весьма эффективна из-за небольших расстояний, которые проходят в ней сигналы. Помните, сколько тепла выделяют стандартные ПК и как много места отводится на охлаждение?

У технологии SoC есть и обратная сторона, и причина, по которой эти устройства пока не доминируют на потребительском и общем рынке. Речь идет об отсутствии возможности обновления. Обновление Raspberry Pi невозможно, потому что на плате все собрано в одном месте, из-за чего сложно добавить новую память или обновить основной процессор. Исчерпав все возможности устройства, вы фактически выбрасываете его. Поэтому ваша Raspberry Pi, несомненно, устареет, и вам придется просто заменить ее на более новую модель. Получается, Raspberry Pi — это одноразовый компьютер?

Микросхема и набор команд ARM

В основе SoC платы Raspberry Pi лежит ядро ARM. SoC не всегда работают исключительно на ARM, но его часто предпочитают из-за использования архитектуры RISC, которая обеспечивает хорошее энергосбережение. Сам чип ARM существует уже более 30 лет и постоянно совершенствуется. ARM11, используемый в оригинальной Raspberry Pi, вышел в 2003 году и построен на архитектуре ARMv6. Чип ARMv7, установленный в Raspberry Pi 2, на несколько лет старше и является одним из самых быстрых и наиболее энергоэффективных доступных чипов. Процессор ARM v8 в Raspberry Pi 3 — это тоже шаг в том же направлении, и его скорость те-

перь позволяет запускать популярные операционные системы, такие как Windows 10 и Ubuntu. У этого чипа четырехъядерный процессор, что означает, что он может выполнять несколько задач одновременно, а на исходном одноядерном чипе это было невозможно.

Мы видели два набора команд, которые есть на Raspberry Pi, а именно: ARM и Thumb. Когда-то существовал еще и третий набор команд — Jazelle. Он был в первую очередь предназначен для помощи в разработке программного обеспечения для мобильных телефонов, и с момента выпуска архитектуры ARMv7 больше не использовался.

Сопроцессоры

Устройство ARM позволяет подключать к нему дополнительное оборудование.

ИНТЕРЕСНЫЙ ФАКТ

Такую уникальную конструкцию представила Acorn еще в 1983 году с запуском интерфейса Second Processor Tube, который позволял присоединять к плате BBC Micro дополнительные процессоры. Это давало возможность запускать на ней другие операционные системы, зависящие от процессора, такие как CP/M. Второй процессор ARM был одним из последних, выпущенных для BBC Micro!

Поскольку присоединяемые устройства дополняют собой ARM, их называют *сопроцессорами*. К ARM допускается подключить до 16 устройств, пронумерованных от 0 до 15. Для связи с ними можно использовать команды MCR и MRC, и, кроме того, у многих сопроцессоров есть свои собственные наборы команд, которые стоит изучить.

Мы уже подробно рассмотрели один сопроцессор: сопроцессор Vector Floating-Point, который реализует управление действительными числами на микросхеме ARM. Фактически VFP занимает в системе два слота: CP10 и CP11. Слоты CP14 и CP15 также зарезервированы для использования в системе, но все остальные можно задействовать свободно.

Конвейер

О конвейере речь шла в *главе 13*. Мы рассмотрели трехэтапный процесс его работы и намекнули, что у процессора ARMv6 работа конвейера разделена на 8 этапов. Они представлены в табл. 30.1. Во время работы, в зависимости от выполняемой команды, конвейер идет одним из трех различных способов на этапы 5, 6 и 7, что позволяет повысить эффективность обработки команд микросхемы.

Существует три рабочих блока, которые подключаются на этапах 5, 6 и 7 в зависимости от выполняемой операции. Например, если выполняется команда умножения, инициируются соответствующие ей этапы. Это позволяет ARM выдавать примерно по одной инструкции за цикл.

На этапе выборки может содержаться до четырех команд, а на этапах выполнения, запоминания и записи могут быть прогнозируемая ветвь, команда ALU или умножения, команда загрузки/сохранения и инструкция сопроцессора при параллельном выполнении.

Длину конвейера можно было бы увеличить, но это непрактично, т. к. приведет к увеличению энергопотребления и тепловыделения. Это, в свою очередь, пагубно повлияет на компактные системы вроде Raspberry Pi, т. к. им начнут требоваться более крупные блоки питания и больше пространства между компонентами, а может и вовсе понадобится система охлаждения! С энергоэффективной платой Raspberry Pi 2, у которой четыре доступных для использования ядра, гораздо меньше проблем.

Таблица 30.1. Восьмиступенчатый конвейер ARMv11
(включает в себя вспомогательные конвейеры, которые можно объединять для достижения максимальной эффективности)

Этап	Имя	Описание
1	Fe1	На этом этапе выборки команды в память отправляется адрес, а в ответ возвращается команда
2	Fe2	Второй этап выборки команды. В этот момент ARM старается предсказать направление любой ветви
3	De	Декодирование команды
4	Iss	Считывание регистров и выдача действия команды
5	Sh	Выполнение операций сдвига, если таковые требуются
6	ALU	Выполнение целочисленных операций
7	Sat	Заполнение целочисленных результатов
5	MAC1	Первый этап накопительного умножения на конвейере
6	MAC2	Второй этап накопительного умножения на конвейере
7	MAC3	Третий этап накопительного умножения на конвейере
5	ADD	Этап генерации адреса
6	DC1	Первый этап доступа к кэшам данных
7	DC2	Второй этап доступа к кэшам данных
8	WBi	Запись данных от умножения или основных конвейеров

Память и кэши

В зависимости от вашей модели Raspberry Pi, она будет обладать определенным объемом памяти. Эта память является частью SoC и расположена в верхней части чипа ARM по типу «пакет на пакете».

У процессора ARM также есть особенная область удивительно быстрой памяти, которая называется *кэш-памятью*. Он может использовать ее для своих целей. Кэш предназначен для хранения команд и данных, и его части называются соответственно *Icache* и *Dcache*. Эти кэши управляются сопроцессором управления системой (CP15), как и вся память, что позволяет ARM продолжать выполнять команды обработки данных. Также на плате есть второй кэш, который называется L2. Но он используется исключительно в VideoCore.

GPU

Графический процессор — еще один важный компонент конфигурации SoC. Это устройство Broadcom VideoCore IV, которое позволяет создавать графику разрешения 1080 с помощью открытого программного обеспечения и аппаратного ускорения вычислений. Информации об этой системе мало из-за ограничений, связанных с авторскими правами, но кое-кому удалось все же выложить некоторую информацию на GitHub. Там вы можете найти файлы примеров на C и на других языках, таких как Python и Java.

Обзор ARMv8

Архитектура ARMv8 впервые стала доступна пользователям Raspberry Pi, начиная с версии Raspberry Pi 2B v1.2 с чипом Broadcom BCM2837. Базовая архитектура BCM2837 идентична BCM2836. Единственное существенное различие заключается в том, что четырехъядерный кластер ARMv7 был заменен на четырехъядерный кластер ARM Cortex A53 (ARMv8). Затем то же самое было сделано для различных вариантов Raspberry Pi 4, но использовался четырехъядерный процессор на основе Cortex-A72. То же коснулось VFPv4 и Neon. Кроме того, и в значительной степени это касается Raspberry Pi 4, — редизайн ее плат дал возможность использовать версию с объемом памяти 8 Гбайт, что позволило улучшить работу с Linux, а также перенести ОС Raspberry Pi на 64-разрядную среду.

Изначальная концепция ARMv8 заключалась в том, чтобы дать ARM «чистый лист» для разработки и кодирования совершенно нового набора команд. Суть была в получении 64-битного набора команд, мнемоника и принципы работы которых были бы вам знакомы. С A64 начался новый этап. Будем надеяться, что новый набор команд будет одинаков для разных архитектур, что обеспечит стабильность разработки и непрерывность реализации. И, как мы уже говорили, пользовательские команды дают программистам огромный простор возможностей при работе с будущими версиями.

Из-за огромного количества 32-разрядных ARM на рынке именно эта версия является предпочтительным процессором для нынешних дизайнеров и разработчиков. Не в последнюю очередь это связано с тем, что программисты имеют большой опыт работы с 32-битным кодом и множество приложений было унаследовано от AArch32. Существуют и чисто 64-битные процессоры ARM (например, Cortex

A34), но двойная совместимость все равно остается высоким приоритетом. По этой причине значительная часть этой книги была, насколько это возможно, посвящена вопросам совместимости со старыми архитектурами и оптимизации наборов команд ARM и Thumb. Концепция Unified Assembler Language была разработана с целью обеспечения обратной совместимости, а также для улучшения работы в FPU.

В начале этой книги (см. табл. 1.1) мы видели, что новейшие чипы Raspberry Pi ARM могут работать как в 32-битном (AArch32), так и в 64-битном (AArch64) режимах. В будущем должна появиться возможность запускать приложения AArch32 и Arch64 на ОС AArch64. Однако ОС AArch32 могла запускать только приложения AArch32. Это означает, что под одним и тем же 64-битным ядром можно использовать как 32-битные, так и 64-битные приложения. Как это получится, еще неизвестно.

64-разрядная ОС Raspberry Pi

В мае 2020 года была выпущена бета-версия Raspberry Pi OS, которая представляла собой обновленную версию стандартной 32-разрядной версии. Ее можно установить на Raspberry Pi 3 и Raspberry Pi 4. На момент подготовки этой книги ОС доступна в бета-версии, поэтому в ней могут быть те же ошибки, что и в начальных версиях 32-битной Raspbian. При установке на выделенную SD-карту ОС работала нормально, а в случае каких-либо неожиданностей вы всегда сможете обратиться к активно развивающемуся сообществу ее пользователей.

А что в итоге?

Мы привели краткий обзор технологии, которая управляет вашей Raspberry Pi. В основе технологии лежит чип ARM, который мы в этой книге научились программировать. Это далеко не все возможности, но и их достаточно для понимания того, что на Raspberry Pi появляются захватывающие новые технологии, вместе с которыми вы можете двигаться вперед. У Raspberry Pi с четырехъядерными чипами есть еще больше возможностей для исследователей.

Принцип Архимеда

Систему Acorn Archimedes (рис. 30.2) разработала компания Acorn Computers Ltd в Кембридже в Англии. Система была основана на процессорах архитектуры ARM собственной разработки Acorn и на собственной операционной системе RISC OS. Первая модель системы появилась в 1987 году, после чего системы семейства Archimedes продавались до середины 1990-х годов.

Использование ARM2 обеспечивает лучшую производительность, чем Intel 286, несмотря на то, что в чипе ARM2 на 245 000 транзисторов меньше, чем у большого чипа Intel (фактически на чипе ARM2 было всего 25 000 транзисторов!). Такая «не-

хватка» транзисторов многое говорит об относительной простоте ARM. В конструкции чипа использовался 32-битный ЦП (с 26-битной адресацией), работающий на частоте 8 МГц, что было гораздо круче 8-битных домашних компьютеров того времени.



Рис. 30.2. Acorn Archimedes с процессором ARM2

ПРИЛОЖЕНИЕ

1. Коды символов ASCII

Двоичное значение	Десятичное значение	Шестнадцатеричное значение	ASC	Двоичное значение	Десятичное значение	Шестнадцатеричное значение	ASC
00100000	32	20	Пробел	00110111	55	37	7
00100001	33	21	!	00111000	56	38	8
00100010	34	22	"	00111001	57	39	9
00100011	35	23	#	00111010	58	3A	:
00100100	36	24	\$	00111011	59	3B	;
00100101	37	25	%	00111100	60	3C	<
00100110	38	26	&	00111101	61	3D	=
00100111	39	27	'	00111110	62	3E	>
00101000	40	28	(00111111	63	3F	?
00101001	41	29)	01000000	64	40	@
00101010	42	2A	*	01000001	65	41	A
00101011	43	2B	+	01000010	66	42	B
00101100	44	2C	,	01000011	67	43	C
00101101	45	2D	.	01000100	68	44	D
00101110	46	2E	.	01000101	69	45	E
00101111	47	2F	/	01000110	70	46	F
00110000	48	30	0	01000111	71	47	G
00110001	49	31	1	01001000	72	48	H
00110010	50	32	2	01001001	73	49	I
00110011	51	33	3	01001010	74	4A	J
00110100	52	34	4	01001011	75	4B	K
00110101	53	35	5	01001100	76	4C	L
00110110	54	36	6	01001101	77	4D	M

Двоичное значение	Десятичное значение	Шестнадцатеричное значение	ASC	Двоичное значение	Десятичное значение	Шестнадцатеричное значение	ASC
01001110	78	4E	N	01100111	103	67	g
01001111	79	4F	O	01101000	104	68	h
01010000	80	50	P	01101001	105	i	i
01010001	81	51	Q	01101010	106	6A	j
01010010	82	52	p	01101011	107	6B	k
01010011	83	53	S	01101100	108	6C	l
01010100	84	54	T	01101101	109	6D	m
01010101	85	55	U	01101110	110	6E	n
01010110	86	56	V	01101111	111	6F	o
01010111	87	57	W	01110000	112	70	p
01011000	88	58	X	01110001	113	71	q
01011001	89	59	Y	01110010	114	72	r
01011010	90	5A	Z	01110011	115	73	s
01011011	91	5B	[01110100	116	74	t
01011100	92	5C	\	01110101	117	75	u
01011101	93	5D]	01110110	118	76	v
01011110	94	5E	^	01110111	119	77	w
01011111	95	5F	_	01111000	120	78	x
01100000	96	60	`	01111001	121	79	y
01100001	97	61	a	01111010	122	7A	z
01100010	98	62	b	01111011	123	7B	{
01100011	99	63	c	01111100	124	7C	
01100100	100	64	d	01111101	125	7D	}
01100101	101	65	e	01111110	126	7E	~
01100110	102	66	f				

ПРИЛОЖЕНИЕ

2. Набор команд ARM

В этом приложении приведена сводка набора команд ARM. Команды представлены в порядке типа операции. Этот список ни в коем случае не является окончательным или исчерпывающим и играет лишь роль справки, в которую включены многие рассмотренные нами команды. Помните, что в разных архитектурах эти команды могут различаться, при этом некоторые из них могут быть вообще убраны или адаптированы.

Руководства на сайте www.arm.com позволят вам побольше узнать о новых процессорах ARM и об особенностях архитектуры команд для каждого из них.

В приложении применены следующие сокращения и их значения:

Rn	Регистр назначения, где n — номер регистра.
Rm	Реестр оператора, где m — номер регистра.
Rdn	Регистр назначения: Rd или Rn, в зависимости от того, указан ли опциональный параметр {Rd}.
N	Флаг отрицания. Равен 1, если результат отрицательный.
Z	Флаг нуля. Равен 1, если результат команды равен 0.
C	Флаг переноса. Равен 1, если команда приводит к переносу.
V	Флаг переполнения. Равен 1, если команда приводит к переполнению.
#	Константа.
{ }	Необязательные параметры.
{ }	Альтернативные необязательные параметры .
{ () }	Альтернативные обязательные параметры.

Суффиксы загрузки/сохранения:

B	Расширенный байт.
H	Расширенное полуслово: 16 битов.

SB	Расширенный байт со знаком.
SH	Расширенное полуслово со знаком: 16 битов.
S	Необязательный суффикс S указывает на обновление регистра состояния.

Отсутствие суффикса Load/Store означает четырехбайтовую передачу.

Команды сравнения и проверки

Формат	Описание
CMP Rn, Rm	Сравнить регистры, обновить состояние.
CMP Rn, #K	Сравнить с 8-битной константой K, обновить состояние.
TST Rn, Rm	Проверить регистры, обновить состояние.
TST Rn, #K	Проверить с 8-битным K, обновить состояние.
TEQ Rn, Rm	Проверить равенство, обновить состояние.
TEQ Rn, #K	Проверить равенство 8-битного K, обновить состояние.

Команды ветвления

Формат	Описание
B <i>метка</i>	Безусловный переход к метке.
BEQ <i>метка</i>	Переход при равенстве.
BNE <i>метка</i>	Переход при неравенстве.
BLT <i>метка</i>	Переход, если меньше (со знаком).
BLE <i>метка</i>	Переход, если меньше или равно (со знаком).
BGT <i>метка</i>	Переход, если больше (со знаком).
BGE <i>метка</i>	Переход, если больше или равно (со знаком).
BLO <i>метка</i>	Переход, если меньше (без знака).
BLS <i>метка</i>	Переход, если меньше или равно (без знака).
BHI <i>метка</i>	Переход, если больше (без знака).
BHS <i>метка</i>	Переход, если больше или равно (без знака).
BL <i>метка</i>	Обновить регистр ссылки и переход.
BX Rn	Переход по адресу в регистре Rn.

Арифметические команды

Формат	Описание
ADD{S} {Rd, } Rn, Rm	Сложить два регистра.
ADD{S} {Rd, } Rn, #K	Сложить регистр с 8-битной константой K.
ADC{S} {Rd, } Rn, Rm	Сложить два регистра и флаг переноса (C).
ADC{S} {Rd, } Rn, #K	Сложить регистр с 8-битной константой и флагом переноса (C).
SUB{S} {Rd, } Rn, Rm	Вычесть два регистра.
SDIV	Деление со знаком.
SUB{S} {Rd, } Rn, #K	Вычесть 8-битную константу K.
SBC{S} {Rd, } Rn, Rm	Вычесть два регистра с флагом переноса.
SBC{S} {Rd, } Rn, #K	Вычесть 8-битную константу с флагом переноса.
MUL{Rd, } Rn, Rm	Умножение регистров (со знаком или без знака).
MLA	Умножение с накоплением.
MLS	Умножение и вычитание.
NOP	Бездействие.
UMULL RdL, RdH, Rn, Rm	Беззнаковое умножение (64-битный результат).
SMULL RdL, RdH, Rn, Rm	Умножение со знаком (64-битный результат).

Логические команды

Формат	Описание
AND{S} {Rd, } Rn, (Rm #K)	Логическое умножение регистра с 8-битной константой.
BIC{S} {Rd, } Rn AND NOT Rm	Сброс бита.
ORR{S} {Rd, } Rn, (Rm #K)	Логическое сложение регистра с 8-битной константой.
EOR{S} {Rd, } Rn, (Rm #K)	Исключающее ИЛИ с регистром и 8-битной константой K.
NEG{S} {Rd, } Rn	Инвертировать регистр.
LSL{S} {Rd, } Rn, (Rm #K)	Логический сдвиг влево.
LSR{S} {Rd, } Rn, (Rm #K)	Логический сдвиг вправо.
ASR{S} {Rd, } Rn, (Rm #K)	Арифметический сдвиг вправо.
ROR{S} {Rd, } Rn, (Rm #K)	Вращение вправо.

Команды перемещения данных

Формат	Описание
MOV{S} Rd, Rm	Копировать значение из Rm в Rd.
MOVW Rd, #K	Копировать 16-битную константу K в Rd (Zero Ext).
MOVT Rd, #K	Копировать 16-битную константу K в Rd[31:16].
PUSH Rn	Поместить Rn на вершину стопки.
POP Rn	Поместить верхнее значение из стека в Rn.
SVC n	Вызов супервизора (программное прерывание SWI).
STR{B H} Rt, [Rn, #+/-K]	Сохранить: Смещение адреса в виде константы (8-битное K).
STR{B H} Rt, [Rn, #+/-K]!	Сохранить: Преиндексирование, смещение на константу (8-битный K).
STR{B H} Rt, [Rn], #+/-K	Сохранить: Постиндексирование, смещение на константу [Rn].
STR{B H} Rt, [Rn, Rm {, LSL #s}]	Сохранить: Смещение с адресом в регистре.
LDR{B H SB SH} Rt, [Rn, #+/-K]	Загрузить: Смещение адреса в виде константы (8-битный K).
LDR{B H SB SH} Rt, [Rn, #+/-K]!	Загрузить: Преиндексирование, смещение на константу (8-битный K).
LDR{B H SB SH} Rt, [Rn], #+/-K	Загрузить: Постиндексирование, смещение на константу (8-битный K).
LDR{B H SB SH} Rt, [Rn, Rm {, LSL #s}]	Загрузить: Смещение с адресом в регистре.

ПРИЛОЖЕНИЕ

3. Системные вызовы ROS

В этом приложении приведены первые 193 системных вызова ROS с кратким описанием (табл. ПЗ.1). Официальных источников или документации системных вызовов Linux нет, поэтому документация ограничена созданными пользователями источниками.

Таблица ПЗ.1. Системные вызовы ROS

Номер	Действие	Функция
0	restart-syscall	Перезапустить системный вызов
1	exit	Завершить текущий процесс
2	fork	Создать дочерний процесс
3	read	Чтение из файлового дескриптора
4	write	Запись в файловый дескриптор
5	open	Открыть/создать файл или устройство
6	close	Закрывает файловый дескриптор
7	waitpid	Дождаться завершения процесса
8	creat	Создать дочерний процесс
9	link	Присвоить файлу новое имя
10	unlink	Удалить имя и файл, к которому оно относится
11	execve	Выполнить программу
12	chdir	Сменить рабочий каталог
13	time	Получить время в секундах
14	mknod	Создать каталог или специальный/обычный файл
15	chmod	Изменить права доступа к файлу
16	lchown	Изменить владельца файла
18	oldstat	(не используется)
19	lseek	Перемещение смещения файла чтения/записи

Таблица П3.1 (продолжение)

Номер	Действие	Функция
20	getpid	Получить идентификатор процесса
21	mount	Смонтировать файловые системы
22	umount	Размонтировать файловые системы
23	setuid	Установить личность пользователя
24	getuid	Получить идентификационные данные пользователя
25	stime	Установить время
26	ptrace	Трассировка процесса
27	alarm	Установить таймер сигнала тревоги
28	oldfstat	(не используется)
29	pause	Ожидание сигнала
30	utime	Изменить время доступа и/или модификации
33	access	Проверить права пользователя для файла
34	nice	Изменить приоритет процесса
35	ftime	Дата и время
36	sync	Зафиксировать буферный кэш на диске
37	kill	Отправить сигнал процессу
38	rename	Изменить имя или расположение файла
39	mkdir	Создать каталог
40	rmdir	Удалить пустой каталог
41	dup	Дублировать файловый дескриптор
42	pipe	Создать конвейер
43	times	Получить время обработки
45	brk	Изменить размер сегмента данных
46	setgid	Установить id группы
47	getgid	Получить id группы
48	signal	Обработка сигналов ANSI C
49	geteuid	Получить id пользователя
50	getegid	Получить id группы
51	acct *	Включение или отключение учета процессов
52	umount2	Монтировать и размонтировать файловые системы
54	ioctl	Управление устройством
55	fcntl	Управление файловым дескриптором
57	setpgid	Установить/получить группу процессов

Таблица П3.1 (продолжение)

Номер	Действие	Функция
58	ulimit :	Getrlimit (2)
60	umask	Задать маску создания файла
61	chroot	Изменить корневой каталог
62	ustat	Получить статистику файловой системы
63	dup2	Заменить вторую копию первого дескриптора файла
64	getppid	Получить id процесса
65	getpgrp	Установить/получить группу процессов
66	setsid	Создать сеанс, установить id группы процессов
67	sigaction	Проверить и изменить статус сигнала
70	setreuid	Установить реальный и/или действующий идентификатор пользователя
71	setregid	Установить реальный и/или действующий идентификатор группы
72	sigsuspend	Функции обработки сигналов POSIX
73	sigpending	Функции обработки сигналов POSIX
74	sethostname	Установить имя хоста
75	setrlimit	Получить/задать ограничения использования ресурсов
76	getrlimit	Получить/задать ограничения использования ресурсов
77	getrusage	Получить/задать ограничения использования ресурсов
78	gettimeofday	Получить время
79	settimeofday	Получить время
80	getgroups	Получить список id дополнительных групп
81	setgroups	Установить список id идентификаторов групп
82	select+K4:	Мультиплексирование синхронного ввода/вывода
83	symlink	Задать новое имя файла
84	oldlstat	(не используется)
85	readlink	Прочитать значение символической ссылки
86	uselib	Выбрать общую библиотеку
87	swapon	Начать подкачку к файлу/устройству
88	reboot	Перезагрузить или включить/выключить <Ctrl>+<Alt>+
89	readdir	Прочитать запись в каталоге
90	mmap	Сопоставить файлы или устройства с памятью (несуществующие)
91	munmap	Отменить отображение файлов или устройств в память

Таблица П3.1 (продолжение)

Номер	Действие	Функция
92	truncate	Обрезать файл до указанной длины
93	ftruncate	Обрезать файл до указанной длины
94	fchmod	Изменить права доступа к файлу
95	fchown	Изменить владельца файла
96	getpriority	Получить приоритет планирования программы
97	setpriority	Установить приоритет планирования программы
98	profil	Профиль времени выполнения
99	statfs	Получить статистику файловой системы
100	fstatfs	Получить статистику файловой системы
101	ioperm	Установить разрешения порта на ввод/вывод
102	socketcall	Системные вызовы сокетов
103	syslog	Считать или очистить сообщения ядра; лог-уровень
104	setitimer	Установить таймер eval
105	getitimer	Получить значение таймера eval
106	stat	Получить состояние файла
107	lstat	Получить состояние файла
108	fstat	Получить состояние файла
110	iopl	Изменить уровень привилегий ввода-вывода
111	vhangup	Виртуально текущий процесс
112	idle	Сделать процесс 0 неактивным
113	vm86old	Перейти в виртуальный режим 8086
114	wait4	Дождаться завершения процесса BSD
115	swapoff	Прекратить подкачку к файлу/устройству
116	sysinfo	Информация об общей статистике системы
117	ipc	Системные вызовы System V IPC
118	fsync	Синхронизировать полное внутреннее состояние с диском
119	sigreturn	Возврат из сигнала, очистка стека
120	clone	Создать дочерний процесс
121	setdomainname	Получить/установить доменное имя
122	uname	Получить имя и информацию о текущем ядре
123	modify_ldt	Получить или установить ldt
124	adjtimex	Настроить часы ядра
125	mprotect	Управление доступами к памяти

Таблица П3.1 (продолжение)

Номер	Действие	Функция
126	sigprocmask	Функции обработки сигналов POSIX
127	create_module	Создать запись модуля
128	init_module	Инициировать запись загружаемого модуля
129	delete_module	Удалить запись модуля
130	get_kernel_syms	Получить файлы ядра и модулей
131	quotactl	Управление дисковыми квотами
132	getpgid	Установить/получить группу процессов
133	fchdir	Сменить рабочий каталог
134	bdflush	Запустить/очистить/настроить буфер
135	sysfs	Получить информацию о типе файловой системы
136	personality	Установить домен выполнения процесса
138	setfsuid	Установить идентификатор пользователя для проверки файловой системы
139	setfsgid	Установить идентификатор группы для проверки файловой системы
140	llseek	Запись/чтение смещения файла
141	getdents	Получить записи каталога
142	newselect+K4:	Синхронное мультиплексирование ввода/вывода
143	flock	Применить/снять блокировку рекомендательного файла
144	msync	Синхронизировать файл с картой памяти
145	readv	Считать вектор
146	writev	записать вектор
147	getsid	Получить идентификатор сеанса
148	fdatasync	Синхронизировать данные ядра с диском
150	mlock	Отключить подкачку для некоторых частей памяти
151	munlock	Повторно включить подкачку для некоторых частей памяти
152	mlockall	Отключить пейджинг для вызывающего процесса
153	munlockall	Повторное включение пейджинга для вызывающего процесса
154	sched_setparam	Установить параметры планирования
155	sched_getparam	Получить параметры планирования
156	sched_setscheduler	Установить алгоритм/параметр планирования
157	sched_getscheduler	Получить алгоритм/параметр планирования
158	sched_yield	Обработка yield
159	sched_get_priority_max	Получить состояние статического приоритета

Таблица ПЗ.1 (окончание)

Номер	Действие	Функция
160	sched_get_priority_min	Получить состояние статического приоритета
161	sched_rr_get_erval	Получить SCHED_RR erval
162	nanosleep	Приостановить выполнение на указанное время
163	mremap	Переназначить адрес виртуальной памяти
164	setresuid	Задать реальный, действующий и сохраненный идентификатор пользователя
165	getresuid	Получить реальный/действующий/сохраненный идентификатор пользователя
166	vm86 s	Перейти в виртуальный режим 8086
167	query_module	Для бит, относящихся к модулям
168	poll	Ожидание события в файловом дескрипторе
169	nfsservctl	Демон ядра nfs
170	setresgid	Задать реальный, эффективный, сохраненный идентификатор группы
171	getresgid	Получить реальный/эффективный/сохраненный идентификатор группы
172	prctl	Операции над процессом
173	rt_sigreturn	Очистить стек после возврата из сигнала
174	rt_sigaction	Проверить/изменить сигнальное действие
175	rt_sigprocmask	Проверить/изменить сигналы блокировки
176	rt_sigpending :	Проверить ожидающие сигналы
177	rt_sigtimedwait	Ожидание сигналов в очереди
178	rt_sigqueueinfo	Поставить в очередь сигнал и данные
179	rt_sigsuspend	Ожидание сигнала
180	pread	Чтение из файлового дескриптора по заданному смещению
181	pwrite	Запись в файловый дескриптор по заданному смещению
182	chown	Изменить владельца файла
183	getcwd	Получить текущий рабочий каталог
184	capget	Получить возможности процесса
185	capset	Установить возможности процесса
186	sigaltstack	Установить/получить контекст стека сигналов
187	sendfile	Передача данных между файловыми дескрипторами
190	vfork	Создать дочерний процесс и заблокировать родительский
191	getrlimit	Получить лимит ресурсов
192	mmap2	Сопоставить файл или устройство с памятью

ПРИЛОЖЕНИЕ

4. Описание электронного архива

Электронный архив с файлами приведенных в книге программ можно загрузить с FTP-сервера издательства «БХВ» по ссылке: <ftp://ftp.bhv.ru/9785977568012.zip> или со страницы книги на сайте <https://bhv.ru/>.

Файлы программ в архиве размещены в папках с номерами соответствующих глав книги. Так, программы, описанные, например, в *главе 6*, содержатся в папке CH6.

Обратите внимание, что автор использует для нумерации программ в книге и в электронном архиве цифробуквенную запись: 3а, 6а, 6b и т. д., мы же нумеруем программы в книге более привычной нашему читателю цифровой записью: 3.1, 6.1, 6.2 и т. д.

В файле `Readme.doc` приведены пояснения автора книги, помогающие работать с содержащимися в архиве файлами.

Предметный указатель

3

32-разрядный процессор 19

6

64-разрядный процессор 19, 20

A

Acorn Archimedes, система 302

Acorn, компания 29

Adobe Creative Cloud 21

AMD64, архитектура 20

Apple Computers, компания 21

ARM Holdings, компания 21

ARM, микропроцессор 17, 20, 27

◊ адресация памяти 60

◊ архитектура 59

◊ графический процессор 301

◊ длина слов 59

◊ кеш 301

◊ конвейер 299

◊ набор команд 28, 282, 298

◊ память 300

◊ сопроцессоры 299

ARMv8, архитектура 61, 301

ASCII, код 80

B

BBC BASIC, язык программирования 29

BBC Micro, плата 14

Broadcom, компания 23

C

C, язык программирования 17, 29, 179

◊ использование функций в ассемблере 179

CISC-процессор 21

CubeSats, спутники 31

G

GCC, компилятор 198

◊ формат файла исходного кода 180

GDB, отладчик 127

◊ параметры сборки 137

Geany Programmer's Editor, редактор 48

GNU C, компилятор 16

GNU GCC, программное обеспечение 29

GNU, проект 16

GPIO, порт 19

◊ вводы и выводы 211

◊ контроллер 209

◊ описание контактов 222

◊ функции 207, 222

H

HDMI, порт 18

I

i386, архитектура 83

L

libc, библиотека 179

LIFO, структура 155

M

MagPi, журнал 15

Make, программа 83

Microsoft Office 365 21

N

Neon, процессор 247

◊ Neon Intrinsic 256

◊ ассемблер 249

◊ другие форматы команд 256

◊ команды загрузки и сохранения 253

◊ команды и типы данных 251

◊ массивы 257

◊ отличие от VFP 248

◊ режимы адресации 253

◊ чередование 247

P

Python, язык программирования 30

R

Raspberry Pi

- ◇ 64-разрядная ОС 302
- ◇ возможности 14
- Raspberry Pi (*прод.*)
- ◇ история 21
- ◇ объем памяти 22
- ◇ ОС для работы с 16, 19
- ◇ стоимость 15
- ◇ форматы 21
- ◇ чипы ARM 33
- Raspberry Pi 2B (v1.2) 20
- Raspberry Pi 400 20
- Raspberry Pi Foundation, компания 19
- Raspberry Pi Zero 19
- Raspbian, ОС 16, 19
- RISC-процессор 21, 32

S

SIMD, технология 226
SoC, технология 297

T

Terminal, окно 35
Thumb, набор команд 272

- ◇ Thumb-2 281
- ◇ доступ к регистрам 278
- ◇ изменения 282

- ◇ команды ARMv7 280
- ◇ комментарии 275
- ◇ объединение кода ARM и 276
- ◇ одно- и многорегистровые команды 279
- ◇ операторы стека 279
- ◇ отличия от ARM 272
- ◇ функции в 279

V

VFP, сопроцессор 225

- ◇ архитектура 225
- ◇ векторная арифметика 236
- ◇ векторные операции 240
- ◇ выбор типа оператора 242
- ◇ вывод на экран 229
- ◇ загрузка, хранение и перемещение данных 233
- ◇ отличие от Neon 248
- ◇ преобразование точности 235
- ◇ регистровый файл 227
- ◇ сборка и отладка с помощью GDB 231
- ◇ системные регистры 237
- ◇ скалярные операции 240
- ◇ условное исполнение 238

X

x86, архитектура 20
x86-64, архитектура 20

A

Адрес памяти 61

- ◇ обратная запись 146

Адресация памяти

- ◇ косвенная 141
- ◇ относительная 148
- ◇ постиндексированная 146
- ◇ предварительно индексированная 143

Аппаратные векторы 289

Ассемблер 27

- ◇ ошибки 40
- ◇ создание файла 200
- ◇ структура 32

Б

Байтовые условия 148
Бесконечный цикл 126
Бит 64

- ◇ очистка 90

Блок передачи 150

B

Ввода/вывода, операции 78
Вектор 288

- ◇ аппаратный 289
- ◇ программный 289
- ◇ с плавающей точкой 23

Ветви

- ◇ обмен 107
- ◇ расчет 124

Ветвление 239

- ◇ команды 103

Внутренняя функция 256
Вращение 111

- ◇ расширенное 112

Вывод

- ◇ информации 186
- ◇ на экран 79

Выравнивание данных 164
Вычислительный модуль 23
Вычитание 71

Г

Графический процессор 23, 301

Д

Дамп памяти 136

Двоичная система счисления 49

Двоичное число 49

◇ вычитание 54

◇ преобразование в десятичное 50

◇ преобразование в шестнадцатеричное 51

◇ сложение 53

Деление 74

◇ остаток 119

Десятичное число 50

◇ преобразование в двоичное 51

Дизассемблирование программ на С 198

Динамическое связывание 44

Директивы 162

◇ хранения данных 162

Дополнительный код 56

Доступ к байтам памяти 144

З, И

Знаковый разряд 226

Интегрированная среда разработки 48

Исходный файл 36, 83

К

Категории процессоров 19

Клавиатурные вычисления 20

Командная строка 35

Комментарий 43, 47

Компилятор 29

Конвейерная обработка 123

Контрольный символ 80

Копирование блока данных 153

Космический центр Суррея 31

Кросс-компилятор 33

Кэш 301

Л

Логические операции 86

◇ И 86

◇ ИЛИ 87

◇ исключающее ИЛИ 87

◇ команды 88

М

Макрос 165

◇ вызов 166

◇ подключение 168

◇ применение 165

◇ пример использования 270

Мантисса 226

Матрица 262

Матричная математика 262

◇ умножение 265

Матричные вычисления 261

Машинный код 27, 30

Микропроцессор 27

Мнемоника 28

Модуль с плавающей точкой 23

О

Обработка

◇ исключений 287, 292

Обработка данных 67

Операционная система 16, 19

Отладка 33

◇ с использованием GDB 126

Отображение памяти 207

Ошибки 126

◇ ассемблера 40

◇ поиск 33

П

Параллельная обработка 34

Передача данных 139

Перемещение данных 75

◇ отрицательное 75

Планирования, эффект 269

Побитовые операторы 238

Преобразование

◇ строки в число 184

◇ регистра символов 89

Прерывание 294

◇ возврат из 295

◇ написание процедуры 296

◇ решение о 295

Программирование 17

Программные векторы 289

Программный счетчик 63, 122

Пузырьковая сортировка 195

Р

Регистр 42, 61, 150

◇ векторной загрузки 229

◇ настройка 290

◇ программный счетчик 63, 122

◇ системных вызовов 94

◇ состояния программы 63

◇ сохранение и восстановление содержимого 153

◇ ссылок 66, 104

◇ указатель стека 155

◇ управления VFP 237

Регистровый файл 227

Режим

- ◊ быстрого прерывания 294
- ◊ запроса прерывания 294

С

Связывание файлов 44

Сдвиг 108

- ◊ арифметический 110
- ◊ логический 108

Синтаксис 29

Системный вызов 79

- ◊ дизассемблирование 206

Системный чип 23

Скаляр 265

Сложение 68

Совместимость 18

Сортировка

- ◊ пузырьковая 195
- ◊ чисел 77

Сравнение 76

- ◊ использование команд 104
- ◊ по нулю 284

Среда программирования 16

Стек 155

- ◊ передача и извлечение данных 155
 - ◊ применение 159
 - ◊ реализация 156
 - ◊ рост 157
 - ◊ с двумя указателями 156
 - ◊ указатель 155
 - ◊ указатель фрейма 160
 - ◊ фрейм 160
- Степень 226
- Столлман, Ричард 16
- Суффикс S 65

Т

Тактовая частота 34

Точка останова 132

- ◊ метки 135

Транспонирование 259

У

Указатель

- ◊ стека 66
 - ◊ фрейма 160
 - ◊ переменные 205
- Умножение 72
- ◊ длинное 116
 - ◊ с накоплением 118
 - ◊ умное 120

Унифицированный язык ассемблера (UAL) 281

Условное выполнение 95

Ф

Файл

- ◊ атрибуты 177
 - ◊ исследование исполняемого 182
 - ◊ открытие 175
 - ◊ права доступа к нему 176
 - ◊ работа с ним 171
 - ◊ регистровый 227
 - ◊ создание 172
- Флаг 64
- ◊ коды состояния с одним 97
 - ◊ коды, проверяющие несколько 100
 - ◊ проверка 91
 - ◊ установка 64
- Фонд свободного программного обеспечения 16
- Фрейм стека 160, 202
- ◊ указатель 160
- Фреймворк C 199
- Функции 188
- ◊ использование регистров 190
 - ◊ процедуры вывода 193
 - ◊ сохранение ссылок и флагов 192
 - ◊ стандарты 188

Ц

Центр истории вычислительной техники 24

Центральный процессор 27

Ч

Числа с плавающей точкой 225

- ◊ вывод на экран 229
 - ◊ работа с 225
 - ◊ сопроцессор VFP 225
- Число одинарной и двойной точности 226
- Чтение с клавиатуры 81

Ш

Шестнадцатеричное число 49

- ◊ преобразование в десятичное 53

Я

Ядро 34, 180

Язык ассемблера 27

Язык программирования

- ◊ высокого уровня 30
 - ◊ низкого уровня 30
- Языковые уровни 30
- Ярлык 137

Эта книга — наиболее полное и подробное руководство по изучению языка ассемблера для архитектуры ARM на примере микрокомпьютеров Raspberry Pi с операционной системой Raspberry Pi OS. Вы узнаете, как использовать все необходимые инструменты, чтобы создавать собственные программы на ассемблере для процессоров ARM.

Написанная специально для новичков, книга рассказывает о базовых принципах и постепенно раскрывает всю необходимую информацию, призванную помочь читателю стать опытным программистом. Автор дает подробные описания и предлагает множество примеров программ, которые можно попробовать в деле. Описанные в книге программы совместимы со всеми моделями Raspberry Pi, включая Pi 4, 400 и 3.

Вы изучите:

- Практический подход к программированию с наглядными примерами
- Ассемблер и компоновщик GCC
- Регистры и внутреннюю архитектуру ARM
- Использование вызовов операционной системы
- Отладку с GDB
- Использование библиотеки libc
- Функции GPIO
- ARM Neon
- Команды Thumb



Брюс Смит — в прошлом технический редактор журнала «Acorn User Magazine», основатель издательств Bruce Smith Books и Dabs Press. Большинство написанных им книг посвящены компьютерным технологиям и программированию. Брюс стал одним из первых авторов, написавших о чипах ARM в момент их появления в 1987 году. Книги Брюса Смита переведены на пять языков.

Отзывы читателей на Amazon.com

«Эта книга — великолепное введение в программирование для ARM на Raspberry Pi»

«Первая компьютерная книга, которую я прочитал с удовольствием, лежа в постели, в то время как другие больше похожи на лекарство от бессонницы»

«Отличное пособие по ассемблеру для ARM»



Исходный код программ из книги можно скачать по ссылке: <ftp://ftp.bhv.ru/9785977568012.zip>, а также со страницы книги на сайте bhv.ru



191036, Санкт-Петербург,
Гончарная ул., 20
Тел.: (812) 717-10-50,
339-54-17, 339-54-28
E-mail: mail@bhv.ru
Internet: www.bhv.ru

ISBN 978-5-9775-6801-2

