

Изучив это руководство, вы сможете писать и читать исходный код на ассемблере и применять ассемблер совместно с языками программирования высокого уровня, используя необходимые для этого инструменты. В книге главным образом рассматривается программирование в системе Linux, поскольку это самая простая и удобная платформа для изучения языка ассемблера. В заключительных главах дается общее представление об использовании ассемблера в ОС Windows.

Ассемблерный код представлен в виде полноценных завершенных программ, поэтому вы можете протестировать их на своем компьютере, изменять их, экспериментировать с ними и даже «сломать» их.

Рассматриваемые темы:

- как работает процессор и память компьютера;
- как компиляторы языков высокого уровня генерируют машинный код;
- профессиональные методы анализа ошибок в программах;
- как заставить программу работать;
- защита от вредоносных программ;
- что такое AVX.

Книга адресована читателям, имеющим базовые знания в области программирования на языках высокого уровня.

Файлы для работы с книгой можно скачать на сайте издательства www.dmkpress.com

Интернет-магазин: www.dmkpress.com

Оптовая продажа: КТК «Галактика»
books@aliens-kniga.ru

ДМК
издательство
www.dmk.ru
Apress
www.apress.com



Программирование на ассемблере x64

Программирование на ассемблере x64

От начального уровня
до профессионального
использования AVX

Йо Ван Гуй

ДМК
издательство

Йо Ван Гуй

Программирование на ассемблере x64

**От начального уровня
до профессионального
использования AVX**

Beginning x64 Assembly Programming

From Novice to AVX Professional

Jo Van Hoey

Программирование на ассемблере x64

От начального уровня
до профессионального
использования AVX

Йо Ван Гуй



Москва, 2021

УДК 004.4
ББК 32.97
Г93

Йо Ван Гуй

Г93 Программирование на ассемблере x64: от начального уровня до профессионального использования AVX / пер. с англ. А. В. Снастина. – М.: ДМК Пресс, 2021. – 332 с.: ил.

ISBN 978-5-97060-929-3

Цель этой книги – показать, как используются инструкции языка ассемблера, и научить читателей программировать на нем – начиная с создания самых простых программ и заканчивая использованием расширенной системы команд Advanced Vector Extensions (AVX). Для изучения практической части потребуются знания основы программирования на каком-либо языке высокого уровня, например С.

Теоретический материал сведен к необходимому минимуму: немного информации о двоичных числах, краткое описание логических операторов и кое-что об основах линейной алгебры. Исходный ассемблерный код представлен в виде завершенных программ, которые читатель может протестировать на своем компьютере и поэкспериментировать с ними. Рассматриваются инструментальные средства, которыми можно воспользоваться, и потенциальные проблемы при использовании этих инструментов.

Основная часть книги содержит информацию о применении ассемблера в ОС Linux; несколько заключительных глав описывают работу в Windows.

Книга предназначена для программистов на языках высокого уровня, а также для системных инженеров и инженеров по обеспечению безопасности, работающих в области исследования вредоносного программного обеспечения.

УДК 004.4
ББК 32.97

First published in English under the title Beginning x64 Assembly Programming; From Novice to AVX Professional by Jo Van Hoey, edition: 1

Copyright © Jo Van Hoey, 2019 *

This edition has been translated and published under licence from APress Media, LLC, part of Springer Nature. APress Media, LLC, part of Springer Nature takes no responsibility and shall not be made liable for the accuracy of the translation.

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

ISBN (англ.) 978-1-4842-5075-4
ISBN (рус.) 978-5-97060-929-3

© 2019 by Jo Van Hoey
© Оформление, издание, перевод, ДМК Пресс, 2021

Оглавление

Об авторе	12
О техническом рецензенте	13
Предисловие от издательства	14
Введение	15
Прежде чем начать	17
Глава 1. Самая первая программа	19
Редактирование, ассемблирование, связывание и запуск (или отладка).....	20
Структура программы на ассемблере	25
Раздел section .data	25
Раздел section .bss.....	26
Раздел section .txt.....	27
Резюме.....	29
Глава 2. Двоичные и шестнадцатеричные числа и регистры	30
Краткий вводный курс по двоичным числам.....	30
Целые числа	31
Числа с плавающей точкой	32
Краткий вводный курс по регистрам.....	32
Регистры общего назначения	33
Регистр счетчика команд (rip)	34
Регистр флагов	34
Регистры xmm и ymm	35
Резюме.....	35
Глава 3. Анализ программ с помощью отладчика: GDB	36
Начало отладки	36
Двигаемся дальше	42
Некоторые дополнительные команды отладчика GDB	44
Немного улучшенная версия программы hello, world	45
Резюме.....	47

Глава 4. Следующая программа: Alive and Kicking	48
Анализ программы alive	49
Вывод	53
Резюме	56
Глава 5. Ассемблер основан на логике	57
Логический оператор NOT	57
Логический оператор OR	57
Логический оператор XOR	58
Логический оператор AND	58
Резюме	59
Глава 6. Отладчик Data Display Debugger	60
Работа с отладчиком DDD	60
Резюме	63
Глава 7. Переходы и циклы	64
Установка SimpleASM	64
Использование SASM	64
Резюме	72
Глава 8. Память	73
Обследование памяти	73
Резюме	80
Глава 9. Целочисленная арифметика	81
Основы использования целочисленной арифметики	81
Изучение арифметических инструкций	84
Резюме	87
Глава 10. Стек	88
Изучение работы стека	88
Наблюдение за стеком	91
Резюме	93
Глава 11. Арифметика с плавающей точкой	94
Сравнение чисел с обычной и двойной точностью	94
Кодирование с применением чисел с плавающей точкой	96
Резюме	98
Глава 12. Функции	99
Создание простой функции	99

Еще о функциях	100
Резюме.....	102
Глава 13. Выравнивание стека и фрейм стека	103
Выравнивание стека.....	103
Более подробно о фреймах стека	105
Резюме.....	106
Глава 14. Внешние функции	107
Создание и связывание функций.....	107
Расширенная версия makefile.....	110
Резюме.....	111
Глава 15. Соглашения о вызовах функций.....	112
Аргументы функций.....	113
Схема стека	116
Сохранение регистров.....	118
Резюме.....	120
Глава 16. Операции с битами	121
Основные положения.....	121
Арифметика	126
Резюме.....	129
Глава 17. Работа с битами	130
Другие способы изменения битов.....	130
Переменная bitflags.....	132
Резюме.....	133
Глава 18. Макрокоманды	134
Создание макроса.....	134
Использование objdump.....	136
Резюме.....	137
Глава 19. Ввод и вывод в консоли	138
Использование средств ввода/вывода	138
Обработка переполнений	140
Резюме.....	143
Глава 20. Файловый ввод/вывод	144
Использование системных вызовов.....	144
Обработка файла	145

Условное ассемблирование	152
Инструкции для обработки файлов.....	152
Резюме.....	153
Глава 21. Командная строка	154
Доступ к аргументам командной строки.....	154
Отладка программы с аргументами командной строки	155
Резюме.....	157
Глава 22. Использование ассемблера в коде C.....	158
Создание файла исходного кода на языке C.....	158
Создание файла исходного кода на ассемблере.....	160
Резюме.....	163
Глава 23. Встроенный ассемблер.....	164
Простой встроенный ассемблерный код	164
Расширенный встроенный ассемблерный код.....	166
Резюме.....	169
Глава 24. Строки	170
Обработка строк	170
Сравнение и сканирование строк	174
Резюме.....	178
Глава 25. Предъявите ваш идентификатор	179
Использование инструкции <code>cuid</code>	179
Использование инструкции <code>test</code>	181
Резюме.....	183
Глава 26. SIMD.....	184
Скалярные данные и упакованные данные.....	184
Невыровненные и выровненные данные	186
Резюме.....	187
Глава 27. Работа с битами регистра <code>mxcsr</code>	189
Анализ программы.....	194
Резюме.....	196
Глава 28. Выравнивание для SSE.....	197
Пример без выравнивания	197
Пример с выравниванием.....	200
Резюме.....	203

Глава 29. SSE-инструкции для работы с упакованными целыми числами	204
SSE-инструкции для работы с целыми числами	204
Анализ исходного кода	206
Резюме	206
Глава 30. Обработка строк средствами SSE	207
Управляющий байт <code>imm8</code>	208
Использование управляющего байта <code>imm8</code>	209
Биты 0 и 1	209
Биты 2 и 3	209
Биты 4 и 5	210
Бит 6	211
Зарезервированный бит 7	211
Флаги	211
Резюме	212
Глава 31. Поиск символа в строке	213
Определение длины строки	213
Поиск в строках	216
Резюме	219
Глава 32. Сравнение строк	220
Строки с неявно заданной длиной	220
Строки с явно заданной длиной	222
Резюме	226
Глава 33. Перемешиваем данные	227
Основные принципы операций перемешивания	227
Перемешивание в случайном порядке	231
Перемешивание в обратном порядке	233
Перемешивание вращением	234
Перемешивание байтов	234
Резюме	236
Глава 34. SSE-инструкции: маски строк	237
Поиск символов	237
Поиск символов из заданного диапазона	243
Поиск подстроки	246
Резюме	249

Глава 35. AVX	250
Проверка поддержки AVX	250
Пример программы с использованием AVX	252
Резюме	256
Глава 36. Операции с матрицами с использованием AVX	257
Пример исходного кода для операций с матрицами	257
Вывод матрицы: <code>print4x4</code>	264
Умножение матриц: <code>multi4x4</code>	265
Обращение матрицы: <code>inverse4x4</code>	268
Теорема Гамильтона–Кэли	268
Алгоритм Фаддеева–Леве́рье	268
Исходный код	269
Резюме	273
Глава 37. Транспонирование матриц	274
Пример исходного кода для транспонирования матриц	274
Версия с использованием неупакованных данных	277
Версия с применением перемешивания	282
Резюме	285
Глава 38. Оптимизация производительности	286
Производительность вычисления транспонированной матрицы	286
Производительность вычисления следа матрицы	292
Резюме	297
Глава 39. Приветствуем мир Windows	298
Начинаем изучение	298
Пишем код в Windows	300
Отладка	302
Системные вызовы	302
Резюме	303
Глава 40. Использование Windows API	304
Вывод в консоли	304
Создание окон Windows	307
Резюме	308
Глава 41. Функции в Windows	309
Использование более четырех аргументов функции	309
Обработка значений с плавающей точкой	314
Резюме	316

Глава 42. Функции с переменным числом аргументов	317
Функции с переменным числом аргументов в Windows	317
Обработка смешанных значений	319
Резюме.....	320
Глава 43. Работа с файлами в Windows.....	321
Резюме.....	324
Послесловие. Что дальше?	325
Предметный указатель	326

Об авторе



Йо Ван Гуй (Jo Van Hoesy) обладает 40-летним опытом работы в сфере информационных технологий, выполняя разнообразные функции в многочисленных ИТ-компаниях с использованием различных компьютерных платформ. Недавно он уволился из компании IBM, где являлся менеджером по работе с клиентами, использующими ПО для мейнфреймов. Йо всегда интересовался проблемами безопасности в области ИТ, а знание языка ассемблера представляет собой весьма важный профессиональный навык при защите ИТ-инфраструктуры от атак и вредоносных программ.

О техническом рецензенте



Пол Коэн (Paul Kohen) присоединился к компании Intel Corportion в те далекие дни, когда только еще создавалась архитектура x86, начиная с микропроцессора 8086, и уволился из Intel после 26 лет работы, связанной с продажами, маркетингом и управлением. В настоящее время Пол сотрудничает с компанией Douglas Technology Group, деятельность которой сосредоточена на издании книг от имени Intel и других корпораций. Кроме того, Пол читает курс, превращающий учеников средней школы и студентов младших курсов в реальных уверенных в себе предпринимателей, сотрудничая с Young Entrepreneurs Academy (YEA), а также является членом транспортного суда в городе Бивертон (шт. Орегон) и совета директоров нескольких некоммерческих организаций.

Предисловие от издательства

ОТЗЫВЫ И ПОЖЕЛАНИЯ

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв на нашем сайте www.dmkpress.com, зайдя на страницу книги и оставив комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу dmkpress@gmail.com; при этом укажите название книги в теме письма.

Если вы являетесь экспертом в какой-либо области и заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу http://dmkpress.com/authors/publish_book/ или напишите в издательство по адресу dmkpress@gmail.com.

СПИСОК ОПЕЧАТОК

Хотя мы приняли все возможные меры для того, чтобы обеспечить высокое качество наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг – возможно, ошибку в основном тексте или программном коде, – мы будем очень благодарны, если вы сообщите нам о ней. Сделав это, вы избавите других читателей от недопонимания и поможете нам улучшить последующие издания этой книги.

Если вы найдете какие-либо ошибки в коде, пожалуйста, сообщите о них главному редактору по адресу dmkpress@gmail.com, и мы исправим это в следующих тиражах.

НАРУШЕНИЕ АВТОРСКИХ ПРАВ

Пиратство в интернете по-прежнему остается насущной проблемой. Издательство «ДМК Пресс» очень серьезно относится к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконной публикацией какой-либо из наших книг, пожалуйста, пришлите нам ссылку на интернет-ресурс, чтобы мы могли применить санкции.

Ссылку на подозрительные материалы можно прислать по адресу dmkpress@gmail.com.

Мы высоко ценим любую помощь по защите наших авторов, благодаря которой мы можем предоставлять вам качественные материалы.

Введение

Изучение программирования на ассемблере может оказаться обескураживающим, но совсем не потому, что это язык, не прощающий ошибок, ведь компьютер будет «одобрять» ваши действия при каждом удобном случае. А если это не так, то, возможно, где-то в программе скрывается необнаруженная ошибка, которая «укусит» вас во время выполнения программы. Сверх всего прочего кривая сложности обучения весьма крута, язык загадочный и не сразу понятный, официальная документация Intel ошеломляюще велика, а доступные инструменты разработки обладают весьма специфическими особенностями.

Эта книга научит вас программировать на ассемблере, начиная с самых простых программ и постепенно осваивая путь к овладению программированием с использованием расширенной системы команд Advanced Vector Extensions (AVX). Прочитав эту книгу полностью, вы сможете писать и читать код на ассемблере, ассемблерный код, объединенный с языками высокого уровня, поймете, что такое AVX и многое другое. Цель этой книги – показать, как используются инструкции языка ассемблера. Это не руководство по стилю программирования или по оптимизации производительности кода. После того как вы освоите базовые знания об ассемблере, можно будет продолжить обучение по теме оптимизации кода. Эта книга не должна быть вашей первой книгой по программированию: если вы никогда раньше не программировали, то отложите эту книгу на время и изучите основы программирования на каком-либо языке высокого уровня, например на C.

Весь исходный код, используемый здесь, доступен по ссылке Download Source Code на сайте www.apress.com/9781484250754. Исходный код в книге представлен в настолько простом виде, насколько это вообще возможно, т. е. без каких-либо графических пользовательских интерфейсов, без излишеств («бантиков и рюшечек»), без средств проверки на ошибки. Добавление всех этих замечательных функциональных возможностей увело бы нас от истинной цели: изучение языка ассемблера.

Теоретическая часть сведена к необходимому минимуму: немного информации о двоичных (бинарных) числах, краткое описание логических операторов и кое-что об основах линейной алгебры. Здесь не рассматриваются операции преобразования чисел с плавающей точкой. Если требуется преобразовать двоичные или шестнадцатеричные числа, то найдите веб-сайт, который сделает это за вас. Не теряйте времени на вычисления вручную. Помните о главной цели: изучение ассемблера.

Исходный ассемблерный код представлен в виде завершенных программ, так что вы можете протестировать их на своем компьютере, поэкспериментировать, изменять их и «ломать»...

Кроме того, будут рассматриваться инструментальные средства, которыми можно воспользоваться, и потенциальные проблемы при использовании этих

инструментов. Выбор правильных инструментов чрезвычайно важен для преодоления крутой кривой сложности обучения. Иногда будут встречаться ссылки на книги, статьи, прочие документы и веб-сайты, которые могут оказаться полезными при обучении или содержать более подробные описания.

Автор не намеревался предоставить исчерпывающий курс по всем инструкциям ассемблера. Это невозможно сделать в одной книге (взгляните на размеры справочных руководств компании Intel). Читателю предоставлена возможность попробовать на практике основные компоненты, чтобы получить представление о том, куда двигаться дальше. При работе с этой книгой вы получите знания, необходимые для самостоятельного дальнейшего глубокого изучения определенных предметных областей ИТ. После завершения чтения этой книги вы сможете изучать справочные руководства компании Intel и понимать (во всяком случае, пытаться понять) смысл их содержимого.

Основная часть книги содержит информацию о применении ассемблера в Linux, потому что это самая простая и удобная платформа для изучения языка ассемблера. В заключительной части книги представлено несколько глав, описывающих методику использования ассемблера в Windows. Вы сами убедитесь в том, что, вооружившись ассемблером Linux, гораздо проще осваивать ассемблер в Windows.

Существует несколько трансляторов ассемблера, доступных для использования с процессорами Intel, например FASM, MASM, GAS, NASM и YASM (список далеко не полный). В этой книге используется транслятор NASM, поскольку он многоплатформенный: доступен для Linux, Windows и macOS. Кроме того, он обладает относительно большой пользовательской базой. Но не стоит беспокоиться; если вы знаете один ассемблер, то с легкостью освоите любой другой «диалект».

Исходный код в книге тщательно проверен и протестирован. Но если обнаружатся какие-либо опечатки в тексте или ошибки в программах, мы не несем за это никакой ответственности. Автор обвиняет в этом двух своих котов, которые любят прыгать на клавиатуру во время его работы.

Идеи и мнения, представленные в этой книге, принадлежат лично автору и не всегда представляют позицию, стратегии или мнения компании IBM.

Прежде чем начать

Для чтения этой книги вы должны знать некоторые основные темы.

- **Вы должны уметь устанавливать и настраивать программное обеспечение виртуализации (VMware, VirtualBox или аналогичное ПО).** Если эти программы вам неизвестны, то загрузите бесплатное ПО Oracle VirtualBox (<https://www.virtualbox.org>), установите его и научитесь его использовать, установив, например, Ubuntu Desktop Linux как гостевую операционную систему (ОС). ПО виртуализации позволяет устанавливать различные гостевые ОС на основном компьютере, и если вы что-то напутали в такой гостевой ОС, то можете ее удалить и установить заново. Или если имеются мгновенные снимки состояния системы, то можно вернуться к предыдущей версии гостевой ОС. Другими словами, при экспериментах с гостевой ОС не будет нанесено никакого вреда основной операционной системе. В интернете можно найти огромное количество ресурсов, описывающих VirtualBox и другие решения с использованием ПО виртуализации.
- **Вы должны обладать базовыми знаниями об использовании интерфейса командной строки Linux.** В книге используется Ubuntu Desktop Linux и командная строка этой ОС, начиная с главы 1. Если хотите, можете работать в другом дистрибутиве Linux, но при этом необходимо убедиться в том, что имеется возможность установить все инструментальные средства, применяемые в данной книге (NASM, GCC, GDB, SASM и т. д.). Требуются следующие базовые знания: как установить ОС, как устанавливается дополнительное ПО, как запустить терминал с приглашением (промптом) командной строки, а также как создавать, перемещать, копировать и удалять каталоги и файлы в командной строке. Также необходимо уметь использовать утилиты `tag`, `grep`, `find`, `ls`, `time` и т. п. Вы должны знать, как запустить и использовать текстовый редактор. Не требуется никаких продвинутых знаний о Linux, нужны только самые простые, базовые навыки выполнения задач, для того чтобы следовать описаниям, приведенным в этой книге. Если вы незнакомы с ОС Linux, то немного поработайте в ней, чтобы освоить ее использование. В интернете существует множество небольших по объему, но качественных руководств для начинающих (например, <https://www.guru99.com/unix-linux-tutorial.html>). Вы сами убедитесь в том, что после изучения ассемблера на компьютере с ОС Linux освоение ассемблера в других ОС станет не таким уж трудным делом.
- **Вы должны обладать некоторыми базовыми знаниями в области программирования на языке C.** В книге в некоторых случаях применяются функции на языке C для упрощения примеров ассемблерного кода. Кроме того, будет показано, как организовать интерфейс с языком высокого уровня, таким как C. Если вы не знаете C, но намерены в полной мере освоить содержимое этой книги, то рекомендуется изучить пару

бесплатных курсов по C, например tutorialspoint.com. Нет необходимости проходить полный курс, просто внимательно изучите несколько программ на этом языке. В дальнейшем всегда можно вернуться к курсу по C, чтобы узнать больше подробностей.

ЗАЧЕМ НУЖНО ИЗУЧАТЬ АССЕМБЛЕР

Знание ассемблера дает некоторые преимущества:

- вы узнаете, как работает (центральный) процессор (ЦПУ) и оперативная память;
- вы узнаете, как совместно работают компьютер и операционная система;
- вы поймете, как компиляторы языков высокого уровня генерируют код на машинном языке, а эти знания могут помочь писать более эффективный код;
- вы получите более эффективные средства для анализа ошибок в программах;
- вы получите огромное удовольствие, когда, наконец, ваша программа заработает правильно;
- и причина, по которой я написал эту книгу: если необходимо исследовать вредоносное ПО, то в вашем распоряжении есть только машинный код без исходного кода. Если вы хорошо понимаете код ассемблера, то сможете проанализировать вредоносную программу, выполнить необходимые действия и принять превентивные меры.

РУКОВОДСТВА КОМПАНИИ INTEL

Справочные руководства компании Intel содержат все, что может потребоваться при программировании микропроцессоров Intel. Но информация весьма сложна для освоения начинающими программистами. По мере чтения данной книги вы обнаружите, что описания в этих руководствах Intel постепенно становятся все более понятными. В книге часто встречаются ссылки на эти солидные тома информации.

Справочные руководства Intel можно найти здесь:

<https://software.intel.com/en-us/articles/intel-sdm>.

Только не пытайтесь распечатать их на бумаге – пожалейте деревья, которые вы можете уничтожить. Бегло просмотрите руководства, чтобы убедиться в том, насколько исчерпывающими, подробными и формализованными документами они являются. Попытка изучения ассемблера по этим руководствам может оказаться обескураживающе неудачной. Особый интерес для нас представляет Volume 2 (том 2), в котором вы найдете подробные описания программных инструкций ассемблера.

Полезный источник информации можно найти здесь: <https://www.felixcloutier.com/x86/index.html>. На этом сайте представлен список всех инструкций с краткими описаниями их использования. Если предоставленная здесь информация окажется недостаточной, то вы всегда можете вернуться к справочным руководствам Intel или обратиться к вашему надежному другу Google.

Глава 1

Самая первая программа

Многие поколения разработчиков начали свою карьеру программиста с изучения способа вывода на экран компьютерного дисплея фразы `hello, world`. Эта традиция была заложена в 1970-е гг. Брайаном Керниганом (Brian W. Kernighan) в книге, которую он написал вместе с Деннисом Ритчи (Dennis Ritchie) «The C Programming Language» («Язык программирования С»). Керниган принимал участие в разработке языка программирования С в компании Bell Labs. С тех пор язык С значительно изменился, но он остается языком, с которым должен быть знаком каждый уважающий себя программист. Большинство «новейших», «модных» языков программирования в той или иной степени являются наследниками языка С. Язык С иногда называют переносимым языком ассемблера, и как программист, стремящийся к овладению ассемблером, вы должны знать С. Уважая традицию, начнем с программы на ассемблере, выводящей фразу `hello, world` на экран. В листинге 1.1 показан исходный код версии на языке ассемблера этой программы, которую мы будем анализировать далее в этой главе.

Листинг 1.1. *hello.asm*

```
;hello.asm
section .data
    msg db "hello, world",0
section .bss
section .text
    global main
main:
    mov     rax, 1        ; 1 = запись.
    mov     rdi, 1        ; 1 = в поток стандартного вывода stdout.
    mov     rsi, msg      ; Выводимая строка в регистре rsi.
    mov     rdx, 12       ; Длина строки без конечного 0.
    syscall              ; Вывод строки.
    mov     rax, 60       ; 60 = код выхода из программы.
    mov     rdi, 0        ; 0 = код успешного завершения программы.
    syscall              ; Выход из программы.
```

РЕДАКТИРОВАНИЕ, АССЕМБЛИРОВАНИЕ, СВЯЗЫВАНИЕ И ЗАПУСК (или ОТЛАДКА)

Существует множество хороших текстовых редакторов, как бесплатных, так и коммерческих. Следует искать редактор, поддерживающий подсветку синтаксиса для версии ассемблера NASM 64-bit. В большинстве случаев придется скачать и установить некоторый подключаемый модуль (plugin) или дополнительный пакет, обеспечивающий подсветку синтаксиса.

Примечание. В этой книге мы будем писать код для версии Netwide Assembler (NASM). Существуют и другие версии ассемблера, например YASM, FASM, GAS или MASM компании Microsoft. И как обычно в мире ИТ, иногда возникают оживленные дискуссии о том, какая версия ассемблера является самой лучшей. В этой книге используется версия NASM, потому что она доступна для ОС Linux, Windows и macOS, а кроме того, из-за наличия большого сообщества пользователей NASM. Справочное руководство по NASM можно найти здесь: www.nasm.us.

В этой книге будет использоваться текстовый редактор gedit с установленным дополнительным файлом (модулем) подсветки синтаксиса ассемблера. Gedit – стандартный текстовый редактор, доступный в системе Linux¹, – здесь используется Ubuntu Desktop 18.04.2 LTS. Файл (модуль) поддержки подсветки синтаксиса можно найти здесь: <https://wiki.gnome.org/action/show/Projects/GtkSourceView/LanguageDefinitions>. Загрузите файл *asm-intel.lang*, скопируйте его в каталог */usr/share/gtksourceview*0/language-specs/*, заменив символ звездочки (*) на номер версии, установленной в вашей системе. При первом запуске gedit можно выбрать поддерживаемый язык программирования, в нашем случае Assembler (Intel), в нижней части окна gedit.

На экране файл *hello.asm*, приведенный в листинге 1.1, будет выглядеть так, как показано на рис. 1.1.

```

1 ; hello.asm
2 section .data
3     msg db      "hello, world",0
4 section .bss
5 section .text
6     global main
7 main:
8     mov     rax, 1           ; 1 = write
9     mov     rdi, 1           ; 1 = to stdout
10    mov     rsi, msg         ; string to display in rsi
11    mov     rdx, 12          ; length of the string, without 0
12    syscall                ; display the string
13    mov     rax, 60          ; 60 = exit
14    mov     rdi, 0           ; 0 = success exit code
15    syscall                ; quit

```

Рис. 1.1. Содержимое файла *hello.asm* в текстовом редакторе gedit

¹ Автор не совсем точен – в Linux gedit доступен только при наличии установленной рабочей среды GNOME. Для Windows и macOS все необходимое для работы gedit включено в дистрибутив. – Прим. перев.

Согласитесь, с подсветкой синтаксиса исходный код на ассемблере немного проще читать.

При написании ассемблерной программы на экране открыто два окна – окно `gedit`, содержащее исходный код на ассемблере, и окно с приглашением (промптом) командной строки в каталоге проекта, так что можно с легкостью переключаться между редактированием и управлением файлами проекта (выполнять ассемблирование и запускать программу, заниматься отладкой и т. п.). Разумеется, что в более крупных и сложных проектах такой подход вряд ли можно применять – потребуется интегрированная среда разработки (*integrated development environment* – IDE). Но прямо сейчас работы с простым текстовым редактором и интерфейсом командной строки (CLI – *command line interface*) вполне достаточно. Преимуществом этого процесса становится тот факт, что мы можем сосредоточиться на изучении ассемблера, а не многочисленных функциональных возможностей и особенностей IDE. В последующих главах будут рассматриваться полезные инструментальные средства и утилиты, некоторые из которых обладают графическим пользовательским интерфейсом, прочие же связаны с интерфейсом командной строки. Как бы то ни было, описание и использование IDE не относится к тематике этой книги.

Для любого упражнения из данной книги используется отдельный каталог *project*, содержащий все необходимые для проекта и генерируемые файлы.

Разумеется, в дополнение к текстовому редактору необходимо проверить наличие некоторых других установленных инструментальных средств, таких как `gcc`, `GDB`, `make` и `NASM`. Сначала потребуется `gcc` – используемый по умолчанию в Linux компилятор и редактор связей (линкер).

`gcc` – это сокращение от `GNU Compiler Collection` – стандартного инструментального средства компиляции и редактирования связей в Linux. (Аббревиатура `GNU` является рекурсивной: `GNU is Not Unix`. Использование рекурсивных аббревиатур для имен стало общепринятым в среде разработчиков еще в 1970-е гг., и все началось с программистов `LISP`. Да, эти старые шутки уже не смешны.)

В командной строке введите `gcc -v`. Компилятор `gcc` в ответ выведет несколько сообщений, если он уже установлен. Если комплект `gcc` отсутствует, то необходимо установить его с помощью следующей команды:

```
sudo apt install gcc
```

Далее выполните то же самое для `gdb -v` и `make -v`. Если вы не понимаете смысл этих команд, то, прежде чем продолжать чтение, вам необходимо пополнить знания о Linux.

Также необходимо установить `NASM` и пакет `build-essential`, содержащий ряд инструментальных средств, которые будут использоваться в дальнейшем. В `Ubuntu Desktop 18.04` это делается так:

```
sudo apt install build-essential nasm
```

После установки команда `nasm -v` выведет номер версии, если пакет `NASM` был установлен корректно. После установки всех перечисленных выше программных средств вы полностью готовы к реализации своей первой программы на ассемблере.

Введите исходный код программы, показанной в листинге 1.1, в текстовом редакторе, которым предпочитаете пользоваться, сохраните введенный код в файле с именем *hello.asm*. Как уже было отмечено ранее, для сохранения файлов этого первого проекта используется отдельный каталог. Каждая строка кода будет объяснена немного позже в этой главе. Обратите внимание на следующие характеристики исходного кода на ассемблере («исходный код» – это содержимое файла *hello.asm* с программными инструкциями, которые вы только что ввели):

- в исходном коде можно использовать символы табуляции, пробела и перехода на новую строку, чтобы сделать код более удобным для чтения;
- на каждой строке записывается только одна инструкция;
- текст после точки с запятой является комментарием. Другими словами, описанием, предназначенным для чтения человеком. Компьютеры не обращают никакого внимания на комментарии.

В текстовом редакторе создайте еще один файл, содержащий строки, приведенные в листинге 1.2.

Листинг 1.2. *makefile* для *hello.asm*

```
#makefile для hello.asm
hello: hello.o
    gcc -o hello hello.o -no-pie
hello.o: hello.asm
    nasm -f elf64 -g -F dwarf hello.asm -l hello.lst
```

На рис. 1.2 показано, как этот файл выглядит в редакторе *gedit*.



```
1 #makefile for hello.asm
2 hello: hello.o
3     gcc -o hello hello.o -no-pie
4 hello.o: hello.asm
5     nasm -f elf64 -g -F dwarf hello.asm -l hello.lst
```

Рис. 1.2. Содержимое файла *makefile* в редакторе *gedit*

Сохраните этот файл с именем *makefile* в том же каталоге, где находится файл *hello.asm*, и закройте окно редактора.

Файл *makefile* будет использоваться командой *make* для автоматизации сборки программы. Сборка (building) означает проверку исходного кода на ошибки, добавление всех необходимых сервисов операционной системы и преобразование исходного кода в последовательность инструкций, распознаваемых компьютером. В этой книге будут использоваться простые файлы *makefile*. Если вы хотите узнать больше о файлах *makefile*, то справочное руководство находится здесь:

<https://www.gnu.org/software/make/manual/make.html>.

Учебное руководство можно найти здесь:

<https://www.tutorialspoint.com/makefile/>.

Чтобы понять, что именно делает *makefile*, необходимо читать его снизу вверх. Упрощенное описание: утилита *make* работает с деревом зависимостей. Она определяет, что файл программы *hello* зависит от объектного файла *hello.o*. Затем обнаруживается, что файл *hello.o* зависит от файла исходного кода *hello.asm* и что файл *hello.asm* ни от чего не зависит. Далее *make* сравнивает даты последнего изменения файлов *hello.asm* и *hello.o*, и если дата изменения *hello.asm* более поздняя, то *make* выполняет строку после имени *hello.o*, т. е. *hello.asm*. Затем *make* снова начинает процедуру чтения *makefile* и обнаруживает, что дата изменения файла *hello.o* более поздняя, чем дата изменения файла *hello*. Поэтому выполняется строка после имени *hello*, т. е. *hello.o*.

В самой последней строке файла *makefile* NASM используется как ассемблер (программа ассемблирования, т. е. конечного этапа сборки). За ключом *-f* следует формат вывода, в данном случае *elf64*, означающий Executable and Linkable Format for 64-bit (выполняемый и связываемый формат для 64-битовой системы). Ключ *-g* означает, что необходимо включить отладочную информацию в специальном формате отладки, определенном после ключа *-F*. Здесь используется отладочный формат *dwarf*. Похоже, что программисты, разработавшие этот формат, являются большими поклонниками книг «Хоббит» и «Властелин колец» Дж. Р. Р. Толкиена, возможно, именно поэтому они решили, что DWARF (гном) должен стать великолепным дополнением к ELF (эльфу), на всякий случай, если вам это интересно. Если говорить более серьезно, то DWARF – это сокращение от Debug With Arbitrary Record Format² (отладка с использованием произвольного формата записей).

STABS – это еще один отладочный формат, не имеющий ничего общего с кровавыми битвами или ярким эльфийским светом (*stab* – ранение, нанесение ран; режущий глаза, ослепительный свет) из романов Толкиена, это название происходит от **S**ymbol **T**able **S**trings (строки таблицы символов). Здесь мы не будем использовать формат STABS, чтобы вы не запутались окончательно.

Ключ *-l* сообщает NASM о необходимости генерации файла листинга *.lst*. Файлы *.lst* будут использоваться для исследования результатов ассемблирования. NASM создает объектный файл с расширением *.o*. В дальнейшем этот объектный файл будет использоваться редактором связей (линкером).

Примечание. Часто случается так, что NASM выдает несколько непонятных сообщений и отказывается сгенерировать объектный файл. Иногда NASM начинает выдавать такие «жалобы» настолько часто, что может поставить программиста почти на грань безумия. В таких случаях чрезвычайно важно сохранять хладнокровие, выпить очередную чашечку кофе и еще раз внимательно просмотреть исходный код, потому что именно вы сделали что-то неправильно. Ассемблируя свою программу раз за разом, вы будете находить ошибки все быстрее и быстрее.

Когда вы наконец убедите NASM принять созданный объектный файл, он будет сразу же обработан редактором связей (линкером). Редактор связей рас-

² «Википедия» (англ.) дает другое толкование аббревиатуры DWARF – Debug With Attributed Record Format – отладка с использованием формата записей с атрибутами. Смысл несколько иной, но суть дела не меняется. – Прим. перев.

смаатривает предложенный объектный код и выполняет поиск в системе других необходимых файлов, обычно системных сервисов или прочих объектных файлов. Эти файлы объединяются со сгенерированным объектным кодом, и редактор связей создает выполняемый файл. Разумеется, редактор связей выдаст все возможные сообщения об отсутствующих компонентах и т. п. Если это произошло, выпейте еще одну чашечку кофе и проверьте свой исходный код и содержимое *makefile*.

В рассматриваемом здесь примере используется функциональность GCC (ниже воспроизводятся соответствующие строки из *makefile*):

```
hello: hello.o
    gcc -o hello hello.o -no-pie
```

Последние версии компилятора и линкера GCC по умолчанию генерируют выполняемый код, не зависящий от положения в памяти, или перемещаемый код (position independent executable – PIE). Это делается для защиты от хакеров, исследующих, как память используется программой, что в итоге позволяет им воздействовать на выполнение программы. В рассматриваемом здесь примере не создается перемещаемый выполняемый код, потому что такая программа слишком сложна для анализа (усложнение делается преднамеренно из соображений обеспечения безопасности). Поэтому в *makefile* добавлен ключ *-no-pie*.

В *makefile* можно добавлять комментарии, начинающиеся с символа «решетка» *#*:

```
#makefile for hello.asm
```

Здесь используется GCC для упрощения доступа к функциям стандартной библиотеки C из ассемблерного кода. Иногда мы будем пользоваться функциями языка C, чтобы сделать примеры ассемблерного кода более простыми. Но вы должны знать, что в Linux существует и другой широко известный GNU линкер *ld*.

Если несколько предыдущих абзацев оказались для вас непонятными, не волнуйтесь, выпейте еще чашечку кофе и продолжайте чтение – это всего лишь вспомогательная информация, которая не очень важна на текущем этапе. Просто помните, что *makefile* – ваш друг, который выполняет за вас огромную работу, а единственное, о чем следует беспокоиться сейчас, – не делать собственных ошибок.

В командной строке перейдите в каталог, где сохранены файлы *hello.asm* и *makefile*. Выполните команду *make* для ассемблирования и сборки программы, затем запустите созданную программу, набрав *./hello* в командной строке. Если появилось сообщение *hello, world* перед очередным промптом командной строки, то все работает правильно. Если сообщение не выведено, то в исходном коде была допущена опечатка или какая-то другая ошибка, поэтому необходимо еще раз проверить содержимое файлов *hello.asm* и *makefile*. Наполните очередную чашечку кофе и приступайте к отладке.

На рис. 1.3 показан пример вывода ожидаемого сообщения на экран.

```
jo@UbuntuDesktop:~/Desktop/linux64/gcc/01 hello $
jo@UbuntuDesktop:~/Desktop/linux64/gcc/01 hello $
jo@UbuntuDesktop:~/Desktop/linux64/gcc/01 hello $ make
nasm -f elf64 -g -F dwarf hello.asm -l hello.lst
gcc -o hello hello.o
jo@UbuntuDesktop:~/Desktop/linux64/gcc/01 hello $ ./hello
hello, worldjo@UbuntuDesktop:~/Desktop/linux64/gcc/01 hello $ █
```

Рис. 1.3. Вывод строки hello, world

СТРУКТУРА ПРОГРАММЫ НА АССЕМБЛЕРЕ

Рассматриваемая здесь первая программа демонстрирует базовую структуру любой ассемблерной программы. Ниже перечислены основные части программы на ассемблере:

- section .data
- section .bss
- section .txt

Раздел section .data

В разделе section .data объявляются и определяются инициализируемые данные в следующем формате:

```
<variable name>      <type>      <value>
```

Если переменная включена в раздел section .data, для нее выделяется память при ассемблировании и связывании исходного кода для создания выполняемого кода. Переменные имеют символьные имена и ссылки на локации в памяти, при этом переменная может занимать одну или несколько ячеек памяти. Имя переменной обозначает начальный адрес переменной в памяти. Имена переменных должны начинаться с буквы, за которой следуют буквы, цифры или некоторые специальные символы. В табл. 1.1 перечислены возможные типы данных.

Таблица 1.1. Типы данных

Тип	Длина	Название
db	8 бит	Байт
dw	16 бит	Слово
dd	32 бита	Двойное слово
dq	64 бита	Учетверенное (двойное длинное) слово

В рассматриваемом здесь примере программы section .data содержит одну переменную msg, символьное имя которой указывает на адрес памяти, по которому размещается 'h', первый байт строки "hello, world", 0. Таким образом, msg указывает на букву 'h', msg+1 указывает на букву 'e' и т. д. Эта перемен-

ная называется строкой (string), которая является непрерывным списком символов. Строка – это «список» или «массив» символов в памяти. В действительности любой непрерывный список значений в памяти может считаться строкой, при этом символы могут быть видимыми или невидимыми для человеческого глаза, а сама строка может иметь смысл для человека или не иметь никакого смысла.

Для удобства добавляется ноль, обозначающий конец строки, видимой для человека. Можно не записывать завершающий ноль на свой страх и риск. Этот завершающий 0 не соответствует значению ASCII 0, это числовой ноль, так что в памяти он представлен восемью нулевыми битами. Если аббревиатура ASCII вызвала у вас недоумение, обратитесь к поиску Google. Знание смысла акронима ASCII весьма важно в программировании. Краткое описание: символы, используемые человеком, представлены в виде специальных (числовых) кодов в компьютерах. Прописная (заглавная) буква A (латиница) имеет код 65, В соответствует код 66 и т. д. Для символа перехода на новую строку определен код 10, а невидимый символ NULL получил код 0. Таким образом, мы завершаем строку символом NULL. Если в командной строке ввести `man ascii`, то Linux выведет таблицу символов и кодов ASCII.

Раздел `section .data` также может содержать константы, т. е. значения, которые невозможно изменить в программе. Константы определяются в следующем формате:

```
<constant name>      equ      <value>
```

Пример:

```
pi equ 3.1416
```

Раздел `section .bss`

Аббревиатура `bss` означает **B**lock **S**tarted by **S**ymbol (блок, начинающийся с символа) и ведет свое происхождение с 1950-х гг., когда этот блок был компонентом языка ассемблера, разработанного для IBM 704. В этот раздел помещаются неинициализированные переменные. Для таких неинициализированных переменных выделяемое пространство памяти объявляется в следующем формате:

```
<variable name>      <type>      <number>
```

В табл. 1.2 перечислены возможные типы данных `bss`.

Таблица 1.2. Типы данных `bss`

Тип	Длина	Название
<code>resb</code>	8 бит	Байт
<code>resw</code>	16 бит	Слово
<code>resd</code>	32 бита	Двойное слово
<code>resq</code>	64 бита	Учетверенное (двойное длинное) слово

Например, следующая инструкция объявляет пространство памяти для массива из 20 двойных слов:

```
dArray resd 20
```

Переменные в разделе `section .bss` не содержат каких-либо значений, значения будут присваиваться им в дальнейшем во время выполнения программы. Блоки памяти для этих переменных резервируются не во время компиляции, а во время выполнения. В последующих примерах будет продемонстрировано практическое использование раздела `section .bss`. Когда программа начинает выполняться, она запрашивает у операционной системы необходимую память, выделяемую переменным из раздела `section .bss` и инициализируемую нулями. Если во время выполнения не существует доступной памяти, достаточной для размещения переменных `.bss`, то программа завершается аварийно.

Раздел `section .txt`

Все действия происходят в разделе `section .txt`. Этот раздел содержит код программы и начинается со следующих инструкций:

```
global main
main:
```

Часть `main:` называется меткой (label). Если метка расположена в строке, где после нее нет других символов, то после слова должно быть записано двоеточие, иначе ассемблер выведет предупреждающее сообщение. А предупреждающие сообщения не следует игнорировать. Если за меткой следуют другие инструкции (в той же строке), то двоеточие не обязательно, но все же лучше выработать полезную привычку завершать все метки символом двоеточия. К тому же это повышает удобство чтения исходного кода.

В рассматриваемом здесь примере исходного кода *hello.asm* после метки `main:` регистры `rdi`, `rsi` и `rax` подготавливаются для вывода сообщения на экран. Более подробно о регистрах вы узнаете в главе 2. Здесь просто выводится строка на экран с использованием системного вызова. То есть мы предлагаем операционной системе выполнить эту работу.

- В регистр `rax` записывается код системного вызова 1, означающий `write` (запись).
- Для записи (размещения) некоторого значения в регистр используется инструкция `mov`. В действительности эта инструкция ничего не перемещает (`move` – перемещение), она создает копию источника (`source`) и записывает эту копию в цель (`destination`). Формат команды `mov`:

```
mov destination, source
```

- Инструкцию `mov` можно использовать следующим образом:
 - ♦ `mov` регистр, непосредственное_значение
 - ♦ `mov` регистр, адрес_памяти
 - ♦ `mov` адрес_памяти, регистр
 - ♦ **недопустимое использование:** `mov` адрес_памяти, адрес_памяти

- В рассматриваемом здесь примере целевое устройство вывода для записи сообщения сохраняется в регистре `rdi`, а значение 1 обозначает устройство стандартного вывода (в данном случае это вывод на экран).
- Адрес (памяти) выводимой строки записывается в регистр `rsi`.
- В регистр `rdx` помещается длина сообщения. Количество символов в строке `hello, world`, не включая кавычки, в которые заключена строка, и завершающий `0`. Если в количество включить завершающий `0`, то программа попытается вывести на экран `NULL`-байт, что не имеет особого смысла.
- Затем выполняется системный вызов `syscall`, и строка `msg` выводится на устройство стандартного вывода. `syscall` – это запрос функциональности, предоставляемой операционной системой.
- Чтобы избежать вывода сообщений об ошибках при завершении выполнения программы, необходим корректный («чистый») выход из нее. Для этого сначала в регистр `rax` записывается код 60, обозначающий `exit` (выход). Код «успешного» выхода (завершения программы) помещается в регистр `rdi`, затем выполняется системный вызов. Программа завершает работу без каких-либо сообщений.

Системные вызовы (system calls) используются для запросов к операционной системе для выполнения некоторых конкретных действий. В каждой операционной системе существует собственный набор параметров системных вызовов, и системные вызовы в Linux отличаются от системных вызовов в Windows или macOS. В этой книге мы будем использовать системные вызовы Linux для версии x64, более подробную информацию о них можно найти здесь: http://blog.rchapman.org/posts/Linux_System_Call_Table_for_x86_64/.

Следует учитывать, что системные вызовы 32-битовой версии отличаются от системных вызовов 64-битовой версии. При чтении исходного кода всегда проверяйте, написан ли код для 32-битовых или для 64-битовых систем.

Перейдите в командную строку операционной системы и найдите файл `hello.lst`. Этот файл сгенерирован во время ассемблирования, но перед связыванием (линковкой), как определено в `makefile`. Откройте `hello.lst` в редакторе – и увидите листинг исходного кода на ассемблере, но в самом левом столбце показаны относительные адреса инструкций кода, а во втором слева столбце содержится код, переведенный в машинный язык (в шестнадцатеричном формате). На рис. 1.4 показано содержимое файла `hello.lst`.

1	1		section .data	
2	2	00000000 68656C6C6F2C20776F-	msg db	"hello, world",0
3	3	00000009 726C6400		
4	4		section .bss	
5	5		section .text	
6	6		global main	
7	7		main:	
8	8	00000000 8801000000	mov	rax, 1 ; 1 = write
9	9	00000005 0F01000000	mov	rdi, 1 ; 1 = to stdout
10	10	0000000A 48BE-	mov	rsi, msg ; string to display in rsi
11	11	0000000C [0000000000000000]		
12	12	00000014 BA0C000000	mov	rdx, 12 ; length of the string, without 0
13	13	00000019 0F05	syscall	; display the string
14	14	00000018 883C000000	mov	rax, 60 ; 60 = exit
15	15	00000020 8F00000000	mov	rdi, 0 ; 0 = success exit code
16	16	00000025 0F05	syscall	; quit

Рис. 1.4. Файл `hello.lst` в редакторе

Здесь можно видеть столбец с номерами строк и следующий столбец шириной 8 знакомест. Этот столбец представляет адреса блоков памяти. Когда ассемблер (программа ассемблирования кода) создает объектный файл, он пока еще не знает, какие адреса памяти будут использоваться. Поэтому ассемблер начинает нумерацию с адреса 0 для различных разделов. Для раздела `section .bss` память не выделяется.

Во втором столбце содержится результат преобразования инструкций ассемблера в шестнадцатеричный (машинный) код. Например, инструкция `mov` гах преобразована в `B8`, а `mov edi` в `BF`. Это шестнадцатеричное представление машинных инструкций. Следует также обратить внимание на преобразование строки `msg` в шестнадцатеричные ASCII-символы. Немного позже вы узнаете больше о шестнадцатеричном формате записи. Первая инструкция должна выполняться, начиная с адреса `00000000`, она занимает пять байт: `B8 01 00 00 00`. Здесь двойные нули нужны для заполнения и выравнивания по адресам памяти. Выравнивание по адресам памяти – это функциональная особенность, используемая ассемблерами и компиляторами для оптимизации кода. Ассемблерам и компиляторам можно передавать разнообразные флаги, чтобы получить наименьший возможный размер выполняемого файла, самый быстрый выполняемый код или объединение этих характеристик. В следующих главах будет рассматриваться оптимизация с целью увеличения скорости выполнения кода.

Следующая инструкция начинается с адреса `00000005` и т. д. Адреса памяти содержат восемь знакомест (т. е. 8 байт), в каждом байте 8 бит. Поэтому адреса имеют длину 64 бита, разумеется, если используется 64-битовый ассемблер. Теперь рассмотрим, как представлена ссылка на строку `msg`. Поскольку реальный адрес памяти `msg` пока еще неизвестен, ссылка на эту строку обозначена как `[0000000000000000]`.

Вероятно, вы согласитесь, что мнемонические символьные коды ассемблера и символьные имена адресов памяти (переменных) немного легче читать и запоминать, чем шестнадцатеричные значения, учитывая существование сотен кодов с разнообразными операндами, каждый из которых приводит к генерации еще большего количества шестнадцатеричных инструкций. Когда компьютеры только еще начинали применяться, программисты использовали только машинный язык, язык программирования первого поколения. Язык ассемблера с мнемоническими кодами, «более простыми для запоминания», – это язык программирования второго поколения.

РЕЗЮМЕ

В этой главе вы узнали:

- об основной структуре программы на ассемблере, с ее различными разделами;
- об использовании памяти с символьными именами для адресов;
- о регистрах;
- об одной из инструкций ассемблера `mov`;
- об использовании системного вызова `syscall`;
- о различиях между ассемблерным и машинным кодами.

Глава 2

Двоичные и шестнадцатеричные числа и регистры

В современных компьютерах бит (bit) является наименьшим фрагментом информации. Бит может иметь одно из значений: 1 или 0. В этой главе рассматривается, как объединяются биты для представления данных, таких как целые числа или значения с плавающей точкой. Десятичное представление чисел, весьма удобное для людей, не является наиболее подходящим для компьютерной обработки. При использовании двоичной (бинарной) системы, в которой возможны только два значения (1 или 0), гораздо более эффективно выполняется работа со степенями 2. Когда мы обсуждаем историю поколений компьютеров, то вспоминаем 8-битовые ЦПУ (центральные процессорные устройства, или просто центральные процессоры) (2^3), 16-битовые ЦПУ (2^4), 32-битовые ЦПУ (2^5) и ныне преобладающие 64-битовые ЦПУ (2^6). Но для людей работа с длинными строками единиц и нулей весьма неудобна и чаще всего даже невозможна. В этой главе будет показано, как преобразовать биты в десятичные или шестнадцатеричные значения, с которыми гораздо проще работать. После этого будут рассматриваться регистры (registers) – области хранения данных, которые помогают процессору выполнять логические и арифметические инструкции.

Краткий вводный курс по двоичным числам

Компьютеры используют двоичные (бинарные) цифры (нули и единицы) для выполнения требуемой работы. Восемь бинарных цифр объединяются и называются байтом (byte). Но двоичные числа слишком длинны для того, чтобы человек работал с ними, не говоря уже о запоминании. Шестнадцатеричные числа (немного) более удобны, и не в последнюю очередь потому, что любой 8-битовый байт можно представить в виде всего лишь двух шестнадцатеричных цифр.

Если необходимо увидеть двоичное, десятичное или шестнадцатеричное значение различных форматов вывода, то нужно воспользоваться инструмен-

тальным средством преобразования. В интернете можно найти огромное количество калькуляторов с функциями преобразования. Ниже перечислено несколько таких калькуляторов, которые наиболее просты в использовании:

- www.binaryconvert.com;
- <https://www.binaryhexconverter.com>;
- <https://babbage.cs.qc.cuny.edu/IEEE-754>.

В табл. 2.1 приведены преобразования чисел с 0 до 15, эти преобразования полезно запомнить.

Таблица 2.1. Числа от 0 до 15 в десятичном, шестнадцатеричном и двоичном форматах

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

Целые числа

Существуют два типа целых чисел: знаковые (signed) и беззнаковые (unsigned). Для знаковых целых чисел крайний левый бит равен 1, если число отрицательное, или 0, если число положительное. Беззнаковые целые числа – это 0 и все положительные целые, для знакового бита место не предусмотрено. Для корректной работы с целочисленной арифметикой отрицательные целые числа используются в так называемом представлении числа в дополнительном двоичном коде (two's complement). Получить двоичное представление отрицательного числа можно следующим образом:

- 1) записать абсолютное значение числа в двоичном виде;
- 2) перевести в дополнительный код (заменить все 1 на 0, а все 0 на 1);
- 3) прибавить 1.

Ниже приведен пример с использованием 16-битовых чисел вместо 64-битовых (чтобы пример выглядел более понятно):

```
decimal number =      17
binary number =    0000    0000    0001    0001
hexadecimal number =  0      0      1      1    = 11
decimal number =      -17
binary number absolute value =    0000    0000    0001    0001
complement =          1111    1111    1110    1110
add 1 =              1111    1111    1110    1111
hexadecimal =                f      f      e      f      = ffef

Verify:   -17      11111111 11101111
add:      +17      00000000 00010001
equals:    0        00000000 00000000
```

Обычно перед шестнадцатеричными числами записывается префикс `0x`, чтобы отличать их от десятичных чисел, например `-17` в шестнадцатеричном представлении выглядит как `0xffef`. Если при просмотре листинга на машинном языке в файле `.lst` вы видите число `0xffef`, то из контекста необходимо определить, является ли это значение знаковым или беззнаковым целым числом. Если это знаковое целое, то оно означает `-17` в десятичном представлении. Если это беззнаковое целое, то оно соответствует десятичному числу `65519`. Разумеется, если это адрес памяти, то число беззнаковое (понятно, почему?). Иногда в ассемблерном коде будут встречаться некоторые другие форматы записи (нотации), такие как `0800h`, которые также представляют шестнадцатеричное число, `10010111b` – двоичное число или `420o` – восьмеричное число. Да, не удивляйтесь, восьмеричные числа также можно использовать. Мы будем применять восьмеричные числа, когда будем писать код для файла ввода/вывода. При необходимости преобразования целых чисел не трудитесь сами, пользуйтесь упомянутыми выше веб-сайтами.

Числа с плавающей точкой

Числа с плавающей точкой записываются в двоичном или шестнадцатеричном формате в соответствии с международным стандартом IEEE-754. Этот процесс еще более сложен, чем для целых чисел. Если хотите узнать об этом более подробно, то рекомендуется начать здесь:

<http://mathcenter.oxford.emory.edu/site/cs170/ieee754/>.

Но и в этом случае при необходимости преобразования чисел с плавающей точкой рекомендуется пользоваться веб-сайтами, перечисленными в предыдущем разделе. Здесь мы не будем углубляться в подробности.

Краткий вводный курс по регистрам

Центральное процессорное устройство (ЦПУ), или просто процессор, – мозг компьютера – выполняет инструкции программы, весьма активно используя регистры и оперативную память и производя математические и логические

операции в этих регистрах и в памяти. Следовательно, важно обладать основными знаниями о регистрах и памяти и о том, как они используются. Здесь приводится краткий обзор регистров, подробности об использовании регистров будут постепенно выясняться в следующих главах. Регистры (registers) – это области (локации), используемые процессором для хранения данных, инструкций или адресов памяти. Количество регистров невелико, но процессор может считывать и записывать их чрезвычайно быстро. Можно считать регистры некоторой разновидностью блокнота для процессора, где он хранит временную информацию. Рекомендуется запомнить правило: если важна скорость, то процессор может получать доступ к регистрам намного быстрее, чем доступ к оперативной памяти.

Не беспокойтесь, если этот раздел не совсем понятен, все постепенно станет ясным, когда мы начнем использовать регистры на практике в следующих главах.

Регистры общего назначения

Существует 16 регистров общего назначения, и каждый из них может использоваться как 64-битовый, 32-битовый, 16-битовый или 8-битовый регистр. В табл. 2.2 перечислены имена всех этих регистров, обозначающие соответствующие размеры. Четыре регистра `rax`, `rbx`, `rcx`, `rdx` могут иметь два типа 8-битовых регистров – нижние 8 бит в нижней половине 16-битового регистра и верхние 8 бит в верхней половине 16-битового регистра.

Таблица 2.2. Имена регистров общего назначения

64-битовый	32-битовый	16-битовый	Нижний 8-битовый	Верхний 8-битовый	Комментарий
<code>rax</code>	<code>eax</code>	<code>ax</code>	<code>al</code>	<code>ah</code>	
<code>rbx</code>	<code>ebx</code>	<code>bx</code>	<code>bl</code>	<code>bh</code>	
<code>rcx</code>	<code>ecx</code>	<code>cx</code>	<code>cl</code>	<code>ch</code>	
<code>rdx</code>	<code>edx</code>	<code>dx</code>	<code>dl</code>	<code>dh</code>	
<code>rsi</code>	<code>esi</code>	<code>si</code>	<code>sil</code>	–	
<code>rdi</code>	<code>edi</code>	<code>di</code>	<code>dil</code>	–	
<code>rbp</code>	<code>ebp</code>	<code>bp</code>	<code>bpl</code>	–	Указатель базы (адреса)
<code>rsp</code>	<code>esp</code>	<code>sp</code>	<code>spl</code>	–	Указатель стека
<code>r8</code>	<code>r8d</code>	<code>r8w</code>	<code>r8b</code>	–	
<code>r9</code>	<code>r9d</code>	<code>r9w</code>	<code>r9b</code>	–	
<code>r10</code>	<code>r10d</code>	<code>r10w</code>	<code>r10b</code>	–	
<code>r11</code>	<code>r11d</code>	<code>r11w</code>	<code>r11b</code>	–	
<code>r12</code>	<code>r12d</code>	<code>r12w</code>	<code>r12b</code>	–	
<code>r13</code>	<code>r13d</code>	<code>r13w</code>	<code>r13b</code>	–	
<code>r14</code>	<code>r14d</code>	<code>r14w</code>	<code>r14b</code>	–	
<code>r15</code>	<code>r15d</code>	<code>r15w</code>	<code>r15b</code>	–	

Несмотря на то что регистры `ebp` и `esp` называются регистрами общего назначения, с ними нужно работать с особой осторожностью, так как они используются процессором во время выполнения программы. Правильное использование регистров `ebp` и `esp` будет подробнее рассмотрено в следующих главах.

Каждый 64-битовый регистр содержит набор из 64 бит, нулей и/или единиц, т. е. 8 байт. При записи числа 60 в регистр `eax` в программе `hello, world` этот регистр содержит следующее значение:

```
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00111100
```

Это двоичное представление числа 60 в 64-битовом регистре.

Каждый 32-битовый регистр – это набор 32 нижних (если считать справа) бит 64-битового регистра. Точно так же 16-битовый регистр и 8-битовый регистр состоят из нижних 16 и нижних 8 бит (соответственно) 64-битового регистра.

Примечание. Запомните: «нижние» биты – это всегда крайние правые биты.

Бит с номером 0 – это крайний правый бит, отсчет начинается справа с индекса 0, а не 1. Таким образом, крайний левый бит 64-битового регистра имеет индекс 63, а не 64.

Если регистр `eax` содержит значение 60, то можно также сказать, что регистр `eax` теперь содержит следующее значение:

```
00000000 00000000 00000000 00111100
```

или что регистр `ax` содержит:

```
00000000 00111100
```

а в регистре `al` хранится значение

```
00111100
```

Регистр счетчика команд (`rip`)

Процессор постоянно отслеживает следующую выполняемую инструкцию, сохраняя ее адрес в регистре `rip`. Значение в регистре `rip` можно изменить на любое на свой страх и риск (я вас предупредил). Более безопасный способ изменения значения `rip` – использование инструкций переходов (`jump`), которые рассматриваются в следующей главе.

Регистр флагов

Существует также регистр флагов (процессора) `rflags`, возможные значения для которого приведены в табл. 2.3. После выполнения очередной инструкции программа может проверить, установлен ли конкретный флаг (например, `ZF=1`), чтобы затем действовать соответственно.

Таблица 2.3. Значения флагов процессора

Имя	Символьное обозначение	Бит	Описание
Carry	CF	0	В предыдущей инструкции был выполнен перенос (разрядов)
Parity	PF	2	Последний байт содержит четное число единиц
Adjust	AF	4	Операции BCD (в двоично-десятичном коде)
Zero	ZF	6	Результат предыдущей инструкции равен нулю
Sign	SF	8	В результате выполнения предыдущей инструкции самый значимый бит равен 1
Direction	DF	10	Направление операций со строками (инкремент или декремент)
Overflow	OF	11	В результате выполнения предыдущей инструкции возникло переполнение

Более подробное описание флагов и их практического использования см. немного позже.

Существует еще один регистр флагов с именем MXCSR, который будет использоваться в одной инструкции со многими потоками данных (SIMD), в соответствующей главе регистр MXCSR будет описан более подробно.

Регистры xmm и ymm

Эти регистры используются для операций с числами с плавающей точкой и в инструкциях со многими потоками данных (SIMD). В дальнейшем регистр xmm и связанный с ним регистр ymm будут активно использоваться, начиная с изучения инструкций обработки чисел с плавающей точкой.

В дополнение ко всем описанным выше регистрам существуют и другие регистры, но мы не будем их использовать в этой книге.

С теорией покончено, наступило время для практической работы.

РЕЗЮМЕ

В этой главе вы узнали, как:

- представить значение в десятичном, двоичном и шестнадцатеричном форматах;
- использовать регистры и флаги.

Глава 3

Анализ программ с помощью отладчика: GDB

В этой главе представлено введение в процесс отладки программ на ассемблере. Отладка – это важный навык, потому что с отладчиком можно исследовать содержимое регистров и оперативной памяти в шестнадцатеричном, двоичном или десятичном представлении. Из предыдущей главы вам уже известно, что процессор интенсивно использует регистры и память, а отладчик позволит вам выполнять инструкции шаг за шагом, наблюдая при этом, как изменяется содержимое регистров, памяти и флагов. Возможно, вы уже наблюдали аварийное завершение своей первой программы на ассемблере с выводом непонятного сообщения типа «Memory Segmentation Fault» (Нарушение сегментации памяти). С помощью отладчика вы сможете постепенно (пошагово) пройти по программе во время ее выполнения и точно определить местонахождение и причину ошибки.

Начало отладки

После ассемблирования и связывания (линковки) программы `hello, world` без ошибок на этом этапе вы получаете выполняемый файл. С помощью инструмента-отладчика можно загрузить выполняемую программу в память компьютера и выполнять ее строку за строкой, наблюдая при этом за содержимым различных регистров и блоков памяти. В настоящее время доступно несколько бесплатных и коммерческих отладчиков. В Linux прародителем всех отладчиков является GDB – программа с интерфейсом командной строки с весьма замысловатыми командами. Но так даже интереснее. В следующих главах будет использоваться SASM, инструментальное средство с графическим пользовательским интерфейсом, основанное на GDB. Но освоение базовых знаний о самом отладчике GDB может оказаться полезным, потому что не вся функциональность GDB доступна в SASM.

В своей дальнейшей карьере программиста на ассемблере вы наверняка будете обращать внимание на разнообразные отладчики с превосходными пользовательскими интерфейсами, причем каждый отладчик ориентиро-

ван на конкретную платформу, как, например, Windows, Mac или Linux. Эти GUI-отладчики помогут отлаживать длинные и сложные программы быстрее и проще по сравнению с отладчиком с интерфейсом командной строки. Но GDB представляет собой всеобъемлющий, однако при этом «самый прямой» путь к освоению отладки в Linux. GDB установлен в большинстве систем разработки на платформе Linux, а если он отсутствует, то его можно легко установить. GDB практически не увеличивает нагрузку на систему. Сейчас мы будем использовать GDB, чтобы познакомить вас с его самыми важными свойствами, а также для более плавного перехода к другим инструментальным средствам в следующих главах. Единственное замечание: GDB выглядит так, как будто предназначен для отладки программ на языках высокого уровня, некоторые его функциональные возможности не помогут при отладке ассемблерных программ.

На первых порах отладка программы с использованием отладчика в командной строке может показаться чрезвычайно трудной. Но не отчаивайтесь, читая эту главу, – продвигаясь все дальше и дальше, вы увидите, как все становится проще.

Чтобы начать отладку программы *hello* в командной строке, перейдите в каталог, где была сохранена эта программа. После промпта в командной строке введите:

```
gdb hello
```

GDB загрузит выполняемый файл *hello* в память и выведет собственное приглашение (промпт) (*gdb*), после чего будет ожидать ввода инструкций. Если ввести следующую команду:

```
list
```

то GDB выведет некоторое количество строк исходного кода из загруженного файла. Введите команду *list* еще раз, и GDB выведет следующие строки и т. д. Для вывода конкретной строки, например самой первой строки исходного кода, введите команду *list 1*. На рис. 3.1 показан пример.

Если вывод на ваш экран отличается от показанного на рис. 3.1 и содержит множество символов %, значит, в вашем случае GDB сконфигурирован для использования синтаксиса ассемблера версии AT&T. Мы будем использовать синтаксис ассемблера версии Intel, который более интуитивно понятен (для нас). Ниже будет показано, как изменить версию ассемблера.

После ввода следующей команды

```
run
```

GDB запустит программу *hello*, выведет строку *hello, world* и вернется к своему промпту (*gdb*). На рис. 3.2 показан результат выполнения команды *run* и вывод на экран.


```

jo@UbuntuDesktop:~/Desktop/linux64/gcc/01 hello $ gdb hello
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 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 "x86_64-linux-gnu".
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 hello...done.
(gdb) list
1      section .data
2          msg db      "hello, world",0
3      section .bss
4      section .text
5          global main
6      main:
7          mov     rax, 1          ; 1 = write
8          mov     rdi, 1          ; 1 = to stdout
9          mov     rsi, msg        ; string to display in rsi
10         mov     rdx, 12         ; length of the string, without 0
(gdb) █

```

Рис. 3.1. Вывод команды list отладчика GDB

```

jo@UbuntuDesktop:~/Desktop/linux64/gcc/01 hello $ gdb hello
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 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 "x86_64-linux-gnu".
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 hello...done.
(gdb) run
Starting program: /home/jo/Desktop/linux64/gcc/01 hello /hello
hello, world[Inferior 1 (process 4698) exited normally]
(gdb) █

```

Рис. 3.2. Вывод команды run отладчика GDB

Для выхода из GDB введите команду quit.

Теперь начнем делать некоторые более интересные вещи с помощью GDB.

Но сначала изменим версию дизассемблера – это необходимо сделать, только если вы видите много символов % в выводе предыдущего примера. Загрузите выполняемый файл *hello* в отладчик GDB, если он еще не загружен.

Введите команду

```
set disassembly-flavor intel
```

Эта команда преобразует дизассемблируемый код в формат, который нам уже знаком. Можно сделать версию Intel форматом ассемблера по умолчанию для отладчика GDB, используя соответствующие параметры настройки в профиле командной оболочки Linux. См. документацию по конкретному дистрибутиву Linux. В Ubuntu 18.04 необходимо в своем домашнем каталоге создать файл *.gdbinit*, содержащий приведенную выше инструкцию *set*. Чтобы ассемблер Intel использовался по умолчанию, необходимо выйти из отладчика и снова войти в него.

Запустите отладчик GDB с файлом *hello*, чтобы начать анализ программы. Как вам уже известно, программа *hello, world* сначала инициализирует некоторые данные в разделах *section .data* и *section .bss*, затем переходит к метке *main*. Именно здесь начинается настоящее выполнение, поэтому начнем исследование с этой метки.

После промпта (gdb) введите команду

```
disassemble main
```

Отладчик GDB возвращает исходный код в более или менее похожем виде. Возвращаемый исходный код не совсем похож на код, написанный изначально. Это слегка удивляет. Что произошло? Ситуация требует анализа.

На рис. 3.3 показан код, который GDB вывел на экран в ответ на команду дизассемблирования.

```
(gdb) disassemble main
Dump of assembler code for function main:
0x0000000004004e0 <+0>:      mov     eax,0x1
0x0000000004004e5 <+5>:      mov     edi,0x1
0x0000000004004ea <+10>:     movabs  rsi,0x601030
0x0000000004004f4 <+20>:     mov     edx,0xc
0x0000000004004f9 <+25>:     syscall
0x0000000004004fb <+27>:     mov     eax,0x3c
0x000000000400500 <+32>:     mov     edi,0x0
0x000000000400505 <+37>:     syscall
0x000000000400507 <+39>:     nop     WORD PTR [rax+rax*1+0x0]
End of assembler dump.
(gdb) █
```

Рис. 3.3. Вывод результата дизассемблирования отладчика GDB

Длинные числа слева, начинающиеся с *0x00...*, – это адреса памяти, т. е. локаций, в которых сохранены машинные инструкции программы. Здесь можно видеть по адресам и по *<+5>* во второй строке, что для первой инструкции *mov eax, 0x1* необходимо пять байт памяти. Но в нашем исходном коде была записана инструкция *mov rax, 1*. Почему появился регистр *eax*?

Если вернуться к таблице регистров в главе 2 (табл. 2.2), то мы увидим, что *eax* является нижней 32-битовой частью регистра *rax*. Ассемблер достаточно умен, для того чтобы понять, что 64-битовый регистр – это излишний расход

ресурса для хранения числа 1, поэтому использует 32-битовый регистр. То же самое относится к использованию регистров `edi` и `edx` вместо `rdi` и `rdx`.

64-битовый ассемблер – это расширение 32-битового ассемблера, поэтому в дальнейшем вы увидите, что во всех возможных случаях ассемблер будет использовать 32-битовые инструкции.

Значение `0x1` – это шестнадцатеричное представление десятичного числа 1, `0xd` – представление десятичного числа 13, а `0x3c` соответствует десятичному числу 60.

Инструкция `por` означает «no operation» (отсутствие операции), ассемблер вставил ее здесь по причине, связанной с управлением памятью.

А что произошло со строкой `msg`? Инструкция `mov rsi, msg` заменена на инструкцию `movabs rsi, 0x601030`. Пока не обращайтесь особого внимания на `movabs` – эта команда появилась здесь из-за 64-битовой адресации, она применяется для прямой передачи (значения) в регистр. Число `0x601030` – это адрес памяти, по которому сохранена строка `msg` в конкретном компьютере. В вашем случае это может быть совершенно другой адрес.

После промпта (`gdb`) введите команду

```
x/s 0x601030 (или x/s 'адрес_памяти_в_вашем_случае')
```

GDB отвечает, как показано на рис. 3.4.

```
(gdb) x/s 0x601030
0x601030 <msg>: "hello, world"
(gdb) █
```

Рис. 3.4. Вывод отладчика GDB

Команда `x` означает `examine` (обследовать), ключ `s` – `string` (строка). GDB отвечает, что `0x601030` является начальным адресом строки `msg`, и пытается вывести всю строку в целом до завершающего ее `0`. Теперь вам известна одна из причин, по которой после строки `hello, world` записывается завершающий `0`.

Кроме того, можно ввести команду

```
x/c 0x601030
```

Вывод этой команды показан на рис. 3.5.

```
(gdb) x/c 0x601030
0x601030 <msg>: 104 'h'
(gdb) █
```

Рис. 3.5. Вывод отладчика GDB

Ключ `c` позволяет вывести один символ. Здесь GDB возвращает первый символ строки `msg`, перед которым выводится его десятичный код ASCII. Найдите с помощью Google таблицу кодов ASCII, чтобы проверить правильность выведенного кода, и сохраните таблицу – она будет полезна в будущем, а запоминать ее нет никакой необходимости. Или откройте еще одно окно терминала и в командной строке введите `man ascii`.

Рассмотрим несколько других примеров.

Следующая команда выводит 13 символов, начиная с заданного адреса памяти (см. рис. 3.6):

```
x/13c 0x601030
```

```
(gdb) x/13c 0x601030
0x601030 <msg>: 104 'h' 101 'e' 108 'l' 108 'l' 111 'o' 44 ',' 32 ' ' 119 'w'
0x601038:      111 'o' 114 'r' 108 'l' 100 'd' 0 '\000'
(gdb) █
```

Рис. 3.6. Вывод отладчика GDB

А эта команда выводит 13 символов в десятичном представлении, начиная с заданного адреса памяти (см. рис. 3.7):

```
x/13d 0x601030
```

```
(gdb) x/13d 0x601030
0x601030 <msg>: 104      101      108      108      111      44      32      119
0x601038:      111      114      108      100      0
(gdb) █
```

Рис. 3.7. Вывод отладчика GDB

Еще одна команда позволяет получить шестнадцатеричное представление тех же 13 символов (см. рис. 3.8):

```
x/13x 0x601030
```

```
(gdb) x/13x 0x601030
0x601030 <msg>: 0x68      0x65      0x6c      0x6c      0x6f      0x2c      0x20      0x77
0x601038:      0x6f      0x72      0x6c      0x64      0x00
(gdb) █
```

Рис. 3.8. Вывод отладчика GDB

Строку можно вывести и по ссылке на ее имя `msg` (см. рис. 3.9):

```
x/s &msg
```

```
(gdb) x/s &msg
0x601030 <msg>: "hello, world"
(gdb) █
```

Рис. 3.9. Вывод отладчика GDB

Вернемся к дизассемблированному листингу. Введите команду

```
x/2x 0x004004e0
```

Выводится шестнадцатеричное представление содержимого по двум адресам памяти, начиная с `0x004004e0` (см. рис. 3.10).

```
(gdb) x/2x 0x004004e0
0x4004e0 <main>:      0xb8      0x01
(gdb) █
```

Рис. 3.10. Вывод отладчика GDB

Это первая инструкция программы `mov eax, 0x1` на машинном языке. Это представление инструкции мы видели при просмотре файла *hello.lst*.

ДВИГАЕМСЯ ДАЛЬШЕ

Сделаем еще один шаг по коду программы с помощью отладчика. Если программа не загружена в отладчик, то еще раз загрузите ее.

Сначала мы прерываем работу программы, приостанавливая выполнение, что позволит обследовать ее некоторые внутренние элементы. Введите команду

```
break main
```

В ответ GDB выводит информацию, показанную на рис. 3.11.

```
(gdb) break main
Breakpoint 1 at 0x4004e0: file hello.asm, line 7.
(gdb) █
```

Рис. 3.11. Вывод отладчика GDB

Введите следующую команду:

```
run
```

На рис. 3.12 показан вывод результата.

```
(gdb) run
Starting program: /home/jo/Desktop/linux64/gcc/01 hello /hello

Breakpoint 1, main () at hello.asm:8
8      mov     rax, 1          ; 1 = write
(gdb) █
```

Рис. 3.12. Вывод отладчика GDB

Отладчик останавливается в заданной точке останова и показывает следующую инструкцию, которая должна быть выполнена. Таким образом, инструкция `mov rax, 1` пока еще не выполнена.

Введите команду

```
info registers
```

GDB выводит информацию, показанную на рис. 3.13.


```
(gdb) info registers
rax          0x4004e0 4195552
rbx          0x0      0
rcx          0x0      0
rdx          0x7fffffffddd8 140737488346584
rsi          0x7fffffffddc8 140737488346568
rdi          0x1      1
rbp          0x400510 0x400510 <__libc_csu_init>
rsp          0x7fffffffddce8 0x7fffffffddce8
r8           0x400580 4195712
r9           0x7ffff7de7ab0 140737351940784
r10          0x846     2118
r11          0x7ffff7a2d740 140737348032320
r12          0x4003e0 4195296
r13          0x7fffffffddc0 140737488346560
r14          0x0      0
r15          0x0      0
rip          0x4004e0 0x4004e0 <main>
eflags       0x246     [ PF ZF IF ]
cs           0x33     51
ss           0x2b     43
ds           0x0      0
es           0x0      0
fs           0x0      0
---Type <return> to continue, or q <return> to quit---
```

Рис. 3.13. Отладчик GDB вывел информацию о регистрах

Сейчас содержимое регистров для нас не важно, за исключением регистра `rip`, счетчика (указателя) инструкций. В регистре `rip` содержится значение `0x4004e0` – это адрес памяти следующей инструкции, которая должна быть выполнена. Проверьте листинг дизассемблированного кода: адрес `0x4004e0` (в рассматриваемом здесь примере) указывает на первую инструкцию `mov %ax, 1`. GDB останавливается прямо перед этой инструкцией и ожидает ввода команд пользователя. Очень важно помнить, что инструкция, на которую указывает регистр `rip`, пока еще не выполнена.

В вашем случае GDB может вывести адрес, отличающийся от `0x4004e0`. Это нормально, так как это адрес конкретной строки в памяти, который может быть различным в конкретной конфигурации компьютера.

Чтобы продвинуться на один шаг, введите команду

```
step
```

Для вывода содержимого регистров можно пользоваться сокращенной версией команды `info registers`:

```
i r
```

На рис. 3.14 показан вывод этой команды.

```

(gdb) step
9      mov     rdi, 1                ; 1 = to stdout
(gdb) i r
rax            0x1      1
rbx            0x0      0
rcx            0x0      0
rdx            0x7fffffffddd8  140737488346584
rsi            0x7fffffffddc8  140737488346568
rdi            0x1      1
rbp            0x400510 0x400510 <__libc_csu_init>
rsp            0x7fffffffddce8 0x7fffffffddce8
r8             0x400580 4195712
r9             0x7ffff7de7ab0 140737351940784
r10            0x846     2118
r11            0x7ffff7a2d740 140737348032320
r12            0x4003e0 4195296
r13            0x7fffffffddc0 140737488346560
r14            0x0      0
r15            0x0      0
rip            0x4004e5 0x4004e5 <main+5>
eflags         0x246     [ PF ZF IF ]
cs             0x33     51
ss             0x2b     43
ds             0x0      0
es             0x0      0
fs             0x0      0
gs             0x0      0
(gdb)

```

Рис. 3.14. Отладчик GDB вывел информацию о регистрах

Разумеется, теперь регистр `rax` содержит значение `0x1`, а в `rip` находится адрес следующей выполняемой инструкции.

Самостоятельно пройдите дальше по программе и обратите внимание на то, как регистр `rsi` принимает адрес строки `msg`, как выводится на экран строка `hello, world` и происходит выход из программы. Еще раз следует отметить, что при каждом шаге регистр `rip` указывает на следующую выполняемую инструкцию.

Некоторые дополнительные команды отладчика GDB

Команда `break` или `b` устанавливает точку останова (временного прекращения выполнения программы), как мы наблюдали в предыдущем разделе.

```

disable breakpoint число
enable breakpoint  число
delete breakpoint  число

```

Команда `continue` или `c` продолжает выполнение до следующей точки останова.

Команда `step` или `s` – шаг в текущую строку, в итоге выполняется переход в вызываемую функцию.

Команда `next` или `n` – шаг через текущую строку (из текущей строки) и останов перед следующей строкой.

Команда `help` или `h` – вывод справочной информации.

Команда `tui enable` – разрешение использования простого текстового пользовательского интерфейса. Для отключения этого интерфейса введите команду `tui disable`.

Команда `print` или `p` – вывод значения переменной, регистра и т. д.

Несколько примеров:

- вывод содержимого регистра `rax`: `p $rax`;
- вывод содержимого регистра `rax` в двоичном формате: `p/t $rax`;
- вывод содержимого регистра `rax` в шестнадцатеричном формате: `p/x $rax`.

Еще одно важное замечание, касающееся отладчика GDB: для его правильного использования необходимо обязательно вставить в исходный код пролог функции (function prologue) и эпилог функции (function epilogue). В следующей главе будет показано, как это сделать, а в очередной главе пролог и эпилог функции будут рассматриваться при обсуждении кадров стека. Для коротких программ, таких как `hello, world`, проблемы не возникают. Но в длинных программах GDB будет демонстрировать непредвиденное поведение при отсутствии пролога и эпилога.

Поэкспериментируйте с отладчиком GDB, ознакомьтесь с онлайнным руководством (в командной строке введите `man gdb`), в общем, осваивайте GDB в полной мере, потому что даже если вы используете отладчик с графическим пользовательским интерфейсом, некоторые функциональные возможности могут оказаться недоступными. Или возможна ситуация, когда в системе вообще не установлен отладчик с GUI.

Немного улучшенная версия программы hello, world

Вероятно, вы заметили, что после вывода фразы `hello, world` промпт появляется в той же строке. Необходимо, чтобы фраза `hello, world` выводилась на отдельной строке, а командный промпт появлялся в новой строке.

В листинге 3.1 приведен код, позволяющий сделать это.

Листинг 3.1. Улучшенная версия программы `hello, world`

```
;hello2.asm
section .data
    msg     db     "hello, world",0
    NL      db     0xa ; Код ASCII для символа перехода на новую строку.
section .bss
section .text
    global main
main:
    mov     rax, 1      ; 1 = запись
    mov     rdi, 1      ; 1 = в устройство стандартного вывода stdout.
    mov     rsi, msg    ; Выводимая строка.
    mov     rdx, 12     ; Длина строки без завершающего 0.
    syscall           ; Вывод строки.
    mov     rax, 1      ; 1 = запись
    mov     rdi, 1      ; 1 = в устройство стандартного вывода stdout.
```



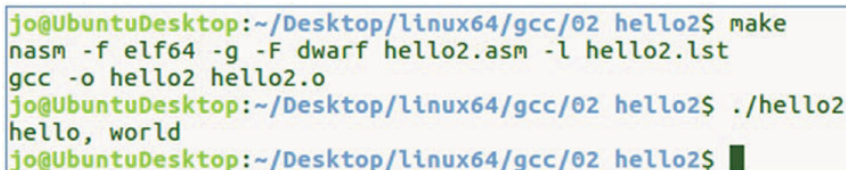
```

mov    rsi, NL      ; Вывод символа перехода на новую строку.
mov    rdx, 1       ; Длина строки.
syscall                ; Вывод строки.
mov    rax, 60      ; 60 = выход.
mov    rdi, 0       ; 0 = код успешного завершения программы.
syscall                ; Выход.

```

Введите этот код в текстовом редакторе и сохраните его в файле *hello2.asm* в новом каталоге. Скопируйте ранее созданный *makefile* в этот новый каталог, затем в этой копии *makefile* замените все имена файлов *hello* на *hello2* и сохраните файл.

В программу добавлена переменная *NL*, содержащая шестнадцатеричное значение *0xa* – код ASCII для символа перехода на новую строку, и инструкции вывода этой переменной *NL* сразу после вывода строки *msg*. Ассемблируйте этот код и выполните программу (результат см. на рис. 3.15).



```

jo@UbuntuDesktop:~/Desktop/linux64/gcc/02 hello2$ make
nasm -f elf64 -g -F dwarf hello2.asm -l hello2.lst
gcc -o hello2 hello2.o
jo@UbuntuDesktop:~/Desktop/linux64/gcc/02 hello2$ ./hello2
hello, world
jo@UbuntuDesktop:~/Desktop/linux64/gcc/02 hello2$ █

```

Рис. 3.15. Результат выполнения улучшенной версии программы *hello, world*

Другой способ выполнения этой задачи – изменение самой строки *msg*, как показано ниже:

```
msg    db    "hello, world",10,0
```

Здесь *10* – это десятичное представление кода символа перехода на новую строку (шестнадцатеричного кода *0xa*). Попробуйте и этот вариант. Но не забудьте увеличить значение для регистра *rdx* до *13*, чтобы включить в строку дополнительный символ с кодом *10*.

В листинге 3.2 показан код этой версии. Сохраните код в файле *hello3.asm* в отдельном каталоге, скопируйте и измените соответствующим образом файл *makefile*, затем выполните сборку и запуск программы.

Листинг 3.2. Еще одна версия программы *hello, world*

```

;hello3.asm
section .data
    msg    db    "hello, world",10,0
section .bss
section .text
    global main
main:
    mov    rax, 1      ; 1 = запись
    mov    rdi, 1      ; 1 = в устройство стандартного вывода stdout.
    mov    rsi, msg    ; Выводимая строка.
    mov    rdx, 13     ; Длина строки без завершающего 0.
    syscall            ; Вывод строки.

```

```
mov    rax, 60      ; 60 = выход.  
mov    rdi, 0       ; 0 = код успешного завершения программы.  
syscall            ; Выход.
```

Но использование этой версии означает, что символ перехода на новую строку является частью выводимой строки, а это не всегда приемлемо, потому что переход на новую строку – это инструкция форматирования, которая, вероятнее всего, предназначена только для вывода строки, но не для выполнения других операций обработки строк. С другой стороны, исходный код становится короче и проще. Выбор за вами.

РЕЗЮМЕ

В этой главе вы узнали, как:

- использовать GDB – отладчик с текстовым интерфейсом командной строки;
- вывести символ перехода на новую строку.

Глава 4

Следующая программа: Alive and Kicking

После того как вы овладели основными навыками отладки с помощью GDB и узнали, что представляет собой программа на ассемблере, попробуем добавить сложности. В этой главе будет показано, как получить длину строковой переменной, а также как вывести значения целого числа и числа с плавающей точкой, используя функцию `printf`. Кроме того, мы немного расширим знания о командах отладчика GDB.

Листинг 4.1 содержит пример исходного кода, который будет использоваться для демонстрации того, как можно определить длину строки и как числовые значения хранятся в памяти.

Листинг 4.1. Исходный код программы *alive.asm*

```
;alive.asm
section .data
    msg1 db "Hello, World!",10,0 ; Строка с NL и 0.
    msg1Len equ $-msg1-1 ; Мера длины без 0.
    msg2 db "Alive and Kicking!",10,0 ; Строка с NL и 0.
    msg2Len equ $-msg2-1 ; Мера длины без 0
    radius dq 357 ; Это не строка, можно ли вывести?
    pi dq 3.14 ; Это не строка, можно ли вывести?
section .bss
section .text
    global main
main:
    push rbp ; Пролог функции.
    mov rbp, rsp ; Пролог функции.
    mov rax, 1 ; 1 = запись
    mov rdi, 1 ; 1 = в устройство стандартного вывода stdout.
    mov rsi, msg1 ; Выводимая строка.
    mov rdx, msg1Len ; Длина строки.
    syscall ; Вывод строки.
    mov rax, 1 ; 1 = запись
    mov rdi, 1 ; 1 = в устройство стандартного вывода stdout.
    mov rsi, msg2 ; Вывод строки.
    mov rdx, msg2Len ; Длина строки.
    syscall ; Вывод строки.
    mov rsp, rbp ; Эпилог функции.
```

```

pop          rbp          ; Эпилог функции.
mov          rax, 60      ; 60 = выход
mov          rdi, 0       ; 0 = код успешного завершения.
syscall

```

Введите этот исходный код в текстовом редакторе и сохраните его в файле *alive.asm*. Создайте файл *makefile*, содержащий строки, показанные в листинге 4.2.

Листинг 4.2. *makefile* для программы *alive.asm*

```

#makefile для alive.asm
alive: alive.o
    gcc -o alive alive.o -no-pie
alive.o: alive.asm
    nasm -f elf64 -g -F dwarf alive.asm -l alive.lst

```

Сохраните этот файл и выходите из текстового редактора.

После промпта командной строки введите команду `make`, чтобы ассемблировать и создать выполняемый файл, затем запустите его из командной строки: `./alive`. Если вы увидите вывод, показанный на рис. 4.1, то все работает, как предполагалось, в противном случае вы где-то допустили опечатку или ошибку. Начинайте отладку.



```

jo@UbuntuDesktop:~/Desktop/linux64/gcc/04 alive$ make
nasm -f elf64 -g -F dwarf alive.asm -l alive.lst
gcc -o alive alive.o -ggdb -no-pie
jo@UbuntuDesktop:~/Desktop/linux64/gcc/04 alive$ ./alive
Hello, World!
Alive and Kicking!
jo@UbuntuDesktop:~/Desktop/linux64/gcc/04 alive$

```

Рис. 4.1. Вывод программы *alive.asm*

АНАЛИЗ ПРОГРАММЫ ALIVE

В первой программе *hello.asm* мы поместили длину строки `msg`, равную 13 символам, в регистр `rdi`, чтобы вывести строку `msg`. В исходном коде *alive.asm* используется весьма удобная функциональная возможность вычисления длины переменных, как показано ниже:

```
msg1Len equ $-msg1-1
```

Часть инструкции `$-msg1-1` означает следующее: взять эту локацию (адрес) памяти (`$`) и вычесть адрес памяти, где находится строка `msg1`. Результатом является длина строки `msg1`. Эта длина -1 (вычитается позиция нуля, завершающего строку) сохраняется в константе `msg1Len`.

Обратите внимание на использование пролога и эпилога функции в этом коде. Это необходимо для обеспечения корректной работы отладчика GDB, о чем было сказано в предыдущей главе. Код пролога и эпилога будет объяснен в следующей главе.

Приступим к исследованию оперативной памяти с использованием GDB. Введите команду

```
gdb alive
```

Далее после промпта (gdb) введите команду

```
disassemble main
```

На рис. 4.2 показан вывод этой команды.

```
(gdb) disass main
Dump of assembler code for function main:
0x000000004004e0 <+0>:    push    rbp
0x000000004004e1 <+1>:    mov     rbp, rsp
0x000000004004e4 <+4>:    mov     eax, 0x1
0x000000004004e9 <+9>:    mov     edi, 0x1
0x000000004004ee <+14>:   movabs  rsi, 0x601030
0x000000004004f8 <+24>:   mov     edx, 0xe
0x000000004004fd <+29>:   syscall
0x000000004004ff <+31>:   mov     eax, 0x1
0x00000000400504 <+36>:   mov     edi, 0x1
0x00000000400509 <+41>:   movabs  rsi, 0x60103f
0x00000000400513 <+51>:   mov     edx, 0x13
0x00000000400518 <+56>:   syscall
0x0000000040051a <+58>:   mov     rsp, rbp
0x0000000040051d <+61>:   pop     rbp
0x0000000040051e <+62>:   mov     eax, 0x3c
0x00000000400523 <+67>:   mov     edi, 0x0
0x00000000400528 <+72>:   syscall
0x0000000040052a <+74>:   nop     WORD PTR [rax+rax*1+0x0]
End of assembler dump.
(gdb) █
```

Рис. 4.2. Дизассемблированный код программы *alive*

Здесь видно, что на моем компьютере переменная `msg1` расположена в памяти по адресу `0x601030`, это можно проверить следующей командой:

```
x/s 0x601030
```

Вывод показан на рис. 4.3.

```
(gdb) x/s 0x601030
0x601030 <msg1>:      "Hello, World!\n"
(gdb) █
```

Рис. 4.3. Адрес места расположения в памяти строки `msg1`

Пара символов `\n` обозначает «переход на новую строку». Другой способ проверки переменных в GDB:

```
x/s &msg1
```

Вывод показан на рис. 4.4.

```
(gdb) x/s &msg1
0x601030 <msg1>:      "Hello, World!\n"
(gdb) █
```

Рис. 4.4. Адрес места расположения в памяти строки msg1

А как насчет числовых значений?

```
x/dw      &radius
x/xw      &radius
```

Вывод показан на рис. 4.5.

```
(gdb) x/dw &radius
0x601053 <radius>:      357
(gdb) x/xw &radius
0x601053 <radius>:      0x00000165
(gdb) █
```

Рис. 4.5. Вывод числовых значений

Таким образом, мы получили десятичное и шестнадцатеричное представления значения, хранящегося в локации памяти radius. Для переменных, содержащих числа с плавающей точкой, используются следующие команды:

```
x/fg &pi
x/fx &pi
```

Вывод показан на рис. 4.6.

```
(gdb) x/fg &pi
0x60105b <pi>:      3.1400000000000001
(gdb) x/fx &pi
0x60105b <pi>:      0x40091eb851eb851f
(gdb) █
```

Рис. 4.6. Вывод значений с плавающей точкой

(Вы обратили внимание на ошибочное значение числа с плавающей точкой?)

Здесь существует одна тонкость, о которой всегда следует помнить. Для ее наглядной демонстрации откроем только что сгенерированный файл листинга alive.lst. См. рис. 4.7.

```

1 1 ; alive.asm
2 2 section .data
3 3 00000000 48656C6C6F2C20576F- msg1 db "Hello, World!",10,0 ; string with NL and 0
4 4 00000009 726C64210A00
5 5
6 6 0000000F 416C69766520616E64- msg1len equ $-msg1-1 ; measure the length, minus the 0
7 7 00000018 2048696368696E6721- msg2 db "Alive and Kicking!",10,0 ; string with NL and 0
8 8 00000021 0A00
9 9
10 10 00000023 6501000000000000 msg2len equ $-msg2-1 ; measure the length, minus the 0
11 11 0000002B 1F85EB51B81E0940 radius dq 357 ; no string, not displayable?
12 12 9 pi dq 3.14 ; no string, not displayable?
13 13
14 14 section .bss
15 15 global main
16 16 main:
17 17 00000000 55 push rbp ; function prologue
18 18 00000001 4889E5 mov rbp,rbp ; function prologue
19 19 00000004 B801000000 mov rax, 1 ; 1 = write
20 20 00000009 BF01000000 mov rdi, 1 ; 1 = to stdout
21 21 0000000E 48BE- mov rsi, msg1 ; string to display
22 22 00000010 [0000000000000000]
23 23 00000018 BA0E000000 mov rdx, msg1len ; length of the string
24 24 0000001D 0F05 syscall ; display the string
25 25 0000001F B801000000 mov rax, 1 ; 1 = write
26 26 00000024 BF01000000 mov rdi, 1 ; 1 = to stdout
27 27 00000029 48BE- mov rsi, msg2 ; string to display
28 28 0000002B [0F00000000000000]
29 29 00000033 BA13000000 mov rdx, msg2len ; length of the string
30 30 00000038 0F05 syscall ; display the string
31 31 0000003A 4889EC mov rsp,rbp ; function epilogue
32 32 0000003D 5D pop rbp ; function epilogue
33 33 0000003E B83C000000 mov rax, 60 ; 60 = exit
34 34 00000043 BF00000000 mov rdi, 0 ; 0 = success exit code
35 35 00000048 0F05 syscall ; quit

```

Рис. 4.7. Содержимое файла листинга *alive.lst*

Взгляните на строки 10 и 11, в которых слева можно видеть шестнадцатеричное представление значений `radius` и `pi`. Вместо 0165 записано 6501, а вместо 40091EB851EB851F мы видим 1F85EB51B81E0940. Получается, что байты (1 байт содержит два шестнадцатеричных числа) хранятся в обратном порядке.

Эта особенность называется порядком следования байтов (*endianness*). В формате с обратным порядком байтов (от старшего к младшему – *big endian*) числа хранятся в том виде, в котором мы привыкли их видеть, т. е. старшие разряды (наиболее значимые цифры) начинаются слева. В формате с прямым порядком байтов (от младшего к старшему – *little endian*) младшие разряды (наименее значимые цифры) начинаются слева. Процессоры Intel используют формат с прямым порядком байтов (*little endian*), и это может приводить к весьма существенным затруднениям при чтении кода в шестнадцатеричном представлении.

Но почему для обозначения порядка байтов используются такие странные английские термины: *big endian* и *little endian*?

В 1726 г. Джонатан Свифт написал свой знаменитый роман «Путешествия Гулливера». В одной из частей этого романа описаны два вымышленных острова: Лилипутия и Блефуску. Обитатели Лилипутии воюют с народом Блефуску из-за разногласий в способе разбивания вареных яиц: с острого или с тупого конца. Лилипуты предпочитают разбивать яйцо с острого конца – они остроконечники (*little endians*). Жители Блефуску – тупоконечники (*big endians*). Характерный пример того, как современные информационные технологии отдают дань традициям, корни которых уходят в далекое прошлое.

Выделите немного времени для пошагового прохода по этой программе (`break main, run, next, next, next...`). Вы увидите, как GDB пошагово проходит по прологу функции. Отредактируйте исходный код, удалив пролог и эпилог функции, потом снова соберите программу командой `make`. После этого попробуйте еще раз пошагово пройти по программе с помощью GDB. В этом

случае GDB отказывается от пошагового прохода и сразу выполняет программу полностью. При ассемблировании с использованием YASM, другой программы ассемблирования на основе NASM, можно безопасно пропустить код пролога и эпилога и выполнять пошаговый проход с помощью GDB. Иногда это необходимо для экспериментов, дерзайте, и Google вам в помощь.

Вывод

Рассматриваемая здесь программа *alive* выводит две строки:

```
Hello, World!
Alive and Kicking!
```

Но в программе существуют еще две переменные, которые не определены как строки: *radius* и *pi*. Вывод значений этих переменных немного сложнее, чем вывод строк. Чтобы вывести числа, хранящиеся в этих переменных, таким же способом, как это сделано для *msg1* и *msg2*, необходимо преобразовать значения *radius* и *pi* в строки. Вполне возможно добавить код для такого преобразования в нашу программу, но тогда она станет слишком сложной, а это весьма нежелательно в настоящий момент, поэтому применим небольшую хитрость. Мы позаимствуем хорошо известную функцию *printf* из языка программирования C и включим ее в свою программу. Если такой подход разочаровывает вас, потерпите немного. Когда вы станете более опытным программистом на ассемблере, то сможете писать собственные функции преобразования и вывода чисел. Или, возможно, придете к выводу, что при написании собственной функции *printf* затрачивается слишком много времени...

Для демонстрации использования функции *printf* в ассемблерном коде начнем с простой программы. Изменим первую программу *hello.asm*, как показано в листинге 4.3.

Листинг 4.3. *hello4.asm*

```
; hello4.asm
extern    printf    ; Объявление функции как внешней.
section .data
    msg    db    "Hello, World!";0
    fmtstr db    "This is our string: %s",10,0 ; Формат вывода строки.
section .bss
section .text
    global main
main:
    push    rbp
    mov     rbp, rsp
    mov     rdi, fmtstr    ; Первый аргумент для функции printf.
    mov     rsi, msg        ; Второй аргумент для функции printf.
    mov     rax, 0          ; Регистры xmm не применяются.
    call    printf          ; Вызов (внешней) функции.
    mov     rsp, rbp
    pop     rbp
    mov     rax, 60          ; 60 = выход.
    mov     rdi, 0          ; 0 = код успешного завершения.
    syscall                ; Выход.
```


Здесь мы начали с сообщения ассемблеру (и линкеру) о том, что будет использоваться внешняя функция с именем `printf`. Создается строка, определяющая, как `printf` будет выводить `msg`. Синтаксис строки формата похож на синтаксис языка C. Если у вас есть опыт программирования на C, то вы, разумеется, узнали строку формата: `%s` – это шаблон для подстановки выводимой строки.

Не забывайте о прологе и эпилоге функции. Адрес `msg` помещается в регистр `rsi`, адрес `fmtstr` – в регистр `rdi`. Далее обнуляется регистр `rax`, в данном случае это означает, что в `xmm`-регистрах `registers` нет чисел с плавающей точкой для вывода. Работа с числами с плавающей точкой и с `xmm`-регистрами `registers` будет описана в главе 11.

В листинге 4.4 показано содержимое *makefile*.

Листинг 4.4. *makefile* для программы *hello4.asm*

```
#makefile для hello4.asm
hello4: hello4.o
    gcc -o hello4 hello4.o -no-pie
hello4.o: hello4.asm
    nasm -f elf64 -g -F dwarf hello4.asm -l hello4.lst
```

Необходимо убедиться, что флаг `-no-pie` добавлен в *makefile*, иначе использование функции `printf` приведет к ошибке. В главе 1 было отмечено, что современная версия компилятора `gcc` генерирует позиционно независимый выполняемый (`pie`) код для улучшения его защиты от взлома. Одним из последствий такой методики компиляции является невозможность простого использования внешних функций. Для устранения этого затруднения используется флаг `-no-pie`.

Выполните сборку и запустите программу. Найдите информацию о функции `printf`, чтобы получить представление о возможных форматах. Вы узнаете, что `printf` предоставляет возможности гибкого форматирования при выводе целых чисел, чисел с плавающей точкой, строк, данных в шестнадцатеричном виде и т. д. Функция `printf` требует, чтобы строка обязательно завершалась `0` (`NULL`). Если `0` не указан, то `printf` будет выводить все подряд, пока не обнаружит `0`. Завершение строки нулем не является требованием ассемблера, это необходимо для `printf`, `GDB`, а также для некоторых инструкций `SIMD` (инструкции `SIMD` будут рассматриваться в главе 26).

На рис. 4.8 показан вывод программы.

```
jo@UbuntuDesktop:~/Desktop/linux64/gcc/05 hello4$ make
nasm -f elf64 -g -F dwarf hello4.asm -l hello4.lst
gcc -o hello4 hello4.o
jo@UbuntuDesktop:~/Desktop/linux64/gcc/05 hello4$ ./hello4
This is our string: Hello, World!
jo@UbuntuDesktop:~/Desktop/linux64/gcc/05 hello4$ █
```

Рис. 4.8. Вывод программы *hello4*

Но вернемся к программе *alive*. Теперь с помощью функции `printf` можно вывести значения переменных `radius` и `pi`. В листинге 4.5 показан исходный код. Теперь вы знаете, что делать: создать файл исходного кода, скопировать или создать/отредактировать *makefile*, в общем, все как раньше.

Листинг 4.5. *makefile* для программы *alive2.asm*

```

; alive2.asm
section .data
    msg1      db    "Hello, World!",0
    msg2      db    "Alive and Kicking!",0
    radius     dd    357
    pi        dq    3.14
    fmtstr     db    "%s",10,0      ; Формат для вывода строки.
    fmtflt     db    "%lf",10,0     ; Формат для вывода числа с плавающей точкой.
    fmtint     db    "%d",10,0      ; Формат для вывода целого числа.
section .bss
section .text
extern printf
global main
main:
    push rbp
    mov rbp, rsp
; print msg1
    mov rax, 0          ; Без числа с плавающей точкой.
    mov rdi, fmtstr
    mov rsi, msg1
    call printf
; print msg2
    mov rax, 0          ; Без числа с плавающей точкой.
    mov rdi, fmtstr
    mov rsi, msg2
    call printf
; print radius
    mov rax, 0          ; Без числа с плавающей точкой.
    mov rdi, fmtint
    mov rsi, [radius]
    call printf
; print pi
    mov rax, 1          ; Используется 1 регистр xmm.
    movq xmm0, [pi]
    mov rdi, fmtflt
    call printf
    mov rsp, rbp
    pop rbp
ret

```

Здесь добавлены три строки для форматирования вывода. Строка формата помещается в регистр `rdi`, регистр `rsi` указывает на выводимый элемент, запись `0` в регистр `rax` означает, что числа с плавающей точкой не используются. Затем вызывается внешняя функция `printf`. Для вывода числа с плавающей точкой его значение помещается в регистр `xmm0` с помощью специальной инструкции `movq`. Используется только один `xmm`-регистр, поэтому в регистр `rax` записывается `1`. В следующих главах будет рассматриваться более подробно использование регистров `XMM` для вычислений с плавающей точкой и для выполнения инструкций `SIMD`.

Обратите внимание на квадратные скобки `[]`, в которые помещены имена переменных `radius` и `pi`.

```
mov rsi, [radius]
```

Это означает: взять содержимое по адресу `radius` и поместить его в регистр `rsi`. Для функции `printf` необходим адрес памяти выводимой строки, но для вывода чисел ожидается значение, а не адрес памяти. Об этом следует помнить всегда.

Выход из этой программы изменен. Вместо уже знакомого кода

```
mov    rax, 60      ; 60 = выход.
mov    rdi, 0       ; 0 = код успешного завершения.
syscall              ; Выход.
```

используется равнозначная инструкция:

```
ret
```

Важное замечание: функция `printf` принимает строку формата, которая может принимать разнообразные формы и преобразовывать сущность выводимых значений (целые числа, числа с двойной точностью, числа с плавающей точкой и т. д.). Иногда такое преобразование является непреднамеренным и может приводить к путанице. Если вам действительно необходимо знать истинное значение регистра или переменной (области памяти) в программе, то следует воспользоваться отладчиком и обследовать требуемый регистр или область памяти.

На рис. 4.9 показан вывод программы *alive2*.



```
jo@UbuntuDesktop:~/Desktop/linux64/gcc/06 alive2$ make
nasm -f elf64 -g -F dwarf alive2.asm -l alive2.lst
gcc -o alive2 alive2.o -no-pie
jo@UbuntuDesktop:~/Desktop/linux64/gcc/06 alive2$ ./alive2
Hello, World!
Alive and Kicking!
357
3.140000
jo@UbuntuDesktop:~/Desktop/linux64/gcc/06 alive2$ █
```

Рис. 4.9. Вывод программы *alive2*

РЕЗЮМЕ

В этой главе вы узнали:

- о дополнительной функциональности отладчика GDB;
- о прологе и эпилоге функции;
- о прямом и обратном порядках следования байтов;
- об использовании функции `printf` из стандартной библиотеки языка C для вывода строк, целых чисел и чисел с плавающей точкой.

Глава 5

Ассемблер основан на логике

А теперь освежим в памяти теоретические знания о математической логике. Не волнуйтесь – здесь будет рассматриваться только то, что действительно необходимо: логические операторы NOT, OR, XOR и AND.

В этой главе 0 обозначает ложь (false), 1 обозначает истину (true).

ЛОГИЧЕСКИЙ ОПЕРАТОР NOT

Таблица 5.1

A	0	1
NOT A	1	0

Преобразование каждого 0 в 1 и каждой 1 в 0.

Пример:

A = 11001011
NOT A = 00110100

ЛОГИЧЕСКИЙ ОПЕРАТОР OR

Таблица 5.2

A	0	1	0	1
B	0	0	1	1
A OR B	0	1	1	1

Если A или B или и A, и B равны 1, то результат равен 1.

Пример:

A = 11001011
B = 00011000
A OR B = 11011011

ЛОГИЧЕСКИЙ ОПЕРАТОР XOR

Таблица 5.3

A	0	1	0	1
B	0	0	1	1
A XOR B	0	1	1	0

Исключающее ИЛИ: если или A, или B равно 1, то результат равен 1. Если и A, и B одновременно равны 1 или 0, то результат равен 0.

Пример:

```
A =      11001011
B =      00011000
A XOR B = 11010011
```

Логический оператор XOR можно применять как ассемблерную инструкцию для очистки (обнуления) регистра.

```
A =      11001011
A =      11001011
A XOR A = 00000000
```

Таким образом, хог гах, гах – это то же самое, что mov гах, 0. Но инструкция хог выполняется быстрее, чем mov. Также можно использовать хог для изменения знака числа с плавающей точкой.

Пример смены знака для 32-битового числа с плавающей точкой:

```
A      = 17.0  = 0x41880000 = 01000001 10001000 00000000 00000000
B      = -0.0  = 0x80000000 = 10000000 00000000 00000000 00000000
A XOR B = -17.0 = 0xC1880000 = 11000001 10001000 00000000 00000000
```

Для проверки правильности результата этой операции используйте онлайн-новое инструментальное средство, размещенное здесь: www.binaryconvert.com/result_float.html.

Следует отметить, что если необходимо изменить знак целого числа, то нужно просто вычесть его из нуля или воспользоваться инструкцией neg.

ЛОГИЧЕСКИЙ ОПЕРАТОР AND

Таблица 5.4

A	0	1	0	1
B	0	0	1	1
A AND B	0	0	0	1

Если A и B одновременно равны 1, то результат равен 1, во всех прочих случаях результат равен 0.

Пример:

```
A =      11001011
B =      00011000
A AND B = 00001000
```

Инструкция AND может использоваться как маска для выбора и проверки отдельных битов.

В приведенном ниже примере B используется как маска для выбора битов 3 и 6 из A (младший, крайний справа бит имеет индекс 0):

```
A =      11000011
B =      01001000
A AND B = 01000000
```

По результату этой операции можно сделать вывод о том, что бит 6 установлен (равен 1), а бит 3 не установлен (равен 0). Более подробно об этом мы поговорим позже.

Кроме того, инструкцию AND можно использовать для округления чисел с недостатком (в меньшую сторону), а особенно полезна она при округлении адресов с выравниванием по 16-байтовой границе. В дальнейшем мы будем применять инструкцию AND для выравнивания стеков.

Число 16 и числа, кратные 16, в шестнадцатеричном формате всегда оканчиваются 0 или 0000 в двоичном представлении.

```
address = 0x42444213 = 01000010010001000100001000010011
mask =    0xffffffff = 11111111111111111111111111111000
rounded = 0x42444210 = 01000010010001000100001000010000
```

Здесь выполнено округление в меньшую сторону к самому младшему байту адреса. Если адрес уже заканчивается нулевым байтом, то инструкция AND не изменяет ничего. Проверьте, действительно ли округленный адрес делится нацело на 16. Для этого воспользуйтесь онлайн-утилитой преобразования форматов чисел (например, www.binaryconvert.com/convert_unsigned_int.html).

РЕЗЮМЕ

В этой главе вы узнали:

- о логических операторах;
- как использовать логические операторы в качестве инструкций ассемблера.

Глава 6

Отладчик Data Display Debugger

Отладчик Data Display Debugger (DDD) – это инструментальное средство отладки с графическим пользовательским интерфейсом для систем Linux. Необходимо установить его (командой `sudo apt install ddd`), так как отладчик DDD будет использоваться на протяжении всей этой главы. Программа, написанная в этой главе, ничего не выводит, и мы будем наблюдать за выполнением кода и за содержимым регистров с помощью DDD.

РАБОТА С ОТЛАДЧИКОМ DDD

В листинге 6.1 показан пример исходного кода.

Листинг 6.1. Программа *move.asm*

```
; move.asm
section .data
    bNum db 123
    wNum dw 12345
    dNum dd 1234567890
    qNum1 dq 1234567890123456789
    qNum2 dq 123456
    qNum3 dq 3.14
section .bss
section .text
    global main
main:
    push rbp
    mov rbp,rbp
    mov rax, -1 ; Заполнение регистра rax единицами.
    mov al, byte [bNum] ; Верхние (старшие) биты регистра rax НЕ очищать.
    xor rax,rax ; Очистка регистра rax.
    mov al, byte [bNum] ; Теперь rax содержит корректное значение.
    mov rax, -1 ; Заполнение регистра rax единицами.
    mov ax, word [wNum] ; Верхние (старшие) биты регистра rax НЕ очищать.
    xor rax,rax ; Очистка регистра rax.
    mov ax, word [wNum] ; Теперь rax содержит корректное значение.
    mov rax, -1 ; Заполнение регистра rax единицами.
```

```

mov eax, dword [dNum] ; Очистить верхние (старшие) биты регистра гах.
mov гах, -1           ; Заполнение регистра гах единицами.
mov гах, qword [qNum1] ; Очистить верхние (старшие) биты регистра гах.
mov qword [qNum2], гах ; Один операнд всегда должен быть регистром.
mov гах, 123456       ; Операнд-источник - непосредственное значение.
movq xmm0, [qNum3]    ; Инструкция для числа с плавающей точкой.
mov rsp, rbp
pop rbp
ret

```

Сохраните этот исходный код в файле *move.asm*, выполните сборку и запустите полученный выполняемый файл, чтобы убедиться, что он работает. При запуске эта программа не должна ничего выводить на экран. В командной строке выполните команду

```
ddd move
```

Вы увидите окно графического пользовательского интерфейса со специализированной разметкой (см. рис. 6.1). DDD представляет собой старое испытанное инструментальное средство с открытым исходным кодом, и вряд ли кто-то пожелает адаптировать его к современным стандартам GUI.

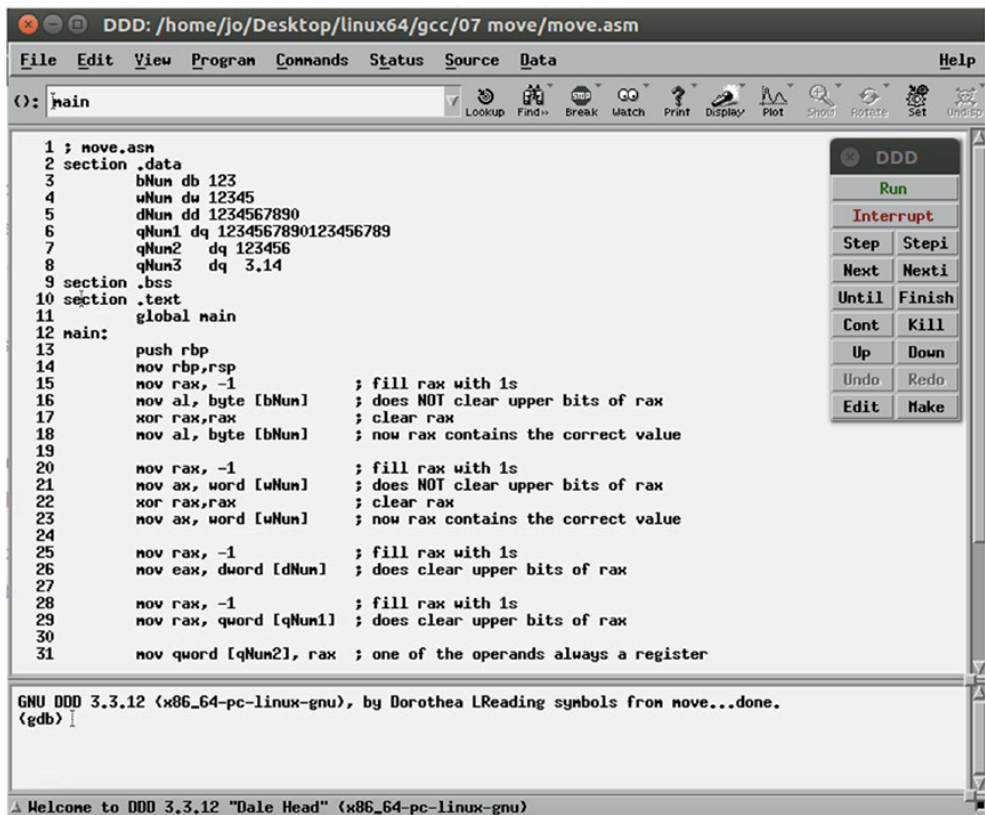


Рис. 6.1. Окно отладчика DDD

На экране отображается окно с исходным кодом и панелью, в которой можно вводить команды отладчика GDB. Также имеется перемещаемая (плавающая) панель, в которой можно щелкать по кнопкам **Run** (Запуск), **Step** (Шаг), **Stepi** (i шагов) и т. д. Щелкните по пункту меню **Source** (Исходный код) и выберите отображение номеров строк. В том же меню можно выбрать создание окна с ассемблированным кодом.

Поместите курсор перед меткой `main`:, затем щелкните правой кнопкой мыши и выберите пункт **Break** (Точка останова) или выберите значок **Stop** (Останов) в меню в верхней части окна. Щелкните по кнопке **Run** (Запуск) на перемещаемой панели, и отладка начнется. Щелкните по пункту **Status** (Состояние) в строке меню в верхней части окна и выберите пункт **Registers** (Регистры). В перемещаемой панели щелкните по кнопке **Step** (Шаг), чтобы выполнить текущую инструкцию. Теперь можно наблюдать за изменением содержимого регистров при пошаговом проходе по программе. Если необходимо обследовать адреса памяти, такие как `qNum1` или `bNum`, то можно воспользоваться пунктом меню **Data** (Данные). Сначала перейдите к пункту **View** (Вид), чтобы сделать видимым окно данных. Затем в спускающемся меню **Data** щелкните по пункту **Memory** (Память). На рис. 6.2 показан пример наблюдения за содержимым ячеек памяти. Поскольку интерфейс DDD не всегда понятен, использование окна ввода отладчика GDB иногда позволяет намного быстрее выполнить требуемую операцию, чем поиск нужной команды в меню.

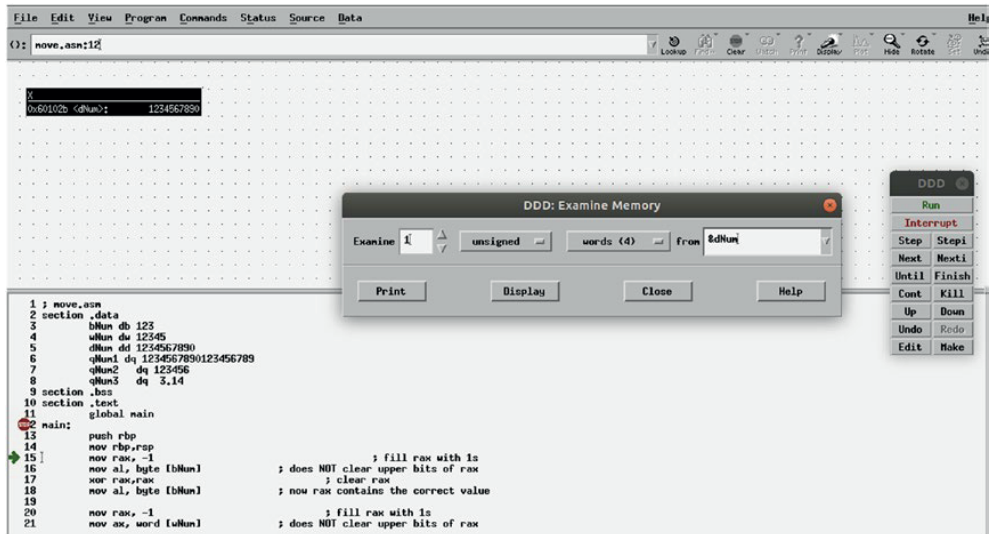


Рис. 6.2. Наблюдение за содержимым памяти в отладчике DDD

Отладчик DDD создан на основе GDB, поэтому обязательным требованием является использование пролога и эпилога функции, для того чтобы избежать проблем. Обратите внимание: при пошаговом проходе по программе DDD просто игнорирует пролог.

Главная цель этого кода – показать, что происходит с содержимым регистров при использовании команды `mov`. Откройте окно регистров в DDD (в меню вы-

берите **Status** → **Registers**). Обратите внимание: изначально регистр `eax` содержит -1, это означает, что все биты в регистре `eax` равны 1. Если вы не понимаете, почему, то вернитесь к главе 2 и еще раз прочтите раздел о представлении двоичных чисел в компьютерах. Вы увидите, что если число помещается в регистр `eax` или `ax`, то верхние биты регистра `eax` не сбрасываются в 0 (не очищаются), и в результате регистр `eax` содержит значение, не равное значению `eax` или `ax`. В рассматриваемом здесь примере `eax` содержит `0xffffffffffff7b`, т. е. большое отрицательное число. Но в регистре `eax` содержится значение `0x7b` – десятичное число 123, как ожидалось. Это может совпадать или не совпадать с вашими намерениями. Если при вычислениях вы по ошибке использовали регистр `eax` вместо `eax`, то результат окажется абсолютно неверным. Но поскольку вы продолжаете пошаговый проход по коду, то увидите, что при передаче 32-битового значения в 64-битовый регистр верхние биты 64-битового регистра будут очищены (сброшены в 0). При записи значения в регистр `eax` верхние биты регистра `eax` очищаются. Об этом важно помнить всегда.

В конце рассматриваемого примера значение из регистра перемещается в `qNum2`. Обратите внимание на квадратные скобки, сообщающие ассемблеру, что `qNum2` – это адрес памяти. И самое последнее действие – «непосредственное значение» помещается в регистр.

РЕЗЮМЕ

В этой главе вы узнали:

- об отладчике DDD, хотя и немного устаревшем, но все еще используемом в качестве отладчика на основе GDB;
- о том, что копирование значения в 8-битовый и 16-битовый регистры не очищает (не сбрасывает в 0) верхнюю часть 64-битового регистра, ...
- ...но копирование значения в 32-битовый регистр очищает (сбрасывает в 0) верхнюю часть 64-битового регистра.

Глава 7

Переходы и циклы

Вероятнее всего, вы согласитесь, что такой отладчик, как DDD, полезен, особенно для обследования больших программ. В этой главе будет представлено программное средство SASM (Simple ASM). Это кроссплатформенная интегрированная среда разработки (integrated development environment – IDE) с открытым исходным кодом. SASM поддерживает подсветку синтаксиса и отладку в режиме графического пользовательского интерфейса. Это великолепное инструментальное средство для программиста на ассемблере.

УСТАНОВКА SIMPLEASM

Необходимо перейти на веб-страницу <https://dman95.github.io/SASM/English.html>, выбрать версию для ОС, которую вы используете, и начать установку. Для Ubuntu 18.04 нужно перейти в каталог *xUbuntu_18.04/amd64/*, затем загрузить и установить пакет *sasm_3.10.1_amd64.deb* следующей командой:

```
sudo dpkg -i sasm_3.10.1_amd64.deb
```

Если выводится сообщение об ошибке из-за проблем с зависимостями, то требуется установить отсутствующие пакеты и повторить процедуру установки SASM. Можно также попробовать выполнить следующую команду:

```
sudo apt --fix-broken install
```

Обычно это позволяет установить все требуемые, но пока отсутствующие пакеты.

ИСПОЛЬЗОВАНИЕ SASM

Запустите SASM, выполнив команду *sasm* в командной строке, и выберите язык интерфейса. SASM начнет загружаться, но если в командной строке появится сообщение об ошибке «Failed to load module "canberra-gtk-module"» (Невозможно загрузить модуль "canberra-gtk-module"), то установите следующие пакеты:

```
sudo apt install libcanberra-gtk*
```

После установки заданной группы файлов сообщение об ошибке больше не должно появляться.

В окне SASM перейдите в диалоговую панель **Settings** (Параметры настройки), как показано на рис. 7.1. На вкладке **Common** (Общие) выберите вариант **Yes** (Да) для пункта **Show all registers in debug:** (Показывать все регистры при отладке:).

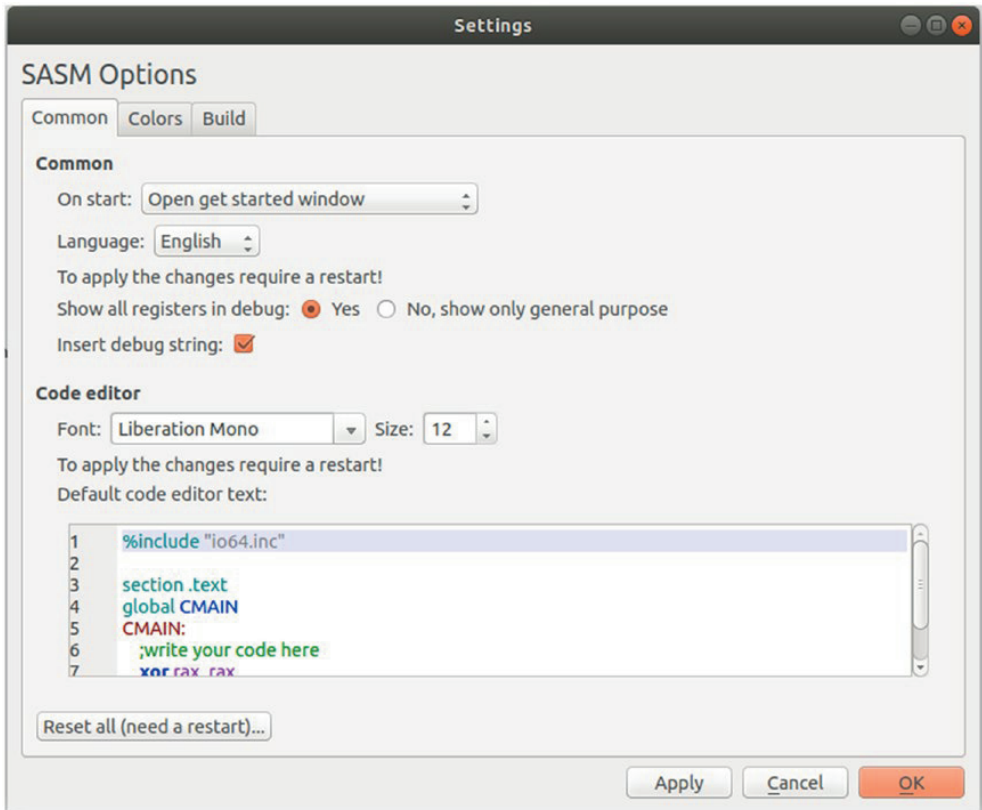


Рис. 7.1. SASM, диалоговое окно **Settings** (Параметры настройки), вкладка **Common** (Общие)

На вкладке **Build** (Сборка) измените параметры настройки, как показано на рис. 7.2.

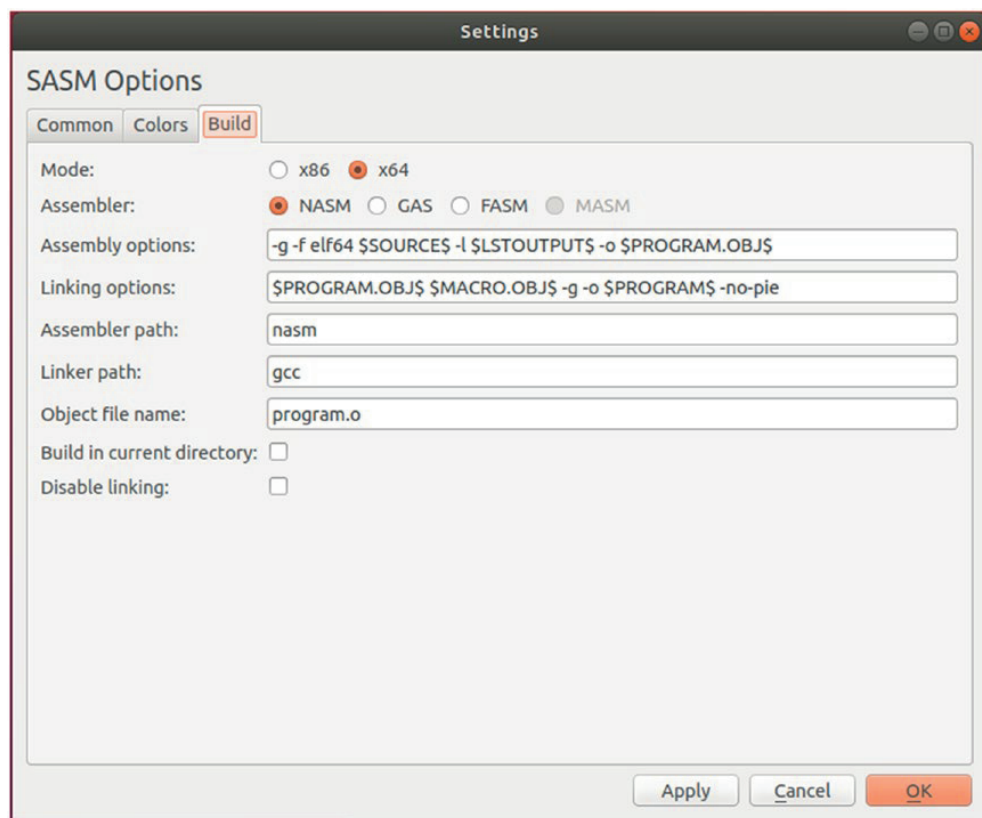


Рис. 7.2. SASM, диалоговое окно **Settings** (Параметры настройки), вкладка **Build** (Сборка)

Будьте весьма внимательны: настройки должны быть в точности такими, как показано на рис. 7.2, – один лишний пробел, даже «спрятанный» в конце строки, и SASM не будет выполнять то, что вам нужно. После правильного ввода всех параметров щелкните по кнопке **OK** и перезапустите SASM.

При начале работы с новым проектом в SASM вы обнаружите некоторый исходный код, который уже находится в окне редактора по умолчанию. Этот код не будет использоваться, поэтому его можно удалить. В командной строке введите

```
sasm jump.asm
```

Если файл *jump.asm* не существует, то SASM запускается с открытием нового окна редактора, в котором нужно просто удалить исходный код по умолчанию. Если файл существует, то его содержимое будет загружено в окно редактора.

В листинге 7.1 показан исходный код, содержащийся в файле *jump.asm*.

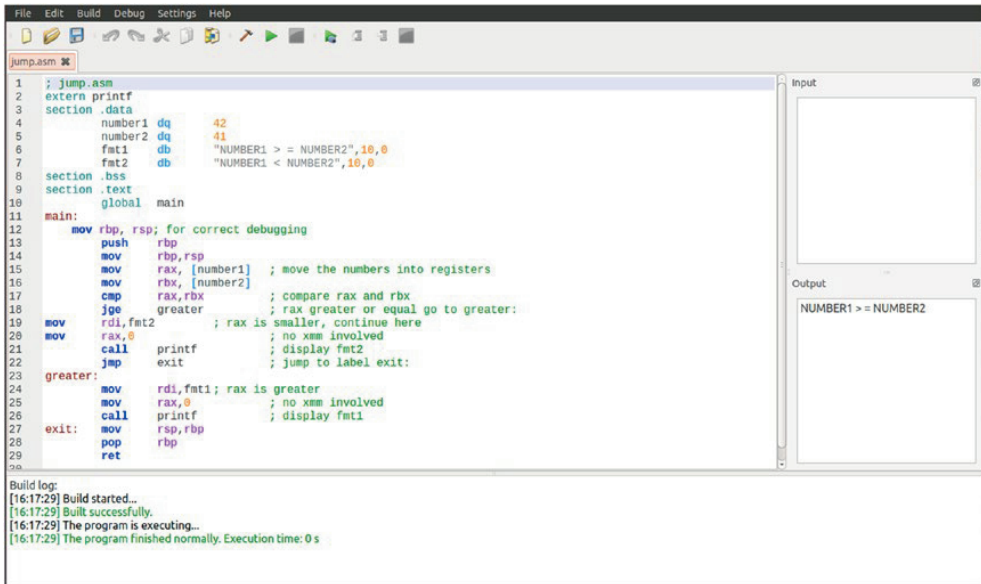
Листинг 7.1. Файл *jump.asm*

```

; jump.asm
extern printf
section .data
    number1    dq    42
    number2    dq    41
    fmt1  db    "NUMBER1 > = NUMBER2",10,0
    fmt2  db    "NUMBER1 < NUMBER2",10,0
section .bss
section .text
    global     main
main:
    push  rbp
    mov   rbp, rsp
    mov   rax, [number1] ; Передача чисел в регистры.
    mov   rbx, [number2]
    cmp   rax, rbx       ; Сравнение регистров rax и rbx.
    jge   greater        ; Если rax больше или равен, то перейти к метке greater:.
mov     rdi, fmt2        ; Если rax меньше, продолжить здесь.
mov     rax, 0           ; Регистр xmm не используется.
    call printf          ; Вывод строки fmt2.
    jmp   exit           ; Переход к метке exit:.
greater:
    mov   rdi, fmt1      ; Регистр rax больше.
    mov   rax, 0         ; Регистр xmm не используется.
    call printf          ; Вывод строки fmt1.
exit:
    mov   rsp, rbp
    pop   rbp
    ret

```

Скопируйте этот исходный код в окно редактора SASM. По умолчанию SASM использует подсветку синтаксиса. После завершения ввода щелкните по кнопке с зеленым треугольником в панели инструментов в верхней части окна. Это кнопка «**Пуск**». Если все было сделано правильно, то в панели **Output** (Вывод) вы увидите результат работы программы, как показано на рис. 7.3.

Рис. 7.3. Вывод программы в панели **Output** окна SASM

При сохранении файла в SASM будет сохранен введенный исходный код. Если вы хотите сохранить выполняемый файл, то необходимо выбрать пункт **Save.exe** в меню **File** (Файл).

Для начала отладки щелкните по столбцу нумерации строк слева от метки **main:**. После этого между меткой **main:** и соответствующим номером строки появится красный кружок. Это символ точки останова. Затем в верхней панели инструментов щелкните по зеленому треугольнику с изображением на нем жука. В главном меню выберите пункт **Debug** (Отладка) и отметьте пункты **Show Registers** (Показывать регистры) и **Show Memory** (Показывать память). На экране появится несколько дополнительных окон: **Registers** (Регистры), **Memory** (Память), а также виджет командной строки отладчика GDB.

Теперь с помощью значка **Step** (Шаг) можно начать пошаговый проход по коду и наблюдать, как изменяются значения в регистрах. Для того чтобы увидеть изменения значения переменной, щелкните правой кнопкой мыши по объявлению переменной в разделе **section .data** и в контекстном меню выберите пункт **Watch** (Наблюдать). Указанная переменная будет добавлена в окно **Memory**, при этом SASM попытается автоматически определить ее тип. Если значение, выведенное SASM, не соответствует ожидаемому, то необходимо вручную изменить тип на правильный. При отладке с использованием SASM для корректной отладки добавляется следующая строка исходного кода:

```
mov rbp, rsp ; Для корректной отладки.
```

Эта строка может запутать другие отладчики, такие как GDB, поэтому не забывайте удалять ее перед запуском GDB отдельно из командной строки.

Следует убедиться в том, что в главном меню **Settings** (Параметры настройки) → **Common** (Общие) выбран вариант **Yes** (Да) для пункта **Show all registers in debug** (Показывать все регистры при отладке). При отладке в SASM выполните прокрутку вниз в окне регистров. В нижней части окна вы увидите 16 xmm-регистров, в каждом из которых указаны два значения в круглых скобках. Первое значение – это соответствующий xmm-регистр. Более подробно эти регистры будут описаны при рассмотрении SIMD.

На рис. 7.4 показан вывод на экран после сборки и запуска программы *jump* тем же способом, который был описан ранее.

```
jo@UbuntuDesktop:~/Desktop/linux64/gcc/08_jump$ make
nasm -f elf64 -g -F dwarf jump.asm -l jump.lst
gcc -o jump jump.o
jo@UbuntuDesktop:~/Desktop/linux64/gcc/08_jump$ ./jump
NUMBER1 > = NUMBER2
jo@UbuntuDesktop:~/Desktop/linux64/gcc/08_jump$ █
```

Рис. 7.4. Вывод программы *jump.asm*

В этой программе использовалась инструкция сравнения *cmp* и две инструкции перехода *jge* и *jmr*. Инструкцию *cmp* называют инструкцией проверки условия или просто условной инструкцией. Здесь *cmp* сравнивает два операнда, в данном случае два регистра. Один из операндов также может быть адресом памяти, а второй операнд может быть непосредственным значением. В любом случае размер обоих операндов обязательно должен быть одинаковым (байт, слово и т. п.). Инструкция *cmp* устанавливает или очищает биты в регистре флагов.

Флаги – это биты, расположенные в регистре *rflags*, которые могут быть установлены в 1 или очищены (сброшены) в 0, в зависимости от нескольких условий. В рассматриваемом здесь примере важными являются флаг нуля (*ZF* – zero flag), флаг переполнения (*OF* – overflow flag) и флаг знака (*SF* – sign flag). Вы можете воспользоваться любым отладчиком для обследования этих и других флагов. При работе с SASM можно с легкостью увидеть, что происходит во всех регистрах, в том числе и в регистре флагов, который в SASM называется *eflags*. Различные значения операндов инструкции *cmp* приводят к установке или очистке разных флагов. Немного поэкспериментируйте со значениями, чтобы наблюдать, что происходит с флагами.

Если необходимо использовать флаги, то вы должны проверять их сразу же после выполнения инструкции *cmp*. Если перед проверкой регистра *rflags* выполняются другие инструкции, то значения флагов могут измениться. В рассматриваемом здесь примере регистр флагов проверяется инструкцией *jge*, означающей «переход, если больше или равно» (*jump if greater than or equal*). Если условие выполнено, то происходит переход (*jump*) на метку, указанную после инструкции *jge*. Если условие не соблюдено, то выполнение продолжается с инструкции, следующей непосредственно за инструкцией *jge*. В табл. 7.1 приведены некоторые наиболее часто используемые условные инструкции, но вы можете найти более подробное описание всех условных инструкций в руководствах Intel.

Таблица 7.1. Инструкции перехода и значения флагов

Ин- струк- ция	Флаги	Описание	Использование
je	ZF=1	Переход, если равно	Знаковые, беззна- ковые
jne	ZF=0	Переход, если не равно	Знаковые, беззна- ковые
jg	((SF XOR OF) OR ZF) = 0	Переход, если больше	Знаковые
jge	(SF XOR OF) = 0	Переход, если больше или равно	Знаковые
jl	(SF XOR OF) = 1	Переход, если меньше	Знаковые
jle	(SF XOR OF) OR ZF = 1	Переход, если меньше или равно	Знаковые
ja	(CF OR ZF) = 0	Переход по условию «больше»	Беззнаковые
jae	CF=0	Переход по условию «больше или равно»	Беззнаковые
jb	CF=1	Переход по условию «меньше»	Беззнаковые
jbe	(CF OR ZF) = 1	Переход по условию «меньше или равно»	Беззнаковые

В рассматриваемой здесь программе также использовалась инструкция безусловного перехода `jmp`. Если при выполнении программы встречается эта инструкция, то происходит переход к метке, указанной после инструкции `jmp`, независимо от значений флагов или каких-либо других условий.

Более сложной формой перехода является зацикливание (looping), означающее многократное повторение группы инструкций, пока выполняется (или не выполняется) некоторое заданное условие. В листинге 7.2 показан пример зацикливания.

Листинг 7.2. Программа *jumploop.asm*

```

; jumploop.asm
extern printf
section .data
    number    dq    5
    fmt       db    "The sum from 0 to %ld is %ld",10,0
section .bss
section .text
    global main
main:
    push     rbp
    mov     rbp, rsp
    mov     rbx,0           ; Счетчик.
    mov     rax,0           ; Сумма будет сохраняться в регистре rax.
jloop:
    add     rax, rbx
    inc     rbx
    cmp     rbx,[number] ; Конечное число итераций цикла достигнуто?
    jle     jloop           ; Конечное число итераций пока не достигнуто, продолжение цикла.
                                ; Конечное число итераций достигнуто, продолжить здесь.

```

```

mov    rdi,fmt      ; Подготовка вывода результата.
mov    rsi,[number]
mov    rdx,rax
mov    rax,0
call   printf
mov    rsp,rbp
pop    rbp
ret

```

Эта программа выполняет сложение чисел от 0 до значения, содержащегося в переменной `number`. Регистр `rbx` используется как счетчик (цикла), а в регистре `rax` сохраняется накапливаемая сумма. Здесь создан цикл (`loop`) – это код между меткой `jloop:` и инструкцией `jle jloop`. В этом цикле значение в регистре `rbx` прибавляется к значению регистра `rax`, после чего значение регистра `rbx` увеличивается на 1, затем выполняется сравнение, чтобы узнать, не было ли достигнуто конечное значение (`number`). Если в регистре `rbx` содержится значение, меньшее или равное значению `number`, то выполнение цикла продолжается, иначе выполнение продолжается с инструкции, следующей непосредственно после цикла, и начинается подготовка к выводу результата. Для увеличения значения регистра `rbx` использована арифметическая инструкция `inc`. Арифметические инструкции будут рассматриваться в следующих главах.

В листинге 7.3 показан другой способ создания цикла.

Листинг 7.3. Программа *betterloop.asm*

```

; betterloop
extern printf
section .data
    number    dq    5
    fmt       db    "The sum from 0 to %ld is %ld",10,0
section .bss
section .text
    global main
main:
    push rbp
    mov  rbp,rsp
    mov  rcx,[number]    ; Инициализация регистра rcx значением number.
    mov  rax, 0
bloop:
    add  rax,rcx          ; Прибавление rcx для получения суммы.
    loop bloop           ; Цикл, пока значение rcx уменьшается на 1.
                        ; До тех пор, когда выполнится условие rcx = 0.
    mov  rdi,fmt          ; rcx = 0, продолжить здесь.
    mov  rsi,[number]    ; Выводимая сумма.
    mov  rdx, rax
    mov  rax,0            ; Без использования чисел с плавающей точкой.
    call printf          ; Вывод результата.
    mov  rsp,rbp
    pop  rbp
    ret

```


Здесь можно видеть, что существует специализированная инструкция `loop`, которая использует регистр `rcx` как уменьшающийся счетчик цикла. При каждом проходе по циклу значение `rcx` автоматически уменьшается на 1, и пока

значение `gsx` не равно 0, цикл выполняется снова и снова. В этом варианте объем вводимого исходного кода меньше.

Можно провести любопытный эксперимент: поместить 1000000000 (единицу с девятью нулями) в переменную `number`, затем пересобрать и запустить обе предыдущие программы. Скорость выполнения программ можно измерить с помощью команды Linux `time`, как показано ниже:

```
time ./jumploop
time ./betterloop
```

Обратите внимание: *betterloop* работает медленнее, чем *jumploop* (см. рис. 7.5). Использование инструкции `loop` более удобно, но за это приходится расплачиваться снижением производительности. Для измерения производительности применялась команда Linux `time`, но в дальнейшем будут продемонстрированы более эффективные способы обследования и тонкой настройки программного кода.



```
jo@UbuntuDesktop:~/Desktop/linux64/gcc/09 loopcompare$ time ./betterloop_long
The sum from 0 to 1000000000 is 500000000500000000

real    0m1.731s
user    0m1.726s
sys     0m0.004s
jo@UbuntuDesktop:~/Desktop/linux64/gcc/09 loopcompare$ time ./jumploop_long
The sum from 0 to 1000000000 is 500000000500000000

real    0m0.404s
user    0m0.391s
sys     0m0.008s
jo@UbuntuDesktop:~/Desktop/linux64/gcc/09 loopcompare$
```

Рис. 7.5. Сравнение использования инструкции цикла и инструкции условного перехода

Возможно, вы удивлены, почему мы все еще используем отладчик DDD, если существует такое удобное инструментальное средство, как SASM. Немного позже вы увидите, что в SASM невозможно наблюдать за состоянием стека, но это можно сделать с помощью отладчика DDD. Поэтому в дальнейшем мы вернемся к работе с DDD.

РЕЗЮМЕ

В этой главе вы узнали, как:

- использовать интегрированную среду разработки SASM;
- применять инструкции перехода;
- использовать инструкцию `cmp`;
- применять инструкцию `loop`;
- проверять значения флагов.

Глава 8

Память

Память используется процессором как место для хранения данных и инструкций. Мы уже рассмотрели регистры – средства хранения данных с весьма высокой скоростью доступа. Доступ к памяти осуществляется значительно медленнее, чем доступ к регистрам. Но количество регистров ограничено. Размер памяти теоретически ограничен 2^{64} адресами, т. е. 18 446 744 073 709 551 616, или 16 Эб (эксабайт). Но такой огромный объем памяти невозможно использовать из-за проблем ее практической реализации. Как бы то ни было, наступило время для более подробного обследования памяти.

ОБСЛЕДОВАНИЕ ПАМЯТИ

В листинге 8.1 показан пример, который будет использоваться постоянно при изучении процесса обследования памяти.

Листинг 8.1. Программа *memory.asm*

```
; memory.asm
section .data
    bNum      db    123
    wNum      dw    12345
    warray    times      5 dw 0      ; Массив из 5 слов, ...
                                         ; ...содержащих 0.
    dNum      dd    12345
    qNum1     dq    12345
    text1     db    "abc",0
    qNum2     dq    3.141592654
    text2     db    "cde",0
section .bss
    bvar      resb   1
    dvar      resd   1
    wvar      resw   10
    qvar      resq   3
section .text
    global main
main:
    push     rbp
    mov     rbp, rsp
    lea     rax, [bNum]      ;Загрузка адреса bNum в регистр rax.
    mov     rax, bNum        ;Загрузка адреса bNum в регистр rax
```

```

mov    rax, [bNum]      ;Загрузка значения bNum в регистр rax
mov    [bvar], rax      ;Загрузка из регистра rax в адрес памяти bvar.
lea    rax, [bvar]      ;Загрузка адреса bvar в регистр rax.
lea    rax, [wNum]      ;Загрузка адреса wNum в регистр rax.
mov    rax, [wNum]      ;Загрузка содержимого wNum в регистр rax.
lea    rax, [text1]     ;Загрузка адреса text1 в регистр rax.
mov    rax, text1       ;Загрузка адреса text1 в регистр rax.
mov    rax, text1+1     ;Загрузка второго символа в регистр rax.
lea    rax, [text1+1]   ;Загрузка второго символа в регистр rax.
mov    rax, [text1]     ;Загрузка, начиная с адреса text1, в регистр rax.
mov    rax, [text1+1]   ;Загрузка, начиная с адреса text1+1, в регистр rax.
mov    rsp,rbp
pop    rbp
ret

```

Введите и сохраните в файле этот исходный код, затем выполните сборку программы. Программа ничего не выводит, поэтому необходимо воспользоваться отладчиком для пошагового прохода по каждой ее инструкции. Здесь будет полезна интегрированная среда разработки SASM.

В рассматриваемом примере определены некоторые переменные с различными размерами, в том числе массив из пяти двойных слов, заполненных нулями. Кроме того, некоторые элементы определены в разделе section .bss. В используемом отладчике посмотрите на содержимое регистра указателя стека `rsp` – он содержит достаточно большое значение. Указатель стека (stack pointer) ссылается на адрес в верхней части памяти. Стек (stack) – это область памяти, используемая для кратковременного хранения данных. Стек растет по мере добавления в него данных в направлении «вниз», т. е. от более старших адресов к младшим. Указатель стека `rsp` уменьшается при каждой операции записи данных в стек. Работа стека будет рассматриваться в отдельной главе, но уже сейчас необходимо запомнить, что стек – это место в области верхней памяти (в области старших адресов памяти). См. рис. 8.1.

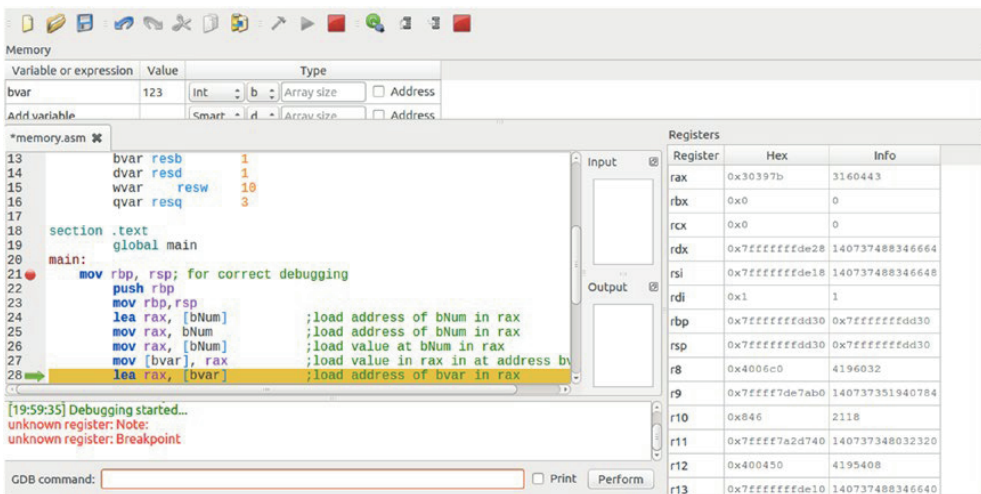


Рис. 8.1. Регистр `rsp` содержит адрес в области верхней памяти (старших адресов)

В рассматриваемом здесь примере использовалась инструкция `lea`, означающая «загрузить эффективный (действительный) адрес» (load effective address) и предназначенная для загрузки адреса памяти переменной `bNum` в регистр `гах`. Тот же результат можно получить с помощью инструкции `mov` без квадратных скобок, окружающих имя `bNum`. Если используются квадратные скобки `[bNum]`, то инструкция `mov` загружает значение, а не адрес переменной `bNum`, в регистр `гах`. Но в этом коде в регистр `гах` загружается не только `bNum`. Поскольку `гах` является 64-битовым (или 8-байтовым) регистром, в него загружается больше байтов. Значение `bNum` является крайним правым байтом в `гах` (порядок байтов от младшего к старшему), поэтому нас интересует только содержимое регистра `al`. Если требуется, чтобы регистр `гах` содержал лишь значение 123, то сначала необходимо очистить `гах`, как показано ниже:

```
xor гah, гah
```

Затем вместо инструкции

```
mov гah, [bNum]
```

нужно использовать инструкцию

```
mov al, [bNum]
```

Внимательно следите за размером данных, которые перемещаются в память и из памяти. Например, рассмотрим следующую инструкцию:

```
mov [bvar], гah
```

В соответствии с этой инструкцией 8 байт из регистра `гах` перемещаются в локацию с адресом `bvar`. Если вы предполагали запись только значения 123 в блок памяти `bvar`, то можете проверить с помощью отладчика и убедиться в том, что вы перезаписали еще 7 байт памяти (в окне памяти `SASM` выберите тип `d` для переменной `bvar`). Это может привести к появлению неожиданных ошибок в программе. Чтобы избежать этого, необходимо заменить некорректную инструкцию на следующую:

```
mov [bvar], al
```

При загрузке содержимого из блока памяти с адресом `text1` в регистр `гах` следует отметить, что значение в `гах` сохраняется с порядком байтов от младшего к старшему (little endian). Выполните пошаговый проход по программе для просмотра результатов выполнения различных инструкций и изменяйте размеры и значения данных, чтобы наблюдать за тем, что происходит при этом.

Существуют два способа загрузки адреса памяти: инструкции `mov` и `lea`. Применение инструкции `lea` может сделать код более удобным для чтения, так как сразу можно видеть, что здесь происходит работа с адресами памяти. Кроме того, инструкцию `lea` можно использовать для ускорения вычислений, но здесь мы не будем применять `lea` для этой цели.

Выполните команду `gdb memory`, затем команду `disass main` и обратите внимание на левый столбец с адресами памяти (рис. 8.2). Не забудьте сначала удалить специальную строку, добавленную SASM для корректной отладки, о которой упоминалось в предыдущей главе. В рассматриваемом здесь примере первая инструкция размещается по адресу `0x4004a0`.

```
(gdb) disass main
Dump of assembler code for function main:
   0x0000000004004a0 <+0>:      mov     rbp, rsp
   0x0000000004004a3 <+3>:      push    rbp
   0x0000000004004a4 <+4>:      mov     rbp, rsp
   0x0000000004004a7 <+7>:      lea     rax, ds:0x601028
   0x0000000004004af <+15>:     movabs  rax, 0x601028
   0x0000000004004b9 <+25>:     mov     rax, QWORD PTR ds:0x601028
   0x0000000004004c1 <+33>:     mov     QWORD PTR ds:0x601058, rax
   0x0000000004004c9 <+41>:     lea     rax, ds:0x601058
   0x0000000004004d1 <+49>:     lea     rax, ds:0x601029
   0x0000000004004d9 <+57>:     mov     rax, QWORD PTR ds:0x601029
   0x0000000004004e1 <+65>:     lea     rax, ds:0x601041
   0x0000000004004e9 <+73>:     movabs  rax, 0x601041
   0x0000000004004f3 <+83>:     movabs  rax, 0x601042
   0x0000000004004fd <+93>:     lea     rax, ds:0x601042
   0x000000000400505 <+101>:    mov     rax, QWORD PTR ds:0x601041
   0x00000000040050d <+109>:    mov     rax, QWORD PTR ds:0x601042
   0x000000000400515 <+117>:    mov     rsp, rbp
   0x000000000400518 <+120>:    pop     rbp
   0x000000000400519 <+121>:    ret
   0x00000000040051a <+122>:    nop     WORD PTR [rax+rax*1+0x0]
End of assembler dump.
(gdb) █
```

Рис. 8.2. Код функции `main`, дизассемблированный отладчиком GDB

Теперь в командной строке воспользуемся утилитой `readelf`. Напомню, что NASM был проинформирован о необходимости ассемблирования исходного кода в выполняемый формат ELF (см. соответствующий *makefile*). `readelf` – это утилита командной строки, используемая для получения более подробной информации о выполняемом файле. Если у вас возникло непреодолимое желание узнать больше о программах редактирования связей (линкерах), то можно порекомендовать качественный источник информации: *Linkers and Loaders*, John R. Levine, 1999, The Morgan Kaufmann, Series in Software Engineering and Programming.

Более краткое описание формата ELF можно найти здесь:

<https://linux-audit.com/elf-binaries-on-linux-understanding-and-analysis/>

или

<https://www.cirosantilli.com/elf-hello-world/>.

Вероятно, вы уже догадались, что в командной строке можно ввести следующую команду:

```
man elf
```

Для рассматриваемого здесь примера необходимо выполнить следующую команду:


```
readelf --file-header ./memory
```

Выводится некоторая общая информация о памяти, используемой выполняемым файлом *memory*. Обратите внимание на строку *Entry point address*: 0x4003b0. Это локация в памяти, в которой расположено начало программы. Очевидно, что между точкой входа (*entry point*) в программу и началом кода, которое показано в отладчике GDB (0x4004a0), существует некоторый добавочный заголовок. Этот заголовок предоставляет нам дополнительную информацию об операционной системе и выполняемом коде. См. рис. 8.3.

```
jo@ubuntu18:~/Desktop/linux64/gcc/10 memory$ readelf --file-header ./memory
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                           ELF64
  Data:                               2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                        0
  Type:                               EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:                0x4003b0
  Start of program headers:           64 (bytes into file)
  Start of section headers:          7192 (bytes into file)
  Flags:                              0x0
  Size of this header:                 64 (bytes)
  Size of program headers:            56 (bytes)
  Number of program headers:           9
  Size of section headers:            64 (bytes)
  Number of section headers:          34
  Section header string table index:  33
jo@ubuntu18:~/Desktop/linux64/gcc/10 memory$ █
```

Рис. 8.3. Вывод заголовка ELF с помощью утилиты *readelf*

Утилита *readelf* удобна для обследования бинарных выполняемых файлов. На рис. 8.4 показано несколько дополнительных примеров.

```
jo@ubuntu18:~/Desktop/linux64/gcc/10 memory$ readelf --symbols ./memory |grep main
1: 0000000000000000 0 FUNC GLOBAL DEFAULT UND __libc_start_main@GLIBC_2.2.5 (2)
64: 0000000000000000 0 FUNC GLOBAL DEFAULT UND __libc_start_main@GLIBC_
74: 00000000004004a0 0 NOTYPE GLOBAL DEFAULT 11 main
jo@ubuntu18:~/Desktop/linux64/gcc/10 memory$ █
```

Рис. 8.4. Вывод символов программы с помощью утилиты *readelf*

Используя утилиту *grep*, можно выполнить поиск всех строк, содержащих слово *main*. На рис. 8.4 можно видеть, что основная функция *main* начинается с адреса 0x4004a0, как показано в отладчике GDB. В следующем примере выполняется поиск в таблице символов каждого вхождения метки *start*. На рис. 8.5 можно видеть начальные адреса разделов *section .data*, *section .bss*, а также начальный адрес самой программы.


```
jo@ubuntu18:~/Desktop/linux64/gcc/10 memory$ readelf --symbols ./memory |grep start
1: 0000000000000000 0 FUNC GLOBAL DEFAULT UND __libc_start_main@GLIBC_2.2.5 (2)
2: 0000000000000000 0 NOTYPE WEAK DEFAULT UND __gmon_start__
57: 00000000000000e50 0 NOTYPE LOCAL DEFAULT 16 __init_array_start
61: 00000000000001018 0 NOTYPE WEAK DEFAULT 21 data_start
64: 0000000000000000 0 FUNC GLOBAL DEFAULT UND __libc_start_main@GLIBC_
65: 00000000000001018 0 NOTYPE GLOBAL DEFAULT 21 __data_start
66: 0000000000000000 0 NOTYPE WEAK DEFAULT UND __gmon_start__
72: 00000000000003b0 43 FUNC GLOBAL DEFAULT 11 __start
Help: 00000000000001051 0 NOTYPE GLOBAL DEFAULT 22 __bss_start
jo@ubuntu18:~/Desktop/linux64/gcc/10 memory$
```

Рис. 8.5. Вывод символов, содержащих слово start, с помощью утилит readelf и grep

Рассмотрим подробнее, что находится в памяти, с помощью команды, показанной ниже:

```
readelf --symbols ./memory |tail +10|sort -k 2 -r
```

Утилита tail позволяет исключить строки, которые в данный момент нас не интересуют. Выполняется сортировка по второму столбцу (по адресам памяти) в обратном порядке. Вы сами видите, как полезны знания о командах системы Linux.

Началу программы соответствует некоторый младший адрес, а начало функции main находится по адресу 0x004004a0. Найден адрес начала раздела section .data (0x00601018) с адресами всех его переменных и адрес начала раздела section .bss (0x00601051) с адресами, зарезервированными для переменных этого раздела.

Подведем итог всех поисков: в начале главы мы выяснили, что стек размещается в верхних (старших) адресах памяти (см. регистр rsp). С помощью утилиты readelf было обнаружено, что выполняемый код размещается в нижней части памяти (в области младших адресов). Выше выполняемого кода находится раздел section .data, а над ним раздел section .bss. Стек в верхней памяти может расти вниз, т. е. в направлении раздела section .bss. Доступная свободная память между стеком и другими разделами программы называется кучей (heap; более формально – динамически распределяемой памятью).

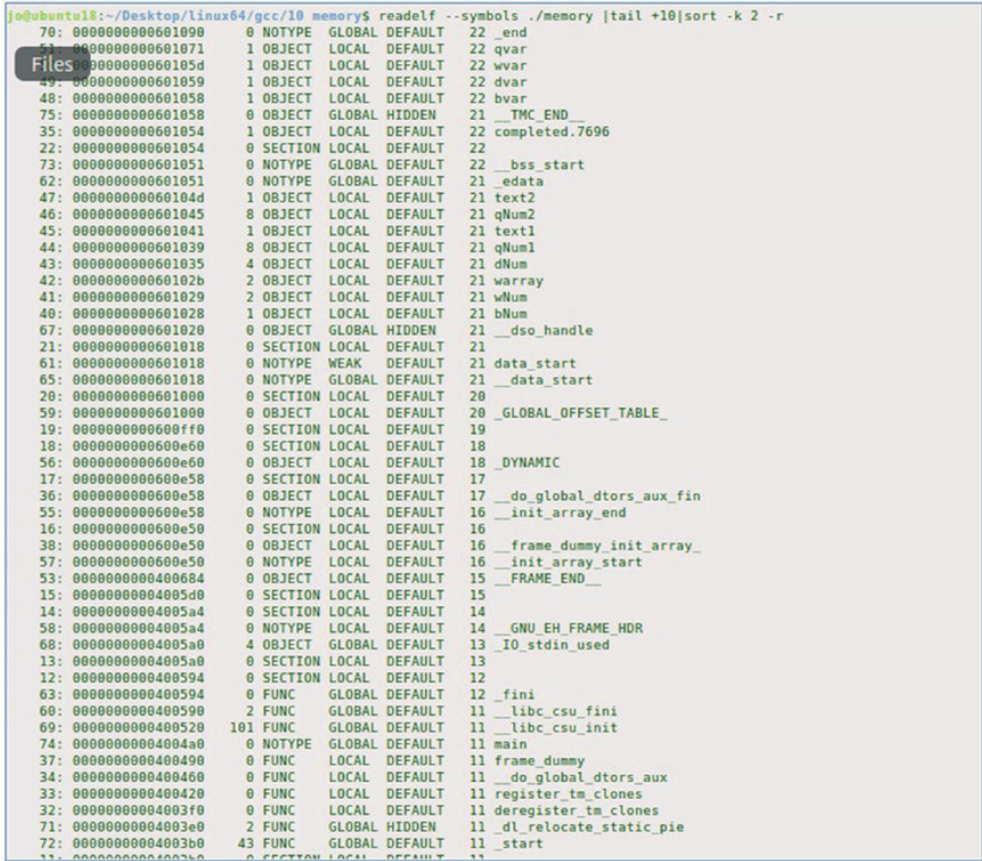
Память в разделе section .bss распределяется во время выполнения программы – это легко проверить. Обратите внимание на размер выполняемого кода, а затем замените, например, строку

```
qvar      resq      3
```

на следующую:

```
qvar      resq      30000
```

После этого пересоберите программу и снова посмотрите на размер выполняемого кода. Размер останется таким же, потому что дополнительная память не резервируется во время ассемблирования и связывания. См. рис. 8.6.



```
jo@ubuntu18:~/Desktop/linux64/gcc/10$ readelf --symbols ./memory | tail +10 | sort -k 2 -r
70: 0000000000001090 0 NOTYPE GLOBAL DEFAULT 22 __end
71: 0000000000001071 1 OBJECT LOCAL DEFAULT 22 qvar
72: 000000000000105d 1 OBJECT LOCAL DEFAULT 22 wvar
49: 0000000000001059 1 OBJECT LOCAL DEFAULT 22 dvar
48: 0000000000001058 1 OBJECT LOCAL DEFAULT 22 bvar
75: 0000000000001058 0 OBJECT GLOBAL HIDDEN 21 __TMC_END__
35: 0000000000001054 1 OBJECT LOCAL DEFAULT 22 completed.7696
22: 0000000000001054 0 SECTION LOCAL DEFAULT 22
73: 0000000000001051 0 NOTYPE GLOBAL DEFAULT 22 __bss_start
62: 0000000000001051 0 NOTYPE GLOBAL DEFAULT 21 __edata
47: 000000000000104d 1 OBJECT LOCAL DEFAULT 21 text2
46: 0000000000001045 8 OBJECT LOCAL DEFAULT 21 qNum2
45: 0000000000001041 1 OBJECT LOCAL DEFAULT 21 text1
44: 0000000000001039 8 OBJECT LOCAL DEFAULT 21 qNum1
43: 0000000000001035 4 OBJECT LOCAL DEFAULT 21 dNum
42: 000000000000102b 2 OBJECT LOCAL DEFAULT 21 warray
41: 0000000000001029 2 OBJECT LOCAL DEFAULT 21 wNum
40: 0000000000001028 1 OBJECT LOCAL DEFAULT 21 bNum
67: 0000000000001020 0 OBJECT GLOBAL HIDDEN 21 __dso_handle
21: 0000000000001018 0 SECTION LOCAL DEFAULT 21
61: 0000000000001018 0 NOTYPE WEAK DEFAULT 21 data_start
65: 0000000000001018 0 NOTYPE GLOBAL DEFAULT 21 __data_start
20: 0000000000001000 0 SECTION LOCAL DEFAULT 20
59: 0000000000001000 0 OBJECT LOCAL DEFAULT 20 __GLOBAL_OFFSET_TABLE__
19: 0000000000000ff0 0 SECTION LOCAL DEFAULT 19
18: 0000000000000e60 0 SECTION LOCAL DEFAULT 18
56: 0000000000000e60 0 OBJECT LOCAL DEFAULT 18 __DYNAMIC
17: 0000000000000e58 0 SECTION LOCAL DEFAULT 17
36: 0000000000000e58 0 OBJECT LOCAL DEFAULT 17 __do_global_dtors_aux_fini
55: 0000000000000e58 0 NOTYPE LOCAL DEFAULT 16 __init_array_end
16: 0000000000000e50 0 SECTION LOCAL DEFAULT 16
38: 0000000000000e50 0 OBJECT LOCAL DEFAULT 16 __frame_dummy_init_array__
57: 0000000000000e50 0 NOTYPE LOCAL DEFAULT 16 __init_array_start
53: 0000000000000e84 0 OBJECT LOCAL DEFAULT 15 __FRAME_END__
15: 0000000000000d0 0 SECTION LOCAL DEFAULT 15
14: 00000000000005a4 0 SECTION LOCAL DEFAULT 14
58: 00000000000005a4 0 NOTYPE LOCAL DEFAULT 14 __GNU_EH_FRAME_HDR
68: 00000000000005a0 4 OBJECT GLOBAL DEFAULT 13 __IO_stdin_used
13: 00000000000005a0 0 SECTION LOCAL DEFAULT 13
12: 0000000000000594 0 SECTION LOCAL DEFAULT 12
63: 0000000000000594 0 FUNC GLOBAL DEFAULT 12 __fini
60: 0000000000000590 2 FUNC GLOBAL DEFAULT 11 __libc_csu_fini
69: 0000000000000520 101 FUNC GLOBAL DEFAULT 11 __libc_csu_init
74: 00000000000004a0 0 NOTYPE GLOBAL DEFAULT 11 main
37: 0000000000000490 0 FUNC LOCAL DEFAULT 11 frame_dummy
34: 0000000000000460 0 FUNC LOCAL DEFAULT 11 __do_global_dtors_aux
33: 0000000000000420 0 FUNC LOCAL DEFAULT 11 register_tm_clones
32: 00000000000003f0 0 FUNC LOCAL DEFAULT 11 deregister_tm_clones
71: 00000000000003e0 2 FUNC GLOBAL HIDDEN 11 __dl_relocate_static_pie
72: 00000000000003b0 43 FUNC GLOBAL DEFAULT 11 __start
11: 0000000000000350 0 SECTION LOCAL DEFAULT 11
```

Рис. 8.6. Вывод команды `readelf --symbols ./memory | tail +10 | sort -k 2 -r`

На рис. 8.7 показано, как в итоге выглядит память после загрузки выполняемого кода.

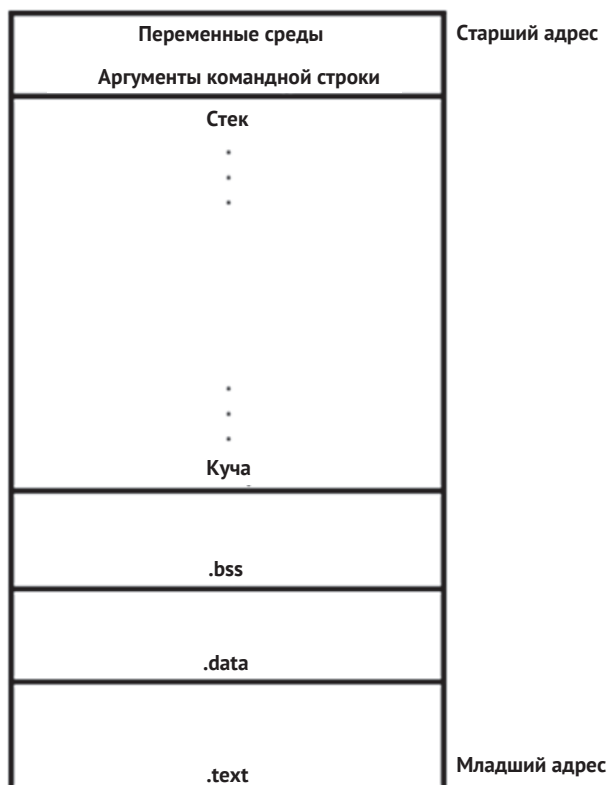


Рис. 8.7. Схема памяти, выделенной программе

Почему так важно знать структуру памяти? Весьма важно понимать, что стек растет в направлении нижней памяти (младших адресов). Это понимание потребуется в следующих главах книги, когда мы будем детально обследовать стек. Кроме того, если вы работаете в сфере судебных расследований, связанных с вредоносным программным обеспечением, то практический навык анализа памяти чрезвычайно важен. Здесь рассматривались лишь некоторые основные положения, но если вы хотите знать больше, то обратитесь к упомянутым выше источникам.

РЕЗЮМЕ

В этой главе вы узнали:

- о структуре памяти (выполняемого) процесса;
- как избежать непреднамеренной перезаписи памяти;
- как использовать утилиту `readelf` для анализа двоичного кода.

Глава 9

Целочисленная арифметика

В этой главе рассматривается группа арифметических инструкций для целых чисел. Арифметика чисел с плавающей точкой описана в главе 11. Рекомендуется бегло просмотреть главу 2, чтобы освежить знания о двоичных (бинарных) числах.

ОСНОВЫ ИСПОЛЬЗОВАНИЯ ЦЕЛОЧИСЛЕННОЙ АРИФМЕТИКИ

В листинге 9.1 показан пример кода, который мы будем анализировать.

Листинг 9.1. Программа *icalc.asm*

```
; icalc.asm
extern printf
section .data
    number1    dq    128    ; Числа, используемые для
    number2    dq    19     ; демонстрации арифметических вычислений
    neg_num     dq    -12    ; и распространения знакового разряда.
    fmt         db    "The numbers are %ld and %ld",10,0
    fmtint      db    "%s %ld",10,0
    sumi        db    "The sum is",0
    difi        db    "The difference is",0
    inci        db    "Number 1 Incremented:",0
    deci        db    "Number 1 Decrementd:",0
    sali        db    "Number 1 Shift left 2 (x4):",0
    sari        db    "Number 1 Shift right 2 (/4):",0
    sariex      db    "Number 1 Shift right 2 (/4) with "
                db    "sign extension:",0
    multi        db    "The product is",0
    divi        db    "The integer quotient is",0
    remi        db    "The modulo is",0
section .bss
    resulti     resq 1
    modulo      resq 1
section .text
    global main
main:
    push rbp
    mov rbp, rsp
```

```
; Вывод чисел.
    mov     rdi, fmt
    mov     rsi, [number1]
    mov     rdx, [number2]
    mov     rax, 0
    call    printf

; Сложение-----
    mov     rax, [number1]
    add     rax, [number2]          ; Сложение number2 с rax.
    mov     [resulti], rax         ; Перемещение суммы в переменную result.
; displaying the result
    mov     rdi, fmtint
    mov     rsi, sumi
    mov     rdx, [resulti]
    mov     rax, 0
    call    printf

; Вычитание-----
    mov     rax, [number1]
    sub     rax, [number2]          ; Вычитание number2 из rax.
    mov     [resulti], rax
; displaying the result
    mov     rdi, fmtint
    mov     rsi, difi
    mov     rdx, [resulti]
    mov     rax, 0
    call    printf

; Инкрементирование-----
    mov     rax, [number1]
    inc     rax                    ; Инкрементирование rax на 1.
    mov     [resulti], rax
; displaying the result
    mov     rdi, fmtint
    mov     rsi, inci
    mov     rdx, [resulti]
    mov     rax, 0
    call    printf

; Декрементирование-----
    mov     rax, [number1]
    dec     rax                    ; Декрементирование rax на 1.
    mov     [resulti], rax
; displaying the result
    mov     rdi, fmtint
    mov     rsi, deci
    mov     rdx, [resulti]
    mov     rax, 0
    call    printf

; Арифметический сдвиг влево-----
    mov     rax, [number1]
    sal     rax, 2                  ; Умножение rax на 4.
    mov     [resulti], rax
; displaying the result
    mov     rdi, fmtint
    mov     rsi, sali
    mov     rdx, [resulti]
    mov     rax, 0
    call    printf
```

```

; Арифметический сдвиг вправо-----
    mov     rax, [number1]
    sar     rax, 2                ; Деление rax на 4.
    mov     [resulti], rax
    ; вывод результата
    mov     rdi, fmtint
    mov     rsi, sari
    mov     rdx, [resulti]
    mov     rax, 0
    call    printf

; Арифметический сдвиг вправо с распространением знакового разряда -----
    mov     rax, [neg_num]
    sar     rax, 2                ; Деление rax на 4.
    mov     [resulti], rax
    ; вывод результата
    mov     rdi, fmtint
    mov     rsi, sariex
    mov     rdx, [resulti]
    mov     rax, 0
    call    printf

; Умножение-----
    mov     rax, [number1]
    imul    qword [number2]      ; Умножение rax на number2.
    mov     [resulti], rax
    ; вывод результата
    mov     rdi, fmtint
    mov     rsi, multi
    mov     rdx, [resulti]
    mov     rax, 0
    call    printf

; Деление-----
    mov     rax, [number1]
    mov     rdx, 0                ; В rdx должен быть 0 перед idiv.
    idiv    qword [number2]      ; Деление rax на number2, остаток в rdx.
    mov     [resulti], rax
    mov     [modulo], rdx        ; Запись содержимого rdx в modulo.

; Вывод результата.
    mov     rdi, fmtint
    mov     rsi, divi
    mov     rdx, [resulti]
    mov     rax, 0
    call    printf
    mov     rdi, fmtint
    mov     rsi, remi
    mov     rdx, [modulo]
    mov     rax, 0
    call    printf

mov     rsp,rbp
pop     rbp
ret
    
```

На рис. 9.1 показан вывод этой программы.

```

jo@UbuntuDesktop:~/Desktop/linux64/gcc/11 icalc$ make
nasm -f elf64 -g -F dwarf icalc.asm -l icalc.lst
gcc -o icalc icalc.o -no-pie
jo@UbuntuDesktop:~/Desktop/linux64/gcc/11 icalc$ ./icalc
The numbers are 128 and 19
The sum is 147
The difference is 109
Number 1 Incremented: 129
Number 1 Decrementd: 127
Number 1 Shift left 2 (x4): 512
Number 1 Shift right 2 (/4): 32
Number 1 Shift right 2 (/4) with sign extension: -3
The product is 2432
The integer quotient is 6
The modulo is 14
jo@UbuntuDesktop:~/Desktop/linux64/gcc/11 icalc$ █

```

Рис. 9.1. Целочисленная арифметика

ИЗУЧЕНИЕ АРИФМЕТИЧЕСКИХ ИНСТРУКЦИЙ

В ассемблере доступно много арифметических инструкций, но здесь будут рассматриваться лишь некоторые из них, а остальные инструкции похожи на изучаемые в этой главе. Перед детальным изучением арифметических инструкций необходимо отметить, что в рассматриваемой здесь программе используется функция `printf`, принимающая более двух аргументов, поэтому требуется дополнительный регистр: первый аргумент записывается в регистр `rdi`, второй – в `rsi`, третий – в `rdx`. Именно такой порядок ожидает `printf` при передаче аргументов в среде Linux. Более подробно об этом вы узнаете немного позже, при обсуждении соглашений при вызовах функций.

Ниже приведено описание некоторых арифметических инструкций:

- первая инструкция `add` – ее можно использовать для сложения целых чисел со знаком или без знака. Вторым операнд (источник) прибавляется к первому операнду (цели), а результат помещается в первый операнд (цель). Целевым операндом может быть регистр или блок памяти. Источником может быть непосредственное значение, регистр или блок памяти. В одной инструкции сложения нельзя одновременно использовать блоки памяти как источник и цель. Если полученная в результате сумма слишком велика и не умещается в цели, то для целых чисел со знаком устанавливается (в 1) флаг `CF`, а для чисел без знака в той же ситуации устанавливается флаг `OF`. Если результат равен 0, то флаг `ZF` устанавливается в 1, а если результат отрицательный, то устанавливается флаг `SF`;
- действие инструкции вычитания `sub` аналогично действию инструкции `add`;
- для инкрементирования регистра или значения в памяти с увеличением на 1 используется инструкция `inc`. Аналогичная, но противоположная по действию инструкция `dec` используется для декрементирования регистра или значения в памяти с уменьшением на 1;
- инструкции арифметического сдвига – это особый тип арифметических инструкций. Инструкция сдвига влево `sal` – это, по существу, умножение: если выполняется сдвиг влево на одну позицию, то число умножается на 2. Каждый бит сдвигается влево на одну позицию, а справа добавля-

ется 0. Рассмотрим двоичное число 1. При сдвиге влево на одну позицию получается двоичное число 10 или 2 в десятичном представлении. Следующий сдвиг влево на одну позицию дает результат 100 (двоичный), т. е. 4 в десятичном представлении. Если выполнить сдвиг влево сразу на две позиции, то это соответствует умножению на 4. А что, если нужно умножить на 6? Выполняется сдвиг влево два раза, затем два раза сложение с первоначальным источником, именно в таком порядке;

- сдвиг вправо `sar` похож на сдвиг влево, но означает деление на 2. Каждый бит сдвигается на одну позицию вправо, а слева добавляется дополнительный бит. Но здесь возникает сложность: если исходное значение было отрицательным, то добавляемый крайний левый бит должен быть равен 1, а если инструкция сдвига добавила бит 0 слева, то значение должно стать положительным, т. е. результат становится неверным. Поэтому при работе с отрицательными числами `sar` добавляет слева бит, равный 1, а в случае с положительным значением слева добавляются биты, равные 0. Это называется распространением (или расширением) знакового разряда (*sign extension*). Кстати, самый быстрый способ узнать, является ли шестнадцатеричное число отрицательным, – проверка байта номер 7 (крайний левый байт при отсчете с 0 от крайнего правого байта). Число отрицательное, если байт номер 7 начинается с 8, 9, A, B, C, D, E или F. Но при этом необходимо принять во внимание содержимое всех 8 байтов. Например, число `0xd12` остается положительным, так как крайний левый байт, который не показан в этом представлении, равен 0;
- существуют также инструкции неарифметического сдвига, которые будут рассматриваться в главе 16;
- далее выполняется умножение целых чисел. Для умножения целых чисел без знака можно воспользоваться инструкцией `mul` – беззнаковое умножение, а инструкция `imul` применяется для умножения с учетом знака. Мы будем использовать `imul` – инструкцию знакового умножения, которая предоставляет большую гибкость: она может принимать один, два или три операнда. В рассматриваемом здесь примере используется один операнд, следующий непосредственно за инструкцией `imul`, – он умножается на значение в регистре `rax`. Возможно, вы предположили, что результат умножения сохраняется в регистре `rax`, но это совершенно неверно. Рассмотрим выполнение этой операции на примере: вы сами можете проверить, что при умножении, скажем, двузначного числа на трехзначное число произведение будет четырехзначным или пятизначным. При умножении 48-битового числа на 30-битовое число вы получите 77-битовое или 78-битовое число, а такое значение не уместится в 64-битовом регистре. Чтобы справиться с этой проблемой, инструкция `imul` сохраняет младшие 64 бит полученного произведения в регистре `rax`, а старшие 64 бит в регистре `rdx`. Такой подход может ввести в заблуждение.

Проведем небольшой эксперимент: вернемся к исходному коду в `SASM`. Изменим значение `number1` так, чтобы эта переменная содержала число 12345678901234567, а переменной `number2` теперь будет присвоено значение 100. Произведение этих чисел будет в точности соответствовать размеру регистра `rax` – это можно проверить в режиме отладки `SASM`. Назначим точку останова перед инструкцией `imul`. Перезапустите режим отладки и нач-

- следующая операция – целочисленное деление `idiv`. В действительности это операция, обратная умножению (а чего вы ожидали?). Делимое находится в регистрах `rdx:rax`, делитель – в операнде-источнике, а результат (частное) целочисленного деления (целое число – `integer`) сохраняется в регистре `rax`. Остаток (`modulo`) сохранен в регистре `rdx`. Легко забыть о весьма важном факте: необходимо убедиться в том, что в регистре `rdx` содержится ноль, перед каждым выполнением инструкции `idiv`, иначе полученное частное может оказаться неверным.

Операциям умножения и деления 64-битовых целых чисел присущи некоторые тонкости, более подробное описание которых можно найти в руководствах Intel. В этой главе приведен лишь краткий обзор, служащий общим введением в целочисленную арифметику. В руководствах Intel представлены подробнейшие описания не только перечисленных выше инструкций, но и многих других арифметических инструкций, которые можно применять в особых ситуациях.

РЕЗЮМЕ

В этой главе вы узнали:

- как использовать целочисленную арифметику;
- как выполнять арифметический сдвиг влево и вправо;
- о том, что операция умножения использует регистры `rax` и `rdx` для сохранения произведения;
- о том, что операция деления использует регистры `rax` и `rdx` для хранения делимого;
- о необходимости внимательного и осторожного использования функции `printf` при выводе значений.

Глава 10

Стек

В предыдущих главах мы рассмотрели использование регистров, особого типа временного быстрого хранилища данных, которые могут применяться для хранения значений или адресов во время выполнения инструкций. Существует также более медленный тип хранилища – оперативная память, куда процессор может записывать значения для более длительного хранения. Наконец, существует стек (stack) – непрерывный массив блоков памяти.

ИЗУЧЕНИЕ РАБОТЫ СТЕКА

Как было отмечено в главе 8, сегмент стека начинается в старших адресах памяти (в верхней памяти) и растет вниз, в направлении к младшим адресам памяти, подобно сосульке. Элементы данных помещаются в стек инструкцией `push`, а удаляются (извлекаются) из стека инструкцией `pop`. При каждом проталкивании (`push`) данных стек растет, при каждом выталкивании (`pop`) стек уменьшается. Это поведение стека можно проверить, наблюдая за регистром указателя стека `esp`, который указывает на вершину стека (в действительности на «дно», т. е. нижнюю границу, так как стек растет вниз).

Стек можно использовать как временное хранилище для записи значений из регистров и последующего возврата в регистры или, что более важно, для передачи значений в функции. Более подробно функции или процедуры будут рассматриваться несколько позже.

В примере исходного кода в листинге 10.1 стек используется для реверсирования строки, т. е. для изменения порядка букв в строке на обратный.

Листинг 10.1. Программа *stack.asm*

```
; stack.asm
extern printf
section .data
    strng      db    "ABCDE",0
    strngLen   equ    $ - strng-1 ; stringlength without 0
    fmt1       db    "The original string: %s",10,0
    fmt2       db    "The reversed string: %s",10,0
section .bss
section .text
    global main
main:
```

```

push rbp
mov rbp, rsp
; Вывод исходной строки.
mov rdi, fmt1
mov rsi, strng
mov rax, 0
call printf
; Проталкивание (push) строки символ за символом в стек.
xor rax, rax
mov rbx, strng ; Адрес строки strng в регистре rbx.
mov rcx, strnglen ; Длина строки в регистре rcx – счетчик.
mov r12, 0 ; Использовать регистр r12 как указатель.
pushLoop:
mov al, byte [rbx+r12] ; Запись символа в регистр rax.
push rax ; Проталкивание (push) значения rax в стек.
inc r12 ; Увеличение указателя на символ на 1
loop pushLoop ; Продолжение цикла.
; Выталкивание (pop) строки символ за символом из стека.
; Это позволяет реверсировать исходную строку.
mov rbx, strng ; Адрес строки strng в регистре rbx.
mov rcx, strnglen ; Длина строки в регистре rcx – счетчик.
mov r12, 0 ; Использовать регистр r12 как указатель.
popLoop:
pop rax ; Выталкивание (pop) символа из стека.
mov byte [rbx+r12], al ; Запись символа в строку strng.
inc r12 ; Увеличение указателя на символ на 1.
loop popLoop ; Продолжение цикла.
mov byte [rbx+r12], 0 ; Завершение строки значением 0.
; Вывод реверсированной строки.
mov rdi, fmt2
mov rsi, strng
mov rax, 0
call printf
mov rsp, rbp
pop rbp
ret

```

На рис. 10.1 показан вывод этой программы.

```

jo@UbuntuDesktop:~/Desktop/linux64/gcc/12 stack$ make
nasm -f elf64 -g -F dwarf stack.asm -l stack.lst
gcc -o stack stack.o
jo@UbuntuDesktop:~/Desktop/linux64/gcc/12 stack$ ./stack
The original string: ABCDE
The reversed string: EDCBA
jo@UbuntuDesktop:~/Desktop/linux64/gcc/12 stack$ █

```

Рис. 10.1. Реверсирование строки

Прежде всего следует отметить, что для вычисления длины строки ее размер был уменьшен на 1, чтобы не учитывать завершающий 0. Если этого не сделать, то реверсированная строка будет начинаться с 0. Затем исходная строка выводится со следующим за ней символом перехода на новую строку. Для записи символов в стек будет использоваться регистр `rax`, поэтому необходимо предварительно инициализировать `rax` нулями с помощью инструкции `xor`.

Адрес строки передается в регистр `gbx`, далее будет применяться инструкция `loop`, поэтому в регистре `gsx` устанавливается значение длины строки. Затем начинается цикл с поочередной записью символов (по одному символу на каждой итерации цикла) в стек, начиная с первого символа. Символ (байт) записывается в регистр `al`. После этого содержимое `gbx` проталкивается в стек. При каждом использовании инструкции `push` в стек перемещается 8 байт. Если бы регистр `gbx` не был предварительно инициализирован нулями, то, возможно, в нем содержались бы некоторые значения в старших байтах, и запись этих байтов в стек привела бы к непредсказуемому результату. После этой операции стек содержит переданный символ и дополнительные нулевые биты во всех разрядах, которые старше `al`.

После завершения цикла `loop` последний символ строки находится на «вершине» стека, которая в действительности является самым младшим адресом этой «сосульки», потому что стек растет вниз. Начинается второй цикл, в котором символы поочередно извлекаются (вытаскиваются) из стека, затем сохраняются в блоке памяти исходной строки один за другим. Обратите внимание: необходим только один байт, поэтому символы извлекаются в регистр `gbx`, но при записи в память используется только регистр `al`.

На рис. 10.2 показана общая схема выполняемого процесса: исходная строка изображена справа, ее символы по одному записываются в стек и добавляются к предыдущему его содержимому. Потом символы извлекаются из стека и записываются обратно по адресу памяти исходной строки, но поскольку стек работает по схеме «последним пришел, первым вышел» (*last in first out* – LIFO), символы располагаются в строке в обратном порядке – строка реверсирована.

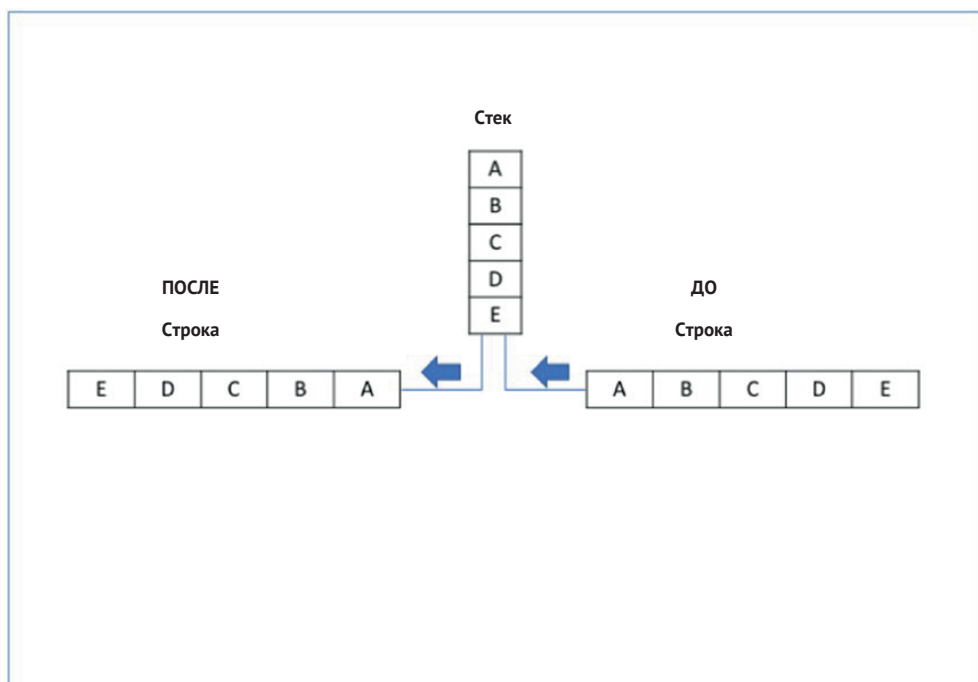


Рис. 10.2. Схема реверсирования строки

Необходимо каким-либо способом отслеживать, что записывается в стек и в каком порядке. Например, при использовании стека для временного хранения содержимого регистров требуется обеспечить извлечение значений регистров в правильном обратном порядке, иначе программа будет работать некорректно или в худшем случае может завершиться аварийно. Таким образом, если запись содержимого регистров в стек выполняется в последовательности, показанной ниже:

```
push eax
push ebx
push ecx
```

то извлекать значения в регистры необходимо в соответствии с правилом «последним пришел, первым вышел» (LIFO), как показано ниже:

```
pop ecx
pop ebx
pop eax
```

Кроме регистров, в стек можно записывать содержимое блоков памяти и непосредственные значения. Извлекать данные из стека можно в регистр или в блок памяти, но не в непосредственное значение, что вполне очевидно.

Об этом полезно знать, однако здесь мы не пользуемся этими возможностями. Если необходимо записывать в стек и извлекать из стека содержимое регистра флагов, то можно воспользоваться инструкциями `pushf` и `popf`.

НАБЛЮДЕНИЕ ЗА СТЕКОМ

Как мы выяснили выше, наблюдение за стеком важно, и наш добрый друг DDD предлагает удобные функциональные возможности для этого. Сначала откройте текстовый редактор с исходным кодом программы и удалите строку, которую добавила среда разработки SASM, затем сохраните файл и закройте редактор. В командной строке выполните сборку программы, потом введите следующую команду:

```
ddd stack
```

В меню выберите пункт **Data** (Данные) → **Status Displays** (Показать состояние) и выполняйте прокрутку в окне, пока не найдете строку параметра **Backtrace of the stack** (Обратная трассировка стека) – ее необходимо активизировать. Установите точку останова, например, на метке `main:`, затем щелкните по кнопке **Run** (Пуск) в перемещаемой панели. После этого начните процесс отладки и пошагово перемещайтесь по программе с помощью кнопки **Next** (Следующий шаг) (нет необходимости построчно проходить по функции `printf`). Наблюдайте в верхнем окне, как выводится и обновляется содержимое стека. Не обращайте внимания на вывод начального содержимого стека. При переходе к инструкции, следующей за инструкцией `push`, вы увидите, что символы помещены в стек и отображены в десятичном представлении ASCII

(41, 42 и т. д.). Наблюдайте за уменьшением стека во время выполнения второго цикла. Это простой способ наблюдения за тем, что находится в стеке и в каком порядке.

На рис. 10.3 показано, как это выглядит в окне отладчика DDD.

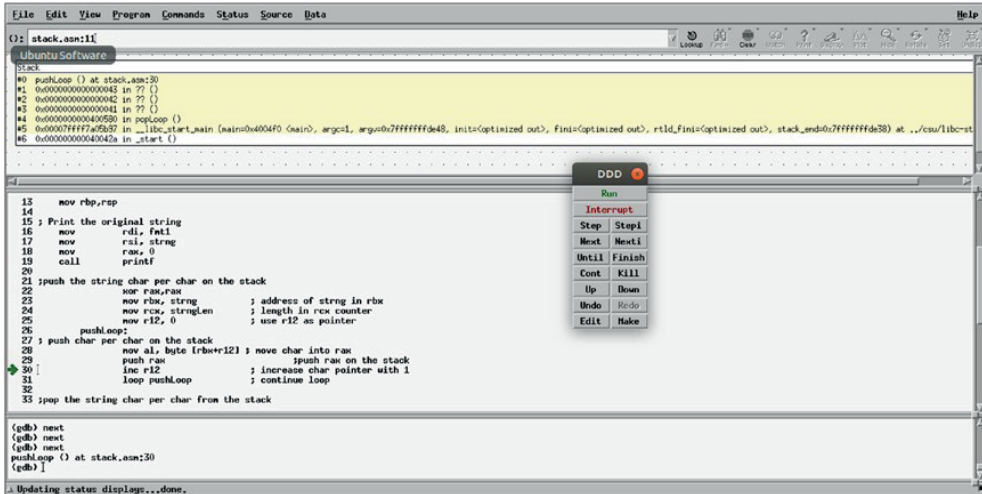


Рис. 10.3. Стек в окне отладчика DDD

Как уже было отмечено ранее, DDD – это довольно-таки старая программа с открытым исходным кодом. Нет никакой уверенности в том, что в будущем она будет продолжать работать корректно, но сейчас DDD делает то, что нам нужно, хотя и выглядит не слишком элегантно.

По справедливости говоря, SASM тоже можно заставить показывать содержимое стека, но для этого потребуется дополнительная ручная работа. Вот как это делается: сначала напомним, что во время отладки в SASM можно выводить значения переменных в памяти, а стек – это просто список блоков памяти с регистром `rsp`, указывающим на самый нижний блок. Таким образом, необходимо убедить SASM показать то, что находится по адресу, хранящемуся в `rsp`, и в блоках памяти с более старшими адресами. На рис. 10.4 показан пример содержимого окна **Memory** (Память) в SASM, где отображено содержимое стека.

File Edit Build Debug Settings Help			
Memory			
Variable or expression	Value	Type	
\$rsp	{68'D',0'000',0'000',0'000',0'000',0'000',0'000',0'000'}	Char b 8	<input checked="" type="checkbox"/> Address
\$rsp+8	{67'C',0'000',0'000',0'000',0'000',0'000',0'000',0'000'}	Char b 8	<input checked="" type="checkbox"/> Address
\$rsp+16	{66'B',0'000',0'000',0'000',0'000',0'000',0'000',0'000'}	Char b 8	<input checked="" type="checkbox"/> Address
\$rsp+24	{65'A',0'000',0'000',0'000',0'000',0'000',0'000',0'000'}	Char b 8	<input checked="" type="checkbox"/> Address
\$rsp+32	{8'370',35'335',1'377',1'377',127'177',0'000',0'000'}	Char b 8	<input checked="" type="checkbox"/> Address
Add variable...		Smart d Array size	<input type="checkbox"/> Address

Рис. 10.4. Содержимое стека в окне SASM

Ссылка на регистр `rsp` выглядит как `$rsp`. Адрес стека каждый раз увеличивается на 8 (`$rsp + 8`), потому что при выполнении каждой инструкции записи `push` в стек передается 8 байт. Для столбца `Type` (Тип) задано значение `Char` (Символ), `b` (байты) и размер 8, а также активизирован атрибут `Address` (Адрес). Тип символов (`Char`) выбран потому, что в стек записывается строка, и в таком представлении ее проще читать, а подтип «байты» выбран потому, что нас интересуют значения байтов (при каждой операции передачи в стек регистр `al` содержит 1 байт), а на каждой итерации цикла в стек записывается 8 байт. Регистр `rsp` содержит адрес (активизирован атрибут `Address`). Выполняйте пошаговый проход по программе и наблюдайте, как изменяется содержимое стека.

Этот прием работает, но для наблюдения все характеристики и атрибуты каждого блока памяти стека необходимо устанавливать вручную – это может оказаться слишком утомительным, если используется большой стек и/или большое количество переменных в памяти, требуемых для наблюдения за содержимым стека.

РЕЗЮМЕ

В этой главе вы узнали:

- что стек начинается со старших адресов памяти и растет в направлении уменьшения адресов;
- что инструкция записи в стек `push` уменьшает указатель стека `rsp`;
- что инструкция извлечения из стека `pop` увеличивает указатель стека `rsp`;
- что инструкции `push` и `pop` работают во взаимно обратном порядке;
- как использовать отладчик DDD для наблюдения за стеком;
- как использовать интегрированную среду разработки SASM для наблюдения за стеком.

Глава 11

Арифметика с плавающей точкой

Вы уже знаете о целочисленной арифметике, а сейчас мы рассмотрим некоторые основы вычислений с плавающей точкой. В этом нет ничего сложного, число с плавающей точкой содержит десятичную точку и ноль или более десятичных знаков после точки. Здесь будут рассматриваться два типа чисел с плавающей точкой: с обычной (одиночной) точностью и с двойной точностью. Числа с двойной точностью более верны, потому что позволяют работать с большим количеством значимых разрядов. После получения этой информации вы обладаете знаниями, достаточными для выполнения и анализа примеров программ в текущей главе.

СРАВНЕНИЕ ЧИСЕЛ С ОБЫЧНОЙ И ДВОЙНОЙ ТОЧНОСТЬЮ

Немного истории для тех, кто ею интересуется.

Число с обычной (одиночной) точностью (single precision) хранится в 32 битах: 1 знаковый бит, 8 бит показателя степени и 23 бита дробной части.

S	EEEEEEEE	FFFFFFFFFFFFFFFFFFFFFF
0	1 8	9 31

Число с двойной точностью (double precision) хранится в 64 битах: 1 знаковый бит, 11 бит показателя степени и 52 бита дробной части.

S	EEEEEEEEEE	FFFFFFFFFFFF.....FFFFFFFF
0	1 11	12 63

Знаковый бит объясняется просто: если число положительное, то этот бит равен 0, если число отрицательное, то этот бит равен 1.

Смысл битов показателя степени объяснить немного сложнее. Рассмотрим пример с десятичными числами.

$$200 = 2.0 \times 10^2$$
$$5000.30 = 5.0003 \times 10^3$$

Теперь пример с двоичными числами:

$$1101010.01011 = 1.0101001011 \times 2^6 \text{ (дробная точка перемещена на 6 позиций влево)}$$

Но показатель степени может быть положительным, отрицательным или равняться нулю. Чтобы сделать эти различия ясными, для чисел с обычной точностью к положительному показателю степени прибавляется число 127 перед сохранением. Это означает, что нулевой показатель степени будет сохранен как 127. Это число 127 называется смещением (bias). Для чисел с двойной точностью смещение равно 1023.

В приведенном выше примере число 1.0101001011 называется мантиссой (significand; mantissa). Предполагается, что первый бит мантиссы всегда равен 1 (т. е. мантисса «нормализована»), поэтому он не сохраняется.

Ниже приводится простой пример, показывающий, как это работает. Для проверки результата проводимого здесь эксперимента воспользуйтесь, например, этим интернет-ресурсом: <https://babbage.cs.qc.cuny.edu/IEEE-754/>.

Пример с десятичным числом с обычной точностью 10:

- десятичное число 10 имеет вид 1010 как двоичное целое число;
- знаковый бит равен 0, потому что число положительное;
- получено число в формате b.bbbb. Здесь 1.010 – мантисса с первой 1 в соответствии с требованиями. Самая первая 1 не будет сохранена;
- следовательно, показатель степени равен 3, потому что дробная точка была перемещена на три позиции влево. Показатель степени положительный, к нему прибавляется 127, в результате показатель степени равен 130, или в двоичном формате 10000010;
- таким образом, десятичное число с обычной точностью 10 будет сохранено в следующем виде:

0 10000010	010000000000000000000000
S EEEEEEE	FFFFFFFFFFFFFFFFFFFFFFFF

или 41200000 в шестнадцатеричном формате.

Следует отметить, что шестнадцатеричное представление значения обычной точности отличается от того же значения, представленного с двойной точностью. Почему бы не использовать всегда числа с двойной точностью и получать соответствующие преимущества? Вычисления с двойной точностью медленнее вычислений с обычной точностью, а операнды с двойной точностью требуют больше памяти.

Если все вышеизложенное кажется вам сложным, то вы правы. Попробуйте найти в интернете подходящее инструментальное средство для выполнения или, по крайней мере, для проверки выполненных преобразований.

В более старых программах вам могут встретиться 80-битовые числа с плавающей точкой, но для таких чисел существуют собственные специализированные инструкции, обозначаемые как FPU-инструкции. Эта функциональность представляет собой наследие прошлого и не рекомендуется к использованию.

в новых разработках. Тем не менее в статьях в интернете время от времени вы будете обнаруживать FPU-инструкции.

Теперь приступим к выполнению более интересных действий.

КОДИРОВАНИЕ С ПРИМЕНЕНИЕМ ЧИСЕЛ С ПЛАВАЮЩЕЙ ТОЧКОЙ

В листинге 11.1 показан пример программы, использующей числа с плавающей точкой.

Листинг 11.1. Программа *fcalc.asm*

```
; fcalc.asm
extern printf
section .data
    number1    dq    9.0
    number2    dq    73.0
    fmt        db    "The numbers are %f and %f",10,0
    fmtfloat    db    "%s %f",10,0
    f_sum       db    "The float sum of %f and %f is %f",10,0
    f_dif       db    "The float difference of %f and %f is %f",10,0
    f_mul       db    "The float product of %f and %f is %f",10,0
    f_div       db    "The float division of %f by %f is %f",10,0
    f_sqrt      db    "The float squareroot of %f is %f",10,0
section .bss
section .text
    global main
main:
    push    rbp
    mov     rbp,rbp
; Вывод чисел.
    movsd   xmm0, [number1]
    movsd   xmm1, [number2]
    mov     rdi,fmt
    mov     rax,2          ; Два числа с плавающей точкой.
    call    printf
; Вычисление суммы.
    movsd   xmm2, [number1] ; Число с плавающей точкой двойной точности в xmm.
    addsd   xmm2, [number2] ; Сложение с другим числом двойной точности, результат в xmm.
    ; Вывод результата.
    movsd   xmm0, [number1]
    movsd   xmm1, [number2]
    mov     rdi,f_sum
    mov     rax,3          ; Три числа с плавающей точкой.
    call    printf
; Вычисление разности.
    movsd   xmm2, [number1] ; Число с плавающей точкой двойной точности в xmm.
    subsd   xmm2, [number2] ; Вычитание другого числа двойной точности из xmm.
    ; Вывод результата.
    movsd   xmm0, [number1]
    movsd   xmm1, [number2]
    mov     rdi,f_dif
    mov     rax,3          ; Три числа с плавающей точкой.
    call    printf
```

```

; Умножение.
movsd xmm2, [number1] ; Число с плавающей точкой двойной точности в xmm.
mulsd xmm2, [number2] ; Умножение заданного числа на xmm.
; Вывод результата.
mov rdi, f_mul
movsd xmm0, [number1]
movsd xmm1, [number2]
mov rax, 3 ; Три числа с плавающей точкой.
call printf

; Деление.
movsd xmm2, [number1] ; Число с плавающей точкой двойной точности в xmm.
divsd xmm2, [number2] ; Деление xmm0.
; Вывод результата.
mov rdi, f_div
movsd xmm0, [number1]
movsd xmm1, [number2]
mov rax, 1 ; Одно число с плавающей точкой.
call printf

; Вычисление квадратного корня.
sqrtsd xmm1, [number1] ; Квадратный корень двойной точности в xmm.
; Вывод результата.
mov rdi, f_sqrt
movsd xmm0, [number1]
mov rax, 2 ; Два числа с плавающей точкой.
call printf

; Выход из программы.
mov rsp, rbp
pop rbp ; Операция, обратная операции push в начале программы.
ret

```

Это простая программа – в действительности организация вывода требует больше усилий, чем вычисления с плавающей точкой.

Используйте отладчик для пошагового прохода по программе и наблюдайте за содержимым регистров и блоков памяти. Например, обратите внимание на то, как числа 9.0 и 73.0 сохраняются в блоках памяти с адресами `number1` и `number2` – это значения с плавающей точкой двойной точности.

Следует напомнить, что при отладке в SASM регистры `xmm` расположены в нижней части окна регистров, в крайней левой области регистров `ymm`.

Инструкция `movsd` означает «move a double precision floating point value» (переместить значение с плавающей точкой двойной точности). Существует также аналогичная инструкция `movss` для чисел с обычной точностью. Арифметические инструкции для чисел с обычной точностью: `addss`, `subss`, `mulss`, `divss` и `sqrts`.

Все прочее должно быть абсолютно понятно на текущий момент. На рис. 11.1 показан вывод этой программы.

```
jo@UbuntuDesktop:~/Desktop/linux64/gcc/13 fcalc$ make
nasm -f elf64 -g -F dwarf fcalc.asm -l fcalc.lst
gcc -o fcalc fcalc.o
jo@UbuntuDesktop:~/Desktop/linux64/gcc/13 fcalc$ ./fcalc
The numbers are 9.000000 and 73.000000
The float sum of 9.000000 and 73.000000 is 82.000000
The float difference of 9.000000 and 73.000000 is -64.000000
The float product of 9.000000 and 73.000000 is 657.000000
The float division of 9.000000 by 73.000000 is 0.123288
The float squareroot of 9.000000 is 3.000000
jo@UbuntuDesktop:~/Desktop/linux64/gcc/13 fcalc$ █
```

Рис. 11.1. Вывод программы *fcalc.asm*

Теперь, когда вы знаете, как используется стек, попробуйте сделать следующее: закомментируйте инструкцию `push rbp` в начале программы и инструкцию `pop rbp` в конце. Выполните сборку, запустите программу и наблюдайте, что происходит: программа «падает» (завершается аварийно). Причина этого аварийного завершения станет понятной немного позже, но следует отметить, что она связана с выравниванием в стеке.

РЕЗЮМЕ

В этой главе вы узнали:

- об основах использования регистров `xmm` для вычислений с плавающей точкой;
- о различиях между числами с обычной (одиночной) точностью и числами с двойной точностью;
- об инструкциях `movsd`, `addsd`, `subsd`, `mulsd`, `divsd` и `sqrtsd`.

Глава 12

Функции

Ассемблер не является «структурным языком». Взгляните на многочисленные инструкции `jmp` и метки, которые позволяют при выполнении программы куда-то переходить и возвращаться обратно. В современных языках программирования высокого уровня существуют такие программные конструкции, как `do...while`, `while...do`, `case` и т. п. Подобных конструкций нет в языке ассемблера. Но, как и в современных языках программирования, в языке ассемблера имеются функции и процедуры, которые помогают сделать исходный код более структурированным. Небольшое уточнение: функция выполняет инструкции и возвращает некоторое значение. Процедура выполняет инструкции, но не возвращает значение.

В этой книге мы уже использовали функции, точнее применяли внешнюю функцию с именем `printf` из стандартной библиотеки языка C. В данной главе представлены простые функции, в следующих главах будут рассматриваться важные подробности использования функций, такие как выравнивание стека, внешние функции и соглашения о вызове функций.

Создание простой функции

В листинге 12.1 показан пример программы на ассемблере с простой функцией, вычисляющей площадь круга.

Листинг 12.1. Программа *function.asm*

```
; function.asm
extern printf
section .data
    radius    dq    10.0
    pi        dq    3.14
    fmt       db    "The area of the circle is %.2f",10,0
section .bss
section .text
    global main
;-----
main:
push    rbp
mov     rbp, rsp
call    area                ; Вызов функции.
mov     rdi,fmt              ; Формат вывода.
```

```

    movsd xmm1, [radius]    ; Запись числа с плав.точкой в регистр xmm1.
    mov    rax,1            ; Значение площади в регистре xmm0.
    call   printf
leave
ret
;-----
area:
push    rbp
mov     rbp, rsp
    movsd xmm0, [radius]    ; Запись числа с плав.точкой в регистр xmm0.
    mulsd xmm0, [radius]    ; Умножение xmm0 на число с плав.точкой.
    mulsd xmm0, [pi]        ; Умножение xmm0 на (другое) число с плав.точкой.
leave
ret

```

На рис. 12.1 показан вывод этой программы.



```

jo@UbuntuDesktop:~/Desktop/linux64/gcc/14 function$ make
nasm -f elf64 -g -F dwarf function.asm -l function.lst
gcc -o function function.o
jo@UbuntuDesktop:~/Desktop/linux64/gcc/14 function$ ./function
The area of the circle is 314.00
jo@UbuntuDesktop:~/Desktop/linux64/gcc/14 function$ █

```

Рис. 12.1. Вывод программы *function.asm*

В этой программе имеется основная часть, определяемая, как и ранее, меткой `main`, а за ней следует функция, определяемая меткой `area`. В основной части `main` вызывается функция `area`, вычисляющая площадь круга с использованием `radius` и `pi` – переменных, значения которых хранятся в блоках памяти. Здесь можно видеть, что функции обязательно должны иметь пролог и эпилог, так же, как и основная часть `main`.

Вычисленная площадь круга сохраняется в регистре `xmm0`. После возврата из функции в основную часть `main` вызывается функция `printf` с регистром `rax`, содержащим 1, – это означает, что один `xmm`-регистр должен быть выведен. Здесь вводится новая инструкция `leave`, которая выполняет те же действия, что и `mov rsp, rbp` и `pop rbp` (эпилог).

Если из функции возвращается значение, то для чисел с плавающей точкой используется регистр `xmm0`, а для прочих значений, таких как целые числа и адреса, используется регистр `rax`. Аргументы функции `pi` и `radius` размещены в памяти. В рассматриваемом здесь примере это приемлемый вариант, но все же лучше применять регистры и стек для хранения аргументов функции. Использование переменных памяти для передачи значений в функцию может привести к конфликту имен для значений, используемых в основной части `main` и в функциях, а кроме того, может сделать код менее «переносимым».

ЕЩЕ О ФУНКЦИЯХ

Рассмотрим некоторые характеристики функций на другом примере, показанном в листинге 12.2.

Листинг 12.2. Программа *function2.asm*

```

; function2.asm
extern printf
section .data
    radius      dq    10.0
section .bss
section .text
;-----
area:
    section .data
        .pi dq    3.141592654      ; Локальная переменная в функции area.
    section .text
push    rbp
mov     rbp, rsp
movsd   xmm0, [radius]
mulsd   xmm0, [radius]
mulsd   xmm0, [.pi]

leave
ret
;-----
circum:
section .data
    .pi dq    3.14      ; Локальная переменная в функции circum.
section .text
push    rbp
mov     rbp, rsp
movsd   xmm0, [radius]
addsd   xmm0, [radius]
mulsd   xmm0, [.pi]

leave
ret
;-----
circle:
section .data
    .fmt_area db    "The area is %f",10,0
    .fmt_circum db    "The circumference is %f",10,0
section .text
push    rbp
mov     rbp, rsp
call    area
mov     rdi,.fmt_area
mov     rax,1      ; Значение площади в регистре xmm0.
call    printf
call    circum
mov     rdi,.fmt_circum
mov     rax,1      ; Значение длины окружности в регистре xmm0.
call    printf

leave
ret
;-----
global main
main:
push    rbp
mov     rbp, rsp
call    circle

leave
ret

```


В этой программе основная часть `main` вызывает функцию `circle`, которая в свою очередь вызывает функции `area` и `circum`. Таким образом, функции могут вызывать другие функции. В действительности основная часть `main` – это тоже функция, вызывающая другие функции. Но помните о том, что функции не могут быть вложенными, т. е. функции не должны содержать код других функций.

Функции также могут иметь собственные разделы, такие как `.data`, `.bss` и `.text`. А что означает точка перед `pi` и перед переменными `fmt`? Точка обозначает локальную переменную, т. е. переменную, известную только внутри функции, в которой эта переменная объявлена. В функции `area` используется значение `pi`, отличающееся от значения `pi` в функции `circum`. Переменная `radius`, объявленная в разделе `section .data` основной функции `main`, известна в каждой функции в этом листинге исходного кода, включая саму функцию `main`. Весьма разумно использовать локальные переменные во всех случаях, когда это возможно, – это снижает риск возникновения конфликта имен переменных.

На рис. 12.2 показан вывод этой программы.

```
jo@UbuntuDesktop:~/Desktop/linux64/gcc/15 function2$ make
nasm -f elf64 -g -F dwarf function2.asm -l function2.lst
gcc -g -o function2 function2.o
jo@UbuntuDesktop:~/Desktop/linux64/gcc/15 function2$ ./function2
The area is 314.16
The circumference is 62.80
jo@UbuntuDesktop:~/Desktop/linux64/gcc/15 function2$ █
```

Рис. 12.2. Вывод программы *function2.asm*

РЕЗЮМЕ

В этой главе вы узнали:

- как использовать функции;
- что функции могут иметь собственные разделы `section .data` и `section .bss`;
- что функции не могут быть вложенными;
- что функции могут вызывать другие функции;
- что `main` – это всего лишь одна из функций;
- как использовать локальные переменные.

Глава 13

Выравнивание стека и фрейм стека

Когда основная программа вызывает функцию, 8-байтовый адрес возврата записывается в стек. Этот 8-байтовый адрес является адресом инструкции, которая должна быть выполнена после выхода из функции. Поэтому после завершения выполнения функции программа находит адрес возврата в стеке и продолжает работу после вызова функции. Внутри функции также может использоваться стек для различных целей. Каждый раз, когда в стек записывается какое-либо значение, указатель стека уменьшается на 8 байт, а при каждом извлечении значения из стека указатель стека увеличивается на 8 байт. Поэтому необходимо обеспечить «восстановление» в стеке корректного значения перед выходом из функции. Если этого не сделать, то вызывающая программа получит неправильный адрес инструкции, которая должна выполняться после вызова функции.

ВЫРАВНИВАНИЕ СТЕКА

В руководствах Intel вы обнаружите упоминание о следующем обязательном требовании: стек непременно должен выравниваться по 16-байтовой границе при каждом вызове функции. Это требование может показаться немного странным, ведь стек создается в памяти с 8-байтовыми (64-битовыми) границами. Причина данного требования состоит в том, что существуют SIMD-инструкции, которые выполняют параллельные операции с более крупными блоками данных, и для этих SIMD-инструкций может потребоваться размещение данных по адресам, кратным 16 байтам. В предыдущих примерах при использовании функции `printf` с `xmm`-регистрами выполнялось выравнивание стека по 16-байтовой границе, но это действие не объяснялось. Вернемся в главу 11, где рассматривалась арифметика с плавающей точкой. В этой главе был приведен пример с аварийным завершением программы после комментирования инструкций `push rbp` и `pop rbp`. Эта программа завершалась аварийно, потому что удаление вышеупомянутых инструкций приводило к тому, что стек не был выровнен по 16-байтовой границе. Если вы используете функцию `printf` без `xmm`-регистров, то можете не обращать внимания на требование выравнивания стека, но однажды вдруг обнаружите, что в программе возникают непонятные ошибки.

SIMD-инструкции и операции выравнивания стека будут рассматриваться в следующих главах, поэтому не беспокойтесь, если приведенное выше описание оказалось не совсем понятным. Сейчас просто следует запомнить, что при вызове функции необходимо выровнять стек по адресам, кратным 16 байтам.

С точки зрения процессора `main` – одна из обычных функций. Прежде чем программа начнет выполняться, происходит выравнивание стека. Непосредственно перед началом выполнения `main` 8-байтовый адрес возврата записывается в стек, т. е. стек не выровнен перед началом выполнения `main`. Если не происходит обращений к стеку в интервале между началом `main` и вызовом какой-либо функции, то указатель стека `rsp` не выровнен по 16-байтовой границе. Это можно проверить, посмотрев содержимое регистра `rsp`: если значение в `rsp` заканчивается нулем, то стек выровнен по 16-байтовой границе. Чтобы обеспечить это нулевое завершение, нужно записать что-нибудь в стек, чтобы выровнять его по 16-байтовой границе. Разумеется, не следует забывать в дальнейшем выполнить соответствующую инструкцию `pop`.

Требование выравнивания стека является одной из причин использования пролога и эпилога функции. Первая инструкция в `main` и в любой другой функции должна записывать что-либо в стек для его выравнивания. Это обоснование применения в прологе инструкции `push rbp`. Регистр `rbp` также называется указателем базы (*base pointer*).

Почему используется именно `rbp`? В прологе, когда используются фреймы стека (их описание будет приведено немного позже), регистр `rbp` изменяется, поэтому перед использованием `rbp` в фрейме стека его значение записывается в стек для сохранения и восстановления при возврате. Даже если фрейм стека не создается, регистр `rbp` является идеальным кандидатом для выравнивания стека, поскольку не используется для передачи аргументов в функцию. Передача аргументов будет рассматриваться немного позже в этой главе. В прологе также используется инструкция `mov rbp, rsp`. Эта инструкция сохраняет регистр `rsp` – указатель стека, содержащий необходимый адрес возврата. Инструкции пролога выполняются с обратным смыслом и в обратном порядке в эпилоге, разумеется, в этом случае лучше всего не трогать `rbp`. В следующих главах мы рассмотрим несколько других методов выравнивания стека.

В листинге 13.1 показан исходный код для некоторых экспериментов со стеком. Наблюдайте за регистром `rsp` при отладке и выполните пошаговый проход по программе с помощью `SASM`. Уберите символы комментариев из инструкций `push rbp` и `pop rbp` и посмотрите, что происходит. Если программа подходит к вызову функции `printf` без выравнивания стека, то происходит аварийное завершение (программа «падает»). Причина в том, что функция `printf` непременно требует выравнивания стека.

В этой программе не используется полноценный пролог и эпилог, т. е. не создаются фреймы стека. Применяются только инструкции `push` и `pop` для демонстрации выравнивания стека.

Листинг 13.1. Программа aligned.asm

```

; aligned.asm
extern printf
section .data
    fmt db      "2 times pi equals %.14f",10,0
    pi  dq      3.14159265358979
section .bss
section .text
;-----
func3:
    push  rbp
    movsd xmm0, [pi]
    addsd  xmm0, [pi]
    mov  rdi,fmt
    mov  rax,1
    call printf      ; Вывод числа с плав.точкой.
    pop  rbp
    ret
;-----
func2:
    push  rbp
    call  func3 ; Вызов третьей функции.
    pop  rbp
    ret
;-----
func1:
    push  rbp
    call  func2 ; Вызов второй функции.
    pop  rbp
    ret
;-----
global main
main:
    push  rbp
    call  func1 ; Вызов первой функции.
    pop  rbp
    ret

```

Обратите внимание: если выполняется определенное количество вызовов (четное или нечетное в зависимости от того, как вы начинаете программу), то стек будет выровнен по 16-байтовой границе, даже без применения инструкций push и pop, и в программе не возникает критический сбой.

БОЛЕЕ ПОДРОБНО О ФРЕЙМАХ СТЕКА

Можно различать два типа функций: функции-ветви и функции-листья. Функции-ветви содержат вызовы других функций, а функции-листья выполняют некоторые команды, затем возвращаются в родительскую функцию без вызова какой-либо другой функции.

Теоретически при каждом вызове функции необходимо создавать фрейм стека (stack frame). Это делается следующим образом: в вызываемой функции в первую очередь выравнивается стек по 16-байтовой границе, т. е. выполня-

ется инструкция `push gbr`. Затем содержимое указателя стека `gsp` сохраняется в регистре `gbr`. При выходе из функции восстанавливается содержимое указателя стека `gsp` (из регистра `gbr`) и выполняется инструкция `pop gbr` для восстановления `gbr`. В этом и заключается роль пролога и эпилога функции. Теперь внутри функции регистр `gbr` служит точкой привязки для исходной локации стека. Каждый раз, когда функция вызывает другую функцию, новая (вызываемая) функция должна создавать собственный фрейм стека.

В функциях-листьях, вообще говоря, можно не уделять внимания выравниванию стека и созданию фрейма стека – в этом нет необходимости, если вы не работаете со стеком. Следует отметить, что при вызове, например, `printf` вызываемая функция не является функцией-листом. Кроме того, если функция не использует SIMD-инструкции, то нет необходимости заботиться о выравнивании стека.

Компиляторы поддерживают функции оптимизации, и иногда при просмотре сгенерированного ими кода можно заметить, что фрейм стека не используется. Это происходит, когда компилятор в процессе оптимизации обнаруживает, что фрейм стека не нужен.

В любом случае лучше выработать правильную привычку всегда включать в функцию создание фрейма стека и выполнять выравнивание стека – это поможет избежать множества проблем в дальнейшем. Весомой причиной включения фрейма стека является тот факт, что GDB и основанные на GDB отладчики (такие как DDD и SASM) предполагают наличие фрейма стека. Если в коде не выполнено создание фрейма стека, то отладчик поведет себя непредсказуемо, например будет игнорировать точки останова или пропускать инструкции. В одном из примеров исходного кода из предыдущей главы (например, *alife.asm*) прокомментируйте пролог и эпилог функции, затем запустите GDB и понаблюдайте за происходящим.

В качестве дополнительного упражнения возьмите исходный код из предыдущей главы (например, *function2.asm*) и в отладчике SASM или GDB наблюдайте за тем, как стек остается выровненным во время выполнения.

Дополнительный упрощенный прием: пролог функции можно заменить на инструкцию `enter 0, 0`, а эпилог функции на инструкцию `leave`. Но для `enter` характерна низкая производительность, поэтому лучше продолжать заменять инструкции `push gbr` и `mov gbr, gsp`, если производительность важна. Для инструкции `leave` проблемы производительности не существует.

РЕЗЮМЕ

В этой главе вы узнали:

- что такое выравнивание стека;
- как использовать фреймы стека;
- как использовать SASM для проверки указателя стека;
- об инструкциях входа и выхода из функций.

Глава 14

Внешние функции

Нам уже известно, как создавать и использовать функции в исходном коде. Но функции не обязательно должны размещаться в том же файле, в котором находится основная программа. Можно написать и ассемблировать функции в отдельном файле и связывать их во время сборки программы. Функция `printf`, которую мы уже использовали несколько раз, представляет собой пример внешней функции. В файле исходного кода, где планируется применение внешней функции, необходимо объявить ее с помощью ключевого слова `extern`, чтобы сообщить ассемблеру о том, что исходный код этой функции не следует искать в текущем файле. Ассемблер предполагает, что эта функция уже ассемблирована в некотором объектном файле. Внешняя функция будет вставлена линкером, если он сможет найти ее в заданном объектном файле.

Можно использовать внешние функции из языка C, но вы также можете создать собственный набор внешних функций и при необходимости связывать их с основной программой.

СОЗДАНИЕ И СВЯЗЫВАНИЕ ФУНКЦИЙ

В листинге 14.1 показан пример программы, состоящей из трех файлов исходного кода с именами: *function4.asm*, *circle.asm* и *rect.asm*. Для этой программы написан новый *makefile*. Внимательно изучите содержимое этих файлов.

Листинг 14.1. Программа *function4.asm*

```
; function4.asm
extern printf
extern c_area
extern c_circum
extern r_area
extern r_circum
global pi
section .data
    pi          dq    3.141592654
    radius      dq    10.0
    side1       dq    4
    side2       dq    5
    fmtf        db    "%s %f",10,0
    fmti        db    "%s %d",10,0
    ca          db    "The circle area is ",0
```

```
cc    db    "The circle circumference is ",0
ra    db    "The rectangle area is ",0
rc    db    "The rectangle circumference is ",0
section .bss
section .text
    global main
main:
push    rbp
mov     rbp, rsp
; Площадь круга.
    movsd xmm0, qword [radius] ; Аргумент radius в регистре xmm0.
    call  c_area                ; Площадь возвращается в регистре xmm0.
; Вывод площади круга.
    mov     rdi, fmtf
    mov     rsi, ca
    mov     rax, 1
    call    printf
; Длина окружности.
    movsd   xmm0, qword [radius] ; Аргумент radius в регистре xmm0.
    call    c_circum              ; Длина окружности в регистре xmm0.
; Вывод длины окружности.
    mov     rdi, fmtf
    mov     rsi, cc
    mov     rax, 1
    call    printf
; Площадь прямоугольника.
    mov     rdi, [side1]
    mov     rsi, [side2]
    call    r_area                ; Площадь возвращается в регистре rax.
; Вывод площади прямоугольника.
    mov     rdi, fmti
    mov     rsi, ra
    mov     rdx, rax
    mov     rax, 0
    call    printf
; Периметр прямоугольника.
    mov     rdi, [side1]
    mov     rsi, [side2]
    call    r_circum              ; Периметр в регистре rax.
; Вывод периметра прямоугольника.
    mov     rdi, fmti
    mov     rsi, rc
    mov     rdx, rax
    mov     rax, 0
    call    printf
mov     rsp, rbp
pop     rbp
ret
```

В исходном коде в листинге 14.1 несколько функций объявлено как `extern` (внешние), как это ранее делалось неоднократно при использовании функции `printf`. В этом нет ничего нового. Но, кроме того, переменная `pi` объявлена как `global`. Таким образом, эта переменная будет доступной во внешних функциях.

В листингах 14.2 и 14.3 показаны отдельные файлы, содержащие только внешние функции.

Листинг 14.2. Файл *circle.asm*

```

; circle.asm
extern pi
section .data
section .bss
section .text
;-----
global c_area
c_area:
    section .text
    push rbp
    mov rbp, rsp
        movsd xmm1, qword [pi]
        mulsd xmm0, xmm0      ; Радиус в регистре xmm0.
        mulsd xmm0, xmm1
    mov rsp, rbp
    pop rbp
    ret
;-----
global c_circum
c_circum:
    section .text
    push rbp
    mov rbp, rsp
        movsd xmm1, qword [pi]
        addsd xmm0, xmm0      ; Радиус в регистре xmm0.
        mulsd xmm0, xmm1
    mov rsp, rbp
    pop rbp
    ret

```

Листинг 14.3. Файл *rect.asm*

```

; rect.asm
section .data
section .bss
section .text
;-----
global r_area
r_area:
    section .text
    push rbp
    mov rbp, rsp
        mov rax, rsi
        imul rax, rdi
        mov rsp, rbp
    pop rbp
    ret
;-----
global r_circum
r_circum:
    section .text
    push rbp
    mov rbp, rsp
        mov rax, rsi
        add rax, rdi
        add rax, rax

```



```
mov rsp,rbp
pop rbp
ret
```

В файле *circle.asm* необходимо использовать переменную *pi*, объявленную в основном файле исходного кода как *global*, что необходимо отметить как не самое удачное решение, но здесь оно применено в демонстрационных целях. Глобальные переменные, подобные *pi*, трудно отслеживать, и они могут даже приводить к конфликту переменных с одинаковыми именами. Самое лучшее практическое решение – использование регистров для передачи значений в функцию. Здесь же приходится определять *pi* как внешнюю переменную. Каждый файл *circle.asm* и *rect.asm* содержит две функции: для вычисления периметра и площади. Необходимо объявить, что эти функции являются глобальными (*global*), как и основная функция *main*. При ассемблировании этих функций добавляются необходимые «служебные данные», позволяющие линкеру включать эти функции в другой объектный код.

РАСШИРЕННАЯ ВЕРСИЯ MAKEFILE

Для выполнения всей работы необходима расширенная версия *makefile*, показанная в листинге 14.4.

Листинг 14.4. *makefile*

```
# makefile для function4, circle и rect.
function4: function4.o circle.o rect.o
    gcc -g -o function4 function4.o circle.o rect.o -no-pie
function4.o: function4.asm
    nasm -f elf64 -g -F dwarf function4.asm -l function4.lst
circle.o: circle.asm
    nasm -f elf64 -g -F dwarf circle.asm -l circle.lst
rect.o: rect.asm
    nasm -f elf64 -g -F dwarf rect.asm -l rect.lst
```

Читать *makefile* нужно снизу вверх: сначала различные файлы исходного кода на ассемблере ассемблируются в объектные файлы, затем эти объектные файлы связываются и объединяются в выполняемый файл *function4*. Здесь проявляется истинная мощь утилиты *make*. При любом изменении одного из файлов исходного кода *make* благодаря древовидной структуре точно знает, какие файлы нужно повторно ассемблировать и связать. Разумеется, если функции стабильны и в дальнейшем не будут изменяться, то нет необходимости их повторного ассемблирования при каждом сеансе обработки *makefile*. Просто сохраните объектные файлы в специально созданном каталоге и в дальнейшем обращайтесь к ним по полному путевому имени в строке *gcc* в *makefile*. Объектный файл – это результат ассемблирования или компиляции исходного кода. Объектный файл содержит машинный код и информацию для линкера о глобальных переменных и внешних функциях, необходимую для создания корректного выполняемого файла. В рассматриваемом здесь примере все объектные файлы размещены в том же каталоге, что и основной файл исходного кода, поэтому полные путевые имена не используются.

Что можно сказать о функции `printf`? Почему в *makefile* нет ссылки на `printf`? Потому что компилятор `gcc` обладает достаточным интеллектом, для того чтобы поискать еще и в библиотеках языка C функции, обращения к которым встречаются в исходном коде. Это означает, что вы не должны использовать имена функций языка C для именования своих функций. Это может запутать кого угодно, не говоря уже о линкере.

В приведенном выше коде регистры использовались для передачи значений из основной программы в функции и в обратном направлении – это самый правильный практический подход. Например, перед вызовом `g_area` значение переменной `side1` записывается в регистр `rdi`, а значение `side2` в регистр `rsi`. Затем вычисленная площадь прямоугольника возвращается в регистре `rax`. Для возврата результата можно было бы воспользоваться глобальной переменной, подобной `pi`, объявленной в разделе `section .data` функции `main`. Но, как уже было отмечено выше, этого следует избегать. Более подробно этот вопрос будет обсуждаться в следующей главе о соглашениях при вызовах функций.

На рис. 14.1 показан вывод этой программы.

```
jo@UbuntuDesktop:~/Desktop/linux64/gcc/17 function4$ make
nasm -f elf64 -g -F dwarf function4.asm -l function4.lst
nasm -f elf64 -g -F dwarf circle.asm -l circle.lst
nasm -f elf64 -g -F dwarf rect.asm -l rect.lst
gcc -g -o function4 function4.o circle.o rect.o
jo@UbuntuDesktop:~/Desktop/linux64/gcc/17 function4$ ./function4
The circle area is 314.159265
The circle circumference is 62.831853
The rectangle area is 20
The rectangle circumference is 18
jo@UbuntuDesktop:~/Desktop/linux64/gcc/17 function4$ █
```

Рис. 14.1. Вывод программы *function4*

При использовании этого примера в среде SASM необходимо в первую очередь ассемблировать внешние функции для получения объектных файлов. Затем в диалоговом окне SASM **Settings** (Параметры настройки) на вкладке **Build** (Сборка) нужно добавить место расположения этих объектных файлов в строке **Linking Options** (Параметры связывания). Для рассматриваемого здесь примера строка Linking Options должна выглядеть следующим образом (при вводе будьте внимательны – не вставляйте лишние пробелы в этой строке):

```
$PROGRAM.OBJ$ -g -o $PROGRAM$ circle.o rect.o -no-pie
```

РЕЗЮМЕ

В этой главе вы узнали, как:

- используются внешние функции;
- применяются глобальные переменные;
- используется *makefile* для внешних функций;
- передаются значения в функции и возвращаются из функций.

Глава 15

Соглашения о вызовах функций

Соглашения о вызовах функций описывают, как переменные передаются в функции и возвращаются из функций. Если вы будете использовать только функции, которые написали сами, то нет особого смысла беспокоиться о соглашениях о вызовах. Но если вы пользуетесь функциями из библиотеки языка C, то необходимо знать, в какие регистры требуется записывать значения, с которыми должны работать такие внешние функции. Кроме того, если вы пишете ассемблерные функции для создания библиотеки, предназначенной для использования другими разработчиками, то настоятельно рекомендуется соблюдать некоторые соглашения о регистрах, в которых передаются конкретные аргументы функций. Без соблюдения этих соглашений будут постоянно возникать проблемы и конфликты с передачей аргументов.

Вам уже известно, что при вызове функции `printf` первый аргумент записывается в регистр `rdi`, второй аргумент в регистр `rsi`, а следующий аргумент передается в регистре `xmm0`. Вызывая функцию `printf`, мы соблюдали соглашения о вызове.

Чтобы избежать конфликтов и их последствий в виде критических сбоев программы, опытные программисты разработали соглашения о вызовах функций (calling conventions), стандартизированный способ вызова функций. Это превосходный замысел, но, как можно было ожидать, не все согласны со сторонниками данного подхода, поэтому существует несколько различных соглашений о вызове функций. До настоящего момента в этой книге использовались соглашения о вызове функций System V AMD64 ABI, являющиеся стандартом для платформ Linux. Но существуют также и другие соглашения о вызове функций, заслуживающие внимания: Microsoft x64, используемые для программирования в ОС Windows.

Эти соглашения о вызовах функций позволяют использовать внешние функции, написанные на ассемблере, а также функции, скомпилированные из исходного кода на других языках, таких как C, без необходимости доступа к их исходному коду. Нужно просто записать правильные аргументы в регистры, определенные в соглашении о вызове функций.

Более подробную информацию о соглашениях о вызовах функций System V AMD64 ABI можно найти здесь: <https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf>. Этот документ компании Intel содержит огромный объем подробнейшей информации о прикладном двоичном (би-

нарном) интерфейсе (ABI) System V. В данной главе содержится все, что вам нужно знать для того, чтобы на уровне начинающего вызывать функции стандартным способом.

АРГУМЕНТЫ ФУНКЦИЙ

Вернемся немного назад – к предыдущим файлам исходного кода: при вычислениях характеристик круга использовался регистр `xmm0` для передачи значений с плавающей точкой из программы `main` в функцию `circle`, и тот же регистр `xmm0` использовался для возвращения результата с плавающей точкой, полученного в этой функции, в основную программу. При вычислениях характеристик прямоугольника для передачи целочисленных значений в функцию использовались регистры `rdi` и `rsi`, а целочисленный результат возвращался в регистре `rax`. Этот способ передачи аргументов в функцию и возвращения результата из функции определяется конкретным соглашением о вызовах функций.

Аргументы, не являющиеся значениями с плавающей точкой, такие как целые числа и адреса, передаются в следующем порядке:

- 1-й аргумент передается в регистре `rdi`;
- 2-й аргумент передается в регистре `rsi`;
- 3-й аргумент передается в регистре `rdx`;
- 4-й аргумент передается в регистре `rcx`;
- 5-й аргумент передается в регистре `r8`;
- 6-й аргумент передается в регистре `r9`.

Дополнительные аргументы передаются через стек – обязательно в обратном порядке, чтобы можно было извлечь их в правильном порядке. Например, при передаче 10 аргументов порядок следующий:

- первым в стек записывается 10-й аргумент;
- затем в стек записывается 9-й аргумент;
- потом в стек записывается 8-й аргумент;
- последним в стек записывается 7-й аргумент.

После перехода в функцию необходимо обеспечить извлечение значений из регистров. При извлечении значений из стека будьте чрезвычайно внимательны: помните о том, что адрес возврата также записывается в стек сразу после аргументов.

- При записи в стек 10-го аргумента указатель стека `rsp` уменьшается на 8 байт.
- При записи в стек 9-го аргумента `rsp` уменьшается на 8 байт.
- При записи в стек 8-го аргумента `rsp` уменьшается на 8 байт.
- При записи в стек 7-го аргумента `rsp` уменьшается на 8 байт.
- Затем при вызове функции содержимое регистра `rip` записывается в стек, и указатель стека `rsp` уменьшается на 8 байт.
- Далее в начале функции регистр `rbp` записывается в стек – это часть пролога, а указатель стека `rsp` уменьшается на 8 байт.
- После этого стек выравнивается по 16-байтовой границе, поэтому, возможно, потребуется еще одна операция записи в стек `push`, чтобы уменьшить `rsp` на 8 байт.

Таким образом, после записи в стек аргументов функции, как минимум, еще два дополнительных регистра записываются в стек, т. е. 16 дополнительных байт. Поэтому после входа в функцию для доступа к аргументам необходимо пропустить первые 16 байт в стеке или, возможно, больше, если выполнялась операция выравнивания стека.

Аргументы с плавающей точкой передаются в xmm-регистрах в следующем порядке:

- 1-й аргумент записывается в регистр xmm0;
- 2-й аргумент записывается в регистр xmm1;
- 3-й аргумент записывается в регистр xmm2;
- 4-й аргумент записывается в регистр xmm3;
- 5-й аргумент записывается в регистр xmm4;
- 6-й аргумент записывается в регистр xmm5;
- 7-й аргумент записывается в регистр xmm6;
- 8-й аргумент записывается в регистр xmm7.

Дополнительные аргументы передаются через стек, но запись в стек выполняется не с помощью инструкции push, как можно было предположить. Эта операция будет показана позже, в главах о SIMD.

Функция возвращает результат с плавающей точкой в регистре xmm0, а целое число или адрес возвращается в регистре rax.

Выглядит слишком сложно? В листинге 15.1 показан пример, который выводит несколько аргументов с помощью функции printf.

Листинг 15.1. Программа *function5.asm*

```
; function5.asm
extern printf
section .data
    first      db    "A",0
    second     db    "B",0
    third      db    "C",0
    fourth     db    "D",0
    fifth      db    "E",0
    sixth      db    "F",0
    seventh    db    "G",0
    eighth     db    "H",0
    ninth      db    "I",0
    tenth      db    "J",0
    fmt1       db    "The string is: %s%s%s%s%s%s%s%s",10,0
    fmt2       db    "PI = %f",10,0
    pi         dq    3.14
section .bss
section .text
    global main
main:
    push    rbp
    mov     rbp, rsp
    mov     rdi, fmt1    ; Сначала используются регистры.
    mov     rsi, first
    mov     rdx, second
    mov     rcx, third
    mov     r8, fourth
```

```

mov    r9, fifth
push   tenth      ; Теперь начинается запись в стек
push   ninth      ; в обратном порядке.
push   eighth
push   seventh
push   sixth
mov    rax, 0
call   printf
and    rsp, 0xfffffffffffffff0 ; Выравнивание стека по 16-байтовой границе.
movsdb xmm0,[pi] ; Вывод числа с плавающей точкой.
mov    rax, 1      ; Выводится 1 число с плав.точкой.
mov    rdi, fmt2
call   printf
leave
ret

```

В этом примере все аргументы передаются в правильном порядке в функцию `printf`. Обратите внимание: в стек аргументы записываются в обратном порядке.

Используйте отладчик для проверки содержимого регистра `rsp` перед вызовом функции `printf`. Стек не выровнен по 16-байтовой границе. В программе не возникает критический сбой, потому что от `printf` не требуется вывод числа с плавающей точкой. Но при следующем вызове `printf` выполняется именно такой вывод. Поэтому перед использованием `printf` необходимо выравнивать стек, следовательно, используется инструкция

```
and    rsp, 0xfffffffffffffff0
```

Эта инструкция сохраняет все байты в регистре `rsp` без изменений, за исключением самого последнего байта: последние (младшие) 4 бита в `rsp` изменяются на 0, т. е. числовое значение в `rsp` уменьшается, и указатель стека `rsp` выравнивается по 16-байтовой границе. Если стек уже был выровнен изначально, то приведенная выше инструкция `and` ничего не делает. Тем не менее будьте внимательны и осторожны. Если необходимо извлекать значения из стека после выполнения этой инструкции `and`, то возникает проблема: необходимо узнать, изменила ли инструкция `and` значение `rsp`, и в итоге выравнивать указатель стека снова до этого значения перед выполнением инструкции `and`.

На рис. 15.1 показан вывод данной функции.

```

jo@UbuntuDesktop:~/Desktop/linux64/gcc/18 function5$ make
nasm -f elf64 -g -F dwarf function5.asm -l function5.lst
gcc -g -o function5 function5.o -no-pie
jo@UbuntuDesktop:~/Desktop/linux64/gcc/18 function5$ ./function5
The string is: ABCDEFGHIJ
PI = 3.140000
jo@UbuntuDesktop:~/Desktop/linux64/gcc/18 function5$ █

```

Рис. 15.1. Вывод функции *function5*

СХЕМА СТЕКА

Рассмотрим пример, в котором можно наблюдать, что происходит в стеке при записи в него аргументов функции. В листинге 15.2 показана программа, использующая функцию для формирования строки, а после возврата из этой функции созданная строка выводится.

Листинг 15.2. Программа *function6.asm*

```
; function6.asm
extern printf
section .data
    first      db      "A"
    second     db      "B"
    third      db      "C"
    fourth     db      "D"
    fifth      db      "E"
    sixth      db      "F"
    seventh    db      "G"
    eighth     db      "H"
    ninth      db      "I"
    tenth      db      "J"
    fmt        db      "The string is: %s",10,0
section .bss
    flist      resb 11      ; Длина строки + завершающий 0.
section .text
    global main
main:
    push rbp
    mov rbp, rsp
    mov rdi, flist          ; Длина.
    mov rsi, first          ; Заполнение регистров.
    mov rdx, second
    mov rcx, third
    mov r8, fourth
    mov r9, fifth
    push tenth              ; Теперь начинается запись в стек
    push ninth              ; в обратном порядке.
    push eighth
    push seventh
    push sixth
    call lfunc              ; Вызов функции.
    ; Вывод результата.
    mov rdi, fmt
    mov rsi, flist
    mov rax, 0
    call printf

leave
ret
;-----
lfunc:
    push rbp
    mov rbp, rsp
    xor rax, rax            ; Очистка rax (особенно старшие биты).
    mov al, byte[rsi]       ; Запись значения 1-го аргумента в al.
    mov [rdi], al           ; Сохранение al в памяти.
```

```

mov    al, byte[rdx]    ; Запись значения 2-го аргумента в al.
mov    [rdi+1], al      ; Сохранение al в памяти.
mov    al, byte[rcx]    ; И т. д. для всех прочих аргументов.
mov    [rdi+2], al
mov    al, byte[r8]
mov    [rdi+3], al
mov    al, byte[r9]
mov    [rdi+4], al
; Теперь извлечение аргументов из стека.
push   rbx              ; Сохраняемый вызываемой функцией.
xor     rbx,rbx
mov     rax, qword [rbp+16] ; Первое значение: начальный указатель стека
                                ; + rip + rbp
mov     bl, byte[rax]    ; Извлечение символа.
mov     [rdi+5], bl      ; Сохранение символа в памяти.
mov     rax, qword [rbp+24] ; Продолжение обработки следующего значения.
mov     bl, byte[rax]
mov     [rdi+6], bl
mov     rax, qword [rbp+32]
mov     bl, byte[rax]
mov     [rdi+7], bl
mov     rax, qword [rbp+40]
mov     bl, byte[rax]
mov     [rdi+8], bl
mov     rax, qword [rbp+48]
mov     bl, byte[rax]
mov     [rdi+9], bl
mov     bl, 0
mov     [rdi+10], bl
pop     rbx              ; Сохраняемый вызываемой функцией.
mov     rsp,rbp
pop     rbp
ret

```

В рассматриваемом здесь примере вместо вывода с помощью функции `printf` сразу после передачи всех аргументов, как это сделано в предыдущем разделе, вызывается функция `lfunc`. Эта функция принимает все аргументы и формирует строку в памяти (`flist`). Эта строка будет выведена после возвращения в основную функцию `main`.

Рассмотрим подробнее функцию `lfunc`. Она принимает только младший байт регистров аргументов, в которых записаны символы, используя инструкцию, показанную ниже:

```
mov    al, byte[rsi]
```

Эти символы поочередно сохраняются в памяти, начиная с адреса, хранящегося в регистре `rdi`, т. е. с адреса `flist`, с помощью инструкции `mov [rdi], al`. Использование ключевого слова `byte` не обязательно, но оно повышает удобство чтения исходного кода.

Для нас наиболее интересно начало извлечения значений из стека. В начале функции `lfunc` значение `rsp`, т. е. адрес стека, сохраняется в регистре `rbp`. Но между этой инструкцией и конечной инструкцией записи значений в стек в `main` содержимое регистра `rsp` было изменено дважды. Во-первых, при вы-

зове `lfunc` адрес возврата был помещен в стек. Затем в стеке был сохранен регистр `rbp` как часть пролога. В сумме указатель стека `rsp` уменьшился на 16 байт. Для доступа к значениям аргументов в стеке необходимо скорректировать их адреса на 16 байт. Именно поэтому для доступа, например, к переменной `sixth` используется инструкция

```
mov rax, qword [rbp+16]
```

Каждая следующая переменная располагается на 8 байт выше предыдущей. Регистр `rbx` используется как временное хранилище при формировании строки в памяти `flist`. Перед применением `rbx` его содержимое сохраняется в стеке. Никогда не известно, применяется ли регистр `rbx` в основной программе `main` для других целей, поэтому его содержимое сохраняется и восстанавливается перед выходом из функции.

На рис. 15.2 показан вывод этой программы.

```
jo@UbuntuDesktop:~/Desktop/linux64/gcc/18 function6$ make
nasm -f elf64 -g -F dwarf function6.asm -l function6.lst
gcc -g -o function6 function6.o -no-pie
jo@UbuntuDesktop:~/Desktop/linux64/gcc/18 function6$ ./function6
The string is: ABCDEFGHIJ
jo@UbuntuDesktop:~/Desktop/linux64/gcc/18 function6$
```

Рис. 15.2. Вывод программы *function6*

СОХРАНЕНИЕ РЕГИСТРОВ

Теперь необходимо объяснить смысл инструкций

```
push    rbx                ; Сохраняемый вызываемой функцией.
```

и

```
pop     rbx                ; Сохраняемый вызываемой функцией.
```

Должна быть очевидной необходимость наблюдения за тем, что происходит с регистрами во время вызова функции. Некоторые регистры будут изменяться во время выполнения функции, другие останутся неизменными. Необходимо предпринять меры предосторожности, чтобы избежать непредсказуемых результатов из-за изменений регистров функциями, используемыми в основной (вызывающей) программе `main`.

В табл. 15.1 дан краткий обзор определений назначения регистров из соглашений о вызовах функций.

Таблица 15.1. Соглашения о вызовах функций

Регистр	Использование	Кто сохраняет
rax	Возвращаемое значение	Вызывающая функция
rbx	Сохраняемый вызываемой функцией	Вызываемая функция
rcx	4-й аргумент	Вызывающая функция
rdx	3-й аргумент	Вызывающая функция
rsi	2-й аргумент	Вызывающая функция
rdi	1-й аргумент	Вызывающая функция
rbp	Сохраняемый вызываемой функцией	Вызываемая функция
rsp	Указатель стека	Вызываемая функция
r8	5-й аргумент	Вызывающая функция
r9	6-й аргумент	Вызывающая функция
r10	Временный	Вызывающая функция
r11	Временный	Вызывающая функция
r12	Сохраняемый вызываемой функцией	Вызываемая функция
r13	Сохраняемый вызываемой функцией	Вызываемая функция
r14	Сохраняемый вызываемой функцией	Вызываемая функция
r15	Сохраняемый вызываемой функцией	Вызываемая функция
xmm0	Первый аргумент и возвращаемое значение	Вызывающая функция
xmm1	Второй аргумент и возвращаемое значение	Вызывающая функция
xmm2-7	Аргументы	Вызывающая функция
xmm8-15	Временные	Вызывающая функция

Вызываемая функция в документации обозначена как *callee*. Если функция использует регистр, сохраняемый вызываемой функцией (*callee-saved register*), то она обязательно должна поместить содержимое этого регистра в стек перед его использованием и перед завершением работы восстановить содержимое в правильном порядке. Вызывающая функция полагает, что регистр, сохраняемый вызываемой функцией, остается неизменным после вызова функции. Регистры аргументов могут быть изменены во время выполнения функции, поэтому ответственность за их сохранение и восстановление из стека остается на вызывающей функции, если в этом есть необходимость. Временные регистры также могут изменяться в вызываемой функции, поэтому при необходимости вызывающая функция должна их сохранять и восстанавливать. Вполне очевидно, что регистр *rax*, содержащий возвращаемое значение, должен сохраняться и восстанавливаться вызывающей функцией.

Проблемы могут возникать при изменении существующей функции, когда в ней начинается использование регистра, сохраняемого вызывающей функцией. Если не добавить инструкции сохранения и восстановления этого регистра в вызывающей функции, то вы получите непредсказуемые результаты.

Регистры, сохраняемые вызываемой функцией, также называются неразрушаемыми, или долговременными (*nonvolatile*). Регистры, которые должна сохранять вызывающая функция, еще называются разрушаемыми, или кратковременными (*volatile*).

Все `xmm`-регистры могут быть изменены вызываемой функцией, поэтому при необходимости вызывающая функция отвечает за их сохранение и восстановление.

Разумеется, если вы абсолютно уверены в том, что не намерены использовать изменяемые регистры, то можете пропустить инструкции их сохранения. Но если в будущем код изменится, то, возможно, возникнут проблемы, если вы начнете использовать эти регистры без сохранения их содержимого. Можете верить или не верить, но через пару недель или месяцев ассемблерный код очень трудно читать, даже если весь исходный код был написан лично вами.

И последнее замечание: `syscall` – это тоже функция, которая изменяет регистры, так что внимательно следите за тем, что делает `syscall`.

РЕЗЮМЕ

В этой главе вы узнали, что такое:

- соглашения о вызовах функций;
- выравнивание стека;
- регистры, сохраняемые вызываемой и вызывающей функциями.

Глава 16

Операции с битами

Мы уже выполняли некоторые операции с битами в главе 9 при использовании целочисленной арифметики: инструкции арифметического сдвига `sar` и `sal` – это битовые операции, сдвигающие биты вправо и влево. Кроме того, инструкция `and`, применяемая для выравнивания стека, описанного в предыдущей главе, – это также битовая операция.

Основные положения

В приведенном ниже примере программы (листинг 16.1) создается специализированная (пользовательская) функция языка С с именем `printb` для вывода строки битов. Для удобства она разделяет строку из 64 бит на 8 байт, по 8 бит в каждом байте. В качестве дополнительного упражнения: после завершения чтения этой главы рассмотрите код на языке С – вам вполне по силам написать программу на ассемблере для формирования строки из битов.

В листингах 16.1, 16.2 и 16.3 показан пример кода с использованием битовых операций на ассемблере, программа на языке С `printb` и *makefile* соответственно.

Листинг 16.1. Программа *bits1.asm*

```
; bits1.asm
extern printb
extern printf
section .data
    msgn1 db "Number 1",10,0
    msgn2 db "Number 2",10,0
    msg1 db "XOR",10,0
    msg2 db "OR",10,0
    msg3 db "AND",10,0
    msg4 db "NOT number 1",10,0
    msg5 db "SHL 2 lower byte of number 1",10,0
    msg6 db "SHR 2 lower byte of number 1",10,0
    msg7 db "SAL 2 lower byte of number 1",10,0
    msg8 db "SAR 2 lower byte of number 1",10,0
    msg9 db "ROL 2 lower byte of number 1",10,0
    msg10 db "ROL 2 lower byte of number 2",10,0
    msg11 db "ROR 2 lower byte of number 1",10,0
    msg12 db "ROR 2 lower byte of number 2",10,0
    number1 dq -72
    number2 dq 1064
```

```
section .bss
section .text
    global main
main:
push    rbp
mov     rbp, rsp
; Вывод number1.
    mov     rsi, msgn1
    call    printmsg
    mov     rdi, [number1]
    call    printb
; Вывод number2.
    mov     rsi, msgn2
    call    printmsg
    mov     rdi, [number2]
    call    printb
; Вывод XOR (исключающего OR)-----
    mov     rsi, msg1
    call    printmsg
; xor и вывод.
    mov     rax, [number1]
    xor     rax, [number2]
    mov     rdi, rax
    call    printb
; Вывод OR -----
    mov     rsi, msg2
    call    printmsg
; or и вывод.
    mov     rax, [number1]
    or      rax, [number2]
    mov     rdi, rax
    call    printb
; Вывод AND -----
    mov     rsi, msg3
    call    printmsg
; and и вывод.
    mov     rax, [number1]
    and     rax, [number2]
    mov     rdi, rax
    call    printb
; Вывод NOT -----
    mov     rsi, msg4
    call    printmsg
; not и вывод.
    mov     rax, [number1]
    not     rax
    mov     rdi, rax
    call    printb
; Вывод SHL (сдвиг влево) -----
    mov     rsi, msg5
    call    printmsg
; shl и вывод.
    mov     rax, [number1]
    shl     al, 2
    mov     rdi, rax
    call    printb
```

```

; Вывод SHR (сдвиг вправо)-----
    mov     rsi, msg6
    call    printmsg
; shr и вывод.
    mov     rax,[number1]
    shr     al,2
    mov     rdi, rax
    call    printb
; Вывод SAL (арифметический сдвиг влево)-----
    mov     rsi, msg7
    call    printmsg
; sal и вывод.
    mov     rax,[number1]
    sal     al,2
    mov     rdi, rax
    call    printb
; Вывод SAR (арифметический сдвиг вправо)-----
    mov     rsi, msg8
    call    printmsg
; sar и вывод.
    mov     rax,[number1]
    sar     al,2
    mov     rdi, rax
    call    printb
; Вывод ROL (вращение влево)-----
    mov     rsi, msg9
    call    printmsg
; rol и вывод.
    mov     rax,[number1]
    rol     al,2
    mov     rdi, rax
    call    printb
    mov     rsi, msg10
    call    printmsg
    mov     rax,[number2]
    rol     al,2
    mov     rdi, rax
    call    printb
; Вывод ROR (вращение вправо)-----
    mov     rsi, msg11
    call    printmsg
; ror и вывод.
    mov     rax,[number1]
    ror     al,2
    mov     rdi, rax
    call    printb
    mov     rsi, msg12
    call    printmsg
    mov     rax,[number2]
    ror     al,2
    mov     rdi, rax
    call    printb
leave
ret
;-----
printmsg:    ; Вывод заголовка для каждой битовой операции.

```

```
section .data
    .fmtstr          db      "%s",0
section .text
    mov    rdi,.fmtstr
    mov    rax,0
    call   printf
    ret
```

Листинг 16.2. Функция *printb.c*

```
// printb.c

#include <stdio.h>

void printb(long long n){
    long long s,c;
    for (c = 63; c >= 0; c--)
    {
        s = n >> c;
        // Пробел после каждого 8-го бита.
        if ((c+1) % 8 == 0) printf(" ");
        if (s & 1)
            printf("1");
        else
            printf("0");
    }
    printf("\n");
}
```

Листинг 16.3. makefile для программы *bits1* и функции *printb*

```
# makefile для bits1 и printb
bits1: bits1.o printb.o
    gcc -g -o bits1 bits1.o printb.o -no-pie
bits1.o: bits1.asm
    nasm -f elf64 -g -F dwarf bits1.asm -l bits1.lst
printb: printb.c
    gcc -c printb.c
```

После сборки запустите эту программу и внимательно изучите ее вывод. Если вы используете SASM, то не забудьте сначала скомпилировать файл *printb.c* и добавить полученный объектный файл в строку параметров **Linking Options** (Параметры связывания) в соответствии с указаниями при обсуждении внешних функций в главе 14.

Это достаточно длинная программа. К счастью, ее исходный код не очень сложен. Программа демонстрирует работу различных инструкций, выполняющих операции с битами. Используйте вывод, показанный на рис. 16.1, для того чтобы лучше ориентироваться в исходном коде.

```

jo@UbuntuDesktop:~/Desktop/linux64/gcc/20_bits1$ make
nasm -f elf64 -g -F dwarf bits1.asm -l bits1.lst
cc -c -o printb.o printb.c
gcc -g -o bits1 bits1.o printb.o
jo@UbuntuDesktop:~/Desktop/linux64/gcc/20_bits1$ ./bits1
Number 1
 11111111 11111111 11111111 11111111 11111111 11111111 11111111 10111000
Number 2
 00000000 00000000 00000000 00000000 00000000 00000000 00000100 00101000
XOR
 11111111 11111111 11111111 11111111 11111111 11111111 11111011 10010000
OR
 11111111 11111111 11111111 11111111 11111111 11111111 11111111 10111000
AND
 00000000 00000000 00000000 00000000 00000000 00000000 00000100 00101000
NOT number 1
 00000000 00000000 00000000 00000000 00000000 00000000 00000000 01000111
SHL 2 lower byte of number 1
 11111111 11111111 11111111 11111111 11111111 11111111 11111111 11100000
SHR 2 lower byte of number 1
 11111111 11111111 11111111 11111111 11111111 11111111 11111111 00101110
SAL 2 lower byte of number 1
 11111111 11111111 11111111 11111111 11111111 11111111 11111111 11100000
SAR 2 lower byte of number 1
 11111111 11111111 11111111 11111111 11111111 11111111 11111111 11101110
ROL 2 lower byte of number 1
 11111111 11111111 11111111 11111111 11111111 11111111 11111111 11100010
ROL 2 lower byte of number 2
 00000000 00000000 00000000 00000000 00000000 00000000 00000100 10100000
ROR 2 lower byte of number 1
 11111111 11111111 11111111 11111111 11111111 11111111 11111111 00101110
ROR 2 lower byte of number 2
 00000000 00000000 00000000 00000000 00000000 00000000 00000100 00001010
jo@UbuntuDesktop:~/Desktop/linux64/gcc/20_bits1$

```

Рис. 16.1. Вывод программы *bits1.asm*

Первое примечание относится к двоичному представлению значения `number1` (-72) – здесь 1 в самом старшем (крайнем справа) бите обозначает отрицательное число.

Инструкции `xor`, `or`, `and` и `not` весьма просты – они выполняются так, как описано в главе 5. Поэкспериментируйте с различными значениями, чтобы лучше понять, как они работают.

Для инструкций `shl`, `shr`, `sal` и `sar` используется самый младший байт регистра `rax`, чтобы наглядно показать, что происходит. При использовании `shl` биты сдвигаются влево, а нули добавляются справа в регистре `al`, т. е. биты перемещаются влево, а те биты, которые оказываются слева от 8-го бита, просто отбрасываются. При использовании `shr` биты сдвигаются вправо, а нули добавляются слева в регистре `al`. Все биты перемещаются вправо, а те биты, которые оказываются справа от самого младшего (нулевого) бита, отбрасываются. При пошаговом проходе по программе внимательно следите за регистром флагов, особенно за флагами знака и переполнения.

Инструкция арифметического сдвига влево `sal` делает в точности то же, что инструкция `shl`, – умножает значение. Инструкция арифметического сдвига вправо `sar` отличается от инструкции `shr`. Здесь происходит то, что называют распространением (расширением) знака (*sign extension*). Если самый левый бит

в регистре `al` равен 1, то регистр `al` содержит отрицательное значение. Чтобы арифметические операции выполнялись корректно при сдвиге вправо, слева вместо нулей добавляются единицы в случае сдвига отрицательного значения. Это называется распространением (расширением) знака.

Инструкция вращения влево `rol` удаляет левый крайний бит, выполняет сдвиг влево, затем добавляет удаленный бит (биты) справа. Инструкция вращения вправо `ror` делает то же самое, но в обратном направлении.

АРИФМЕТИКА

Попробуем более подробно разобраться в сдвиговой арифметике. Почему существуют два типа инструкций сдвига влево и два типа инструкций сдвига вправо? При выполнении арифметических операций с отрицательными значениями инструкции сдвига могут давать неправильные результаты, потому что необходимо учитывать распространение (расширение) знака. Именно поэтому существуют арифметические инструкции сдвига и логические инструкции сдвига.

Рассмотрим пример в листинге 16.4.

Листинг 16.4. Программа *bits2.asm*

```
; bits2.asm
extern printf
section .data
    msgn1 db "Number 1 is = %d",0
    msgn2 db "Number 2 is = %d",0
    msg1 db "SHL 2 = OK multiply by 4",0
    msg2 db "SHR 2 = WRONG divide by 4",0
    msg3 db "SAL 2 = correctly multiply by 4",0
    msg4 db "SAR 2 = correctly divide by 4",0
    msg5 db "SHR 2 = OK divide by 4",0
    number1 dq 8
    number2 dq -8
    result dq 0
section .bss
section .text
    global main
main:
    push rbp
    mov rbp,rbp
;SHL-----
; Положительное число.
    mov rsi, msg1
    call printfmsg ; Вывод заголовка.
    mov rsi, [number1]
    call printnbr ; Вывод числа number1.
    mov rax,[number1]
    shl rax,2 ; Умножение на 4 (логический сдвиг).
    mov rsi, rax
    call printres
; Отрицательное число.
    mov rsi, msg1
    call printfmsg ; Вывод заголовка.
```

```

    mov    rsi, [number2]
    call   printnbr      ; Вывод числа number2.
    mov    rax, [number2]
    shl    rax, 2        ; Умножение на 4 (логический сдвиг).
    mov    rsi, rax
    call   printres

;SAL-----
; Положительное число.
    mov    rsi, msg3
    call   printmsg      ; Вывод заголовка.
    mov    rsi, [number1]
    call   printnbr      ; Вывод числа number1.
    mov    rax, [number1]
    sal    rax, 2        ; Умножение на 4 (арифметический сдвиг)
    mov    rsi, rax
    call   printres

; Отрицательное число.
    mov    rsi, msg3
    call   printmsg      ; Вывод заголовка.
    mov    rsi, [number2]
    call   printnbr      ; Вывод числа number2.
    mov    rax, [number2]
    sal    rax, 2        ; Умножение на 4 (арифметический сдвиг)
    mov    rsi, rax
    call   printres

;SHR-----
; Положительное число.
    mov    rsi, msg5
    call   printmsg      ; Вывод заголовка.
    mov    rsi, [number1]
    call   printnbr      ; Вывод числа number1.
    mov    rax, [number1]
    shr    rax, 2        ; Деление на 4 (логический сдвиг).
    mov    rsi, rax
    call   printres

; Отрицательное число.
    mov    rsi, msg2
    call   printmsg      ; Вывод заголовка.
    mov    rsi, [number2]
    call   printnbr      ; Вывод числа number2.
    mov    rax, [number2]
    shr    rax, 2        ; Деление на 4 (логический сдвиг).
    mov    [result], rax
    mov    rsi, rax
    call   printres

;SAR-----
; Положительное число.
    mov    rsi, msg4
    call   printmsg      ; Вывод заголовка.
    mov    rsi, [number1]
    call   printnbr      ; Вывод числа number1.
    mov    rax, [number1]
    sar    rax, 2        ; Деление на 4 (арифметический сдвиг).
    mov    rsi, rax
    call   printres

; Отрицательное число.

```

```

    mov    rsi, msg4
    call   printmsg      ; Вывод заголовка.
    mov    rsi, [number2]
    call   printnbr      ; Вывод числа number2.
    mov    rax, [number2]
    sar    rax, 2         ; Деление на 4 (арифметический сдвиг).
    mov    rsi, rax
    call   printres

leave
ret
;-----
printmsg:      ; Вывод общего заголовка.
    section .data
        .fmtstr db 10, "%s", 10, 0 ; Формат для вывода строки.
    section .text
        mov    rdi, .fmtstr
        mov    rax, 0
        call   printf
    ret
;-----
printnbr:      ; Вывод исходного числа.
    section .data
        .fmtstr db "The original number is %lld", 10, 0
    section .text
        mov    rdi, .fmtstr
        mov    rax, 0
        call   printf
    ret
;-----
printres:      ; Вывод результата.
    section .data
        .fmtstr db "The resulting number is %lld", 10, 0
    section .text
        mov    rdi, .fmtstr
        mov    rax, 0
        call   printf
    ret

```

Используйте вывод этой программы, показанный на рис. 16.2, для анализа исходного кода.

Обратите внимание: инструкции `shl` и `sal` дают одинаковые результаты даже с отрицательными числами. Но будьте внимательны и осторожны: если инструкция `shl` поместит 1 вместо 0 в левый крайний бит, то результат станет отрицательным, т. е. неверным.

Инструкции `shr` и `sar` дают одинаковые результаты, только если числа положительные. Арифметический результат при использовании `shr` для отрицательных чисел просто некорректен, потому что инструкция `shr` не выполняет распространение знака.

Вывод: при выполнении арифметических операций необходимо использовать инструкции `sal` и `sar`.

Зачем нужны инструкции сдвига, если существуют арифметические инструкции, явно предназначенные для выполнения умножения и деления? Оказывается, сдвиг выполняется намного быстрее, чем инструкции умножения и деления. В общем случае битовые инструкции выполняются чрезвычайно быстро, например `hoge`, `gah` намного быстрее, чем `mov gah, 0`.

```
jo@UbuntuDesktop:~/Desktop/linux64/gcc/21 bits2$ make
nasm -f elf64 -g -F dwarf bits2.asm -l bits2.lst
gcc -g -o bits2 bits2.o
jo@UbuntuDesktop:~/Desktop/linux64/gcc/21 bits2$ ./bits2

SHL 2 = OK multiply by 4
The original number is 8
The resulting number is 32

SHL 2 = OK multiply by 4
The original number is -8
The resulting number is -32

SAL 2 = correctly multiply by 4
The original number is 8
The resulting number is 32

SAL 2 = correctly multiply by 4
The original number is -8
The resulting number is -32

SHR 2 = OK divide by 4
The original number is 8
The resulting number is 2

SHR 2 = wrong divide by 4
The original number is -8
The resulting number is 4611686018427387902

SAR 2 = correctly divide by 4
The original number is 8
The resulting number is 2

SAR 2 = correctly divide by 4
The original number is -8
The resulting number is -2
jo@UbuntuDesktop:~/Desktop/linux64/gcc/21 bits2$
```

Рис. 16.2. Вывод программы *bits2.asm*

РЕЗЮМЕ

В этой главе вы узнали:

- об использовании ассемблерных инструкций для операций с битами;
- о различии между инструкциями логического и арифметического сдвига.

Глава 17

Работа с битами

Вам уже известно о возможности устанавливать или очищать (сбрасывать) биты с помощью битовых операций, таких как `and`, `xor`, `or` и `not`. Но существуют и другие способы изменения отдельных битов: `bts` для установки битов в 1, `btr` для сброса битов в 0 и `bt` для проверки, установлен ли бит в 1.

ДРУГИЕ СПОСОБЫ ИЗМЕНЕНИЯ БИТОВ

В листинге 17.1 показан пример исходного кода.

Листинг 17.1. Программа *bits3.asm*

```
; bits3.asm
extern printf
extern printf
section .data
    msg1 db "No bits are set:",10,0
    msg2 db 10,"Set bit #4, that is the 5th bit:",10,0
    msg3 db 10,"Set bit #7, that is the 8th bit:",10,0
    msg4 db 10,"Set bit #8, that is the 9th bit:",10,0
    msg5 db 10,"Set bit #61, that is the 62nd bit:",10,0
    msg6 db 10,"Clear bit #8, that is the 9th bit:",10,0
    msg7 db 10,"Test bit #61, and display rdi",10,0
    bitflags dq 0
section .bss
section .text
    global main
main:
    push rbp
    mov rbp, rsp
    ; Вывод заголовка.
    mov rdi, msg1
    xor rax, rax
    call printf
    ; Вывод переменной bitflags.
    mov rdi, [bitflags]
    call printf
; Установить в 1 бит 4 (=5-й бит).
; Вывод заголовка.
    mov rdi, msg2
    xor rax, rax
    call printf
```

```

    bts    qword [bitflags],4    ; Установить бит 4.
    ; Вывод переменной bitflags.
    mov    rdi,[bitflags]
    call   printb
; Установить в 1 бит 7 (=8-й бит).
    ; Вывод заголовка.
    mov    rdi,msg3
    xor     rax,rax
    call   printf
    bts    qword [bitflags],7    ; Установить бит 7.
    ; Вывод переменной bitflags.
    mov    rdi,[bitflags]
    call   printb
; Установить в 1 бит 8 (=9-й бит).
    ; Вывод заголовка.
    mov    rdi,msg4
    xor     rax,rax
    call   printf
    bts    qword [bitflags],8    ; Установить бит 8.
    ; Вывод переменной bitflags.
    mov    rdi,[bitflags]
    call   printb
; Установить в 1 бит 61 (=62-й бит).
    ; Вывод заголовка.
    mov    rdi,msg5
    xor     rax,rax
    call   printf
    bts    qword [bitflags],61   ; Установить бит 61.
    ; Вывод переменной bitflags.
    mov    rdi,[bitflags]
    call   printb
; Очистить бит 8 (=9-й бит).
    ; Вывод заголовка.
    mov    rdi,msg6
    xor     rax,rax
    call   printf
    btr    qword [bitflags],8    ; Сброс бита 8.
    ; Вывод переменной bitflags.
    mov    rdi,[bitflags]
    call   printb
; Проверка бита 61 (будет установлен в 1 флаг переноса CF, если этот бит равен 1).
    ; Вывод заголовка.
    mov    rdi,msg7
    xor     rax,rax
    call   printf
    xor     rdi,rdi
    mov     rax,61               ; Нужно проверить бит 61.
    xor     rdi,rdi              ; Чтобы все биты были равны 0.
    bt      [bitflags],rax       ; Проверка бита.
    setc    dil                  ; Установить dil (=младший rdi) в 1, если CF установлен.
    call    printb               ; Вывод регистра rdi.
leave
ret

```

Здесь снова используется программа *printb.c*, поэтому необходимо изменить соответствующим образом *makefile* или параметры настройки сборки в *SASM*.

В рассматриваемом примере переменная `bitflags` является объектом исследования, в ней выполняются различные операции с отдельными битами.

ПЕРЕМЕННАЯ BITFLAGS

Напомню, что отсчет битов (индексирование) начинается с 0. Это значит, что в байте, состоящем из 8 бит, первый бит находится в позиции 0, а последний бит – в позиции 7. Установка битов в 1 с помощью инструкции `bts` и сброс битов в 0 с помощью инструкции `btg` выполняются просто: нужно только указать индекс изменяемого бита во втором операнде.

Проверка бита выполняется немного сложнее. Необходимо поместить индекс проверяемого бита в регистр `rax` и использовать инструкцию `bt`. Если бит равен 1, то флаг переноса `CF` будет установлен в 1, иначе `CF` будет равен 0. На основе значения флага `CF` можно перенаправить программу для выполнения определенных инструкций или продолжить выполнение без изменения порядка. В этом случае используется специальная инструкция `setc` для установки значения по условию. В рассматриваемом здесь примере инструкция `setc` устанавливает `dil` в 1, если флаг переноса равен 1. `dil` – это младшая (нижняя) часть регистра `rdi`. Необходимо обязательно сбросить регистр `rdi` в 0 перед использованием инструкции `setc` для условной установки `dil`. Вполне может случиться, что старшие биты регистра `rdi` установлены во время выполнения предыдущей инструкции.

Инструкция `setc` представляет собой пример из группы инструкций `setCC`. Инструкции `setCC` устанавливают в 1 байт, указанный в операнде, если выполнено условие `CC`, где `CC` – это флаг, например `CF` (инструкция `setc`), `ZF` (инструкция `setz`), `SF` (инструкция `sets`) и т. д. Более подробно об этом см. в руководствах Intel.

На рис. 17.1 показан вывод этой программы.

```
jo@UbuntuDesktop:~/Desktop/linux64/gcc/22 bits3$ make
nasm -f elf64 -g -F dwarf bits3.asm -l bits3.lst
cc -c -o printb.o printb.c
gcc -g -o bits3 bits3.o printb.o
jo@UbuntuDesktop:~/Desktop/linux64/gcc/22 bits3$ ./bits3
No bits are set:
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

Set bit #4, that is the 5th bit:
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00010000

Set bit #7, that is the 8th bit:
00000000 00000000 00000000 00000000 00000000 00000000 00000000 10010000

Set bit #8, that is the 9th bit:
00000000 00000000 00000000 00000000 00000000 00000000 00000001 10010000

Set bit #61, that is the 62nd bit:
00100000 00000000 00000000 00000000 00000000 00000000 00000001 10010000

Clear bit #8, that is the 9th bit:
00100000 00000000 00000000 00000000 00000000 00000000 00000000 10010000

Test bit #61, and display dl
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000001
jo@UbuntuDesktop:~/Desktop/linux64/gcc/22 bits3$
```

Рис. 17.1. Вывод программы `bits3.asm`

РЕЗЮМЕ

В этой главе вы узнали:

- об установке, сбросе и проверке битов с помощью инструкций `bts`, `btr` и `bt`;
- о группе инструкций `setcc`.

Глава 18

Макрокоманды

При многократном использовании в программе одного и того же набора инструкций можно создать функцию и вызывать ее каждый раз, когда необходимо выполнить эти инструкции. Но для функций характерно снижение производительности: при каждом вызове функции поток выполнения переходит к вызываемой функции в некоторой области памяти, а после ее завершения возвращается обратно в вызывающую программу. Вызов функции и возврат из нее требуют определенного времени.

Чтобы избежать подобных проблем с производительностью, можно воспользоваться макрокомандами (или просто макросами – *macros*). Как и функция, макрос представляет собой последовательность инструкций. Макросу присваивается имя, и когда нужно выполнить этот макрос в коде, вы просто указываете его имя, возможно, сопровождаемое аргументами.

Важное отличие от функции: во время ассемблирования везде, где в исходном коде встречается «вызов» макроса, NASM заменяет имя этого макроса на набор инструкций, записанных в его определении. Во время выполнения не происходит никаких переходов и возвратов, потому что NASM уже вставил машинный код во всех необходимых местах.

Макросы не являются функциональным свойством языка ассемблера Intel, но эта функциональность предоставляется NASM (или другой версией программы ассемблирования). Макросы создаются с помощью директив препроцессора, и NASM использует макропроцессор для преобразования макросов в машинный язык и вставляет инструкции машинного языка в соответствующие места в коде программы.

Макросы позволяют улучшить скорость выполнения кода, но они увеличивают объем кода, потому что во время ассемблирования инструкции из определения макроса вставляются в каждую локацию, где используется макрос.

Более подробную информацию о макросах NASM можно получить из руководства NASM, глава 4 «The NASM Preprocessor» (для версии NASM 2.14.02).

СОЗДАНИЕ МАКРОСА

В листинге 18.1 показано несколько примеров определения и использования макросов.

Листинг 18.1. Программа *macro.asm*

```

; macro.asm
extern printf

%define    double_it(r)    sal r, 1    ; Однострочный макрос.

%macro     printf 2        ; Макрос из нескольких строк с 2 аргументами.
    section .data
        %%arg1    db    %1,0        ; Первый аргумент.
        %%fmtint   db    "%s %ld",10,0    ; Строка формата.
    section .text          ; Аргументы для внешней функции printf.
        mov     rdi,%%fmtint
        mov     rsi,%%arg1
        mov     rdx,[%2]    ; Второй аргумент.
        mov     rax,0        ; Числа с плав. точкой не используются.
        call    printf
%endmacro

section .data
    number    dq    15
section .bss
section .text
    global main
main:
push     rbp
mov     rbp,rbp
printf    "The number is", number
mov     rax,[number]
double_it(rax)
mov     [number],rax
printf    "The number times 2 is", number
leave
ret

```

Существует два типа макросов: однострочные макросы и макросы из нескольких строк (многострочные). Однострочный макрос начинается с ключевого слова `%define`. Многострочный макрос определяется между ключевыми словами `%macro` и `%endmacro`. Ключевые слова `%define`, `%macro` и `%endmacro` называются директивами препроцессора ассемблера (*assembler preprocessor directives*).

Однострочный макрос чрезвычайно прост: во время ассемблирования инструкция `double_it(rax)` заменяется на машинный код, соответствующий инструкции `sal r, 1`, где `r` – значение в регистре `rax`.

Многострочный макрос немного более сложен: здесь макрос `printf` вызывается с двумя аргументами. В определении макроса можно видеть, что за именем `printf` следует число 2, определяющее количество аргументов. Чтобы аргументы можно было использовать внутри макроса, они обозначаются символами `%1` для первого аргумента, `%2` для второго и т. д. Обратите внимание: аргумент `%1` позволяет использовать строку, но аргумент `[%2]` (в квадратных скобках) позволяет обратиться к числовому значению по аналогии с требованиями к коду без использования макросов.

Внутри макроса можно использовать переменные, при этом рекомендуется

давать им имена с префиксом %, например %arg1 и %fmtint. Если пропустить символы %, то NASM успешно создает переменные макро при первом вызове `printf`, но сгенерирует ошибку ассемблирования при втором вызове `printf`, сообщая, что вы пытаетесь повторно определить переменные `arg1` и `fmtint`. Символы % информируют NASM о необходимости создания новых экземпляров переменных при каждом вызове макроса. (Дополнительное упражнение: удалите символы % и попробуйте ассемблировать программу.)

Для ассемблерных макросов существует одна крупная проблема: их сложно отлаживать. Попробуйте отладить программу *macro.asm* в GDB или в отладчике на основе GDB, например в SASM, и понаблюдайте за ее поведением.

На рис. 18.1 показан вывод программы *macro.asm*.

```
jo@UbuntuDesktop:~/Desktop/linux64/gcc/23 macro$ make
nasm -f elf64 -g -F dwarf macro.asm -l macro.lst
gcc -o macro macro.o
jo@UbuntuDesktop:~/Desktop/linux64/gcc/23 macro$ ./macro
The number is 15
The number times 2 is 30
jo@UbuntuDesktop:~/Desktop/linux64/gcc/23 macro$ █
```

Рис. 18.1. Вывод программы *macro.asm*

ИСПОЛЬЗОВАНИЕ OBJDUMP

Проверим тот факт, что ассемблированный код макроса действительно вставлен в соответствующих местах в исполняемом файле, т. е. там, где использовался макрос. Для этого воспользуемся утилитой командной строки с именем `objdump`. Если вы установили инструментальные средства разработки в соответствии с рекомендациями в начале этой книги, то утилита `objdump` уже установлена и готова к работе. В командной строке введите следующую строку:

```
objdump -M intel -d macro
```

Флаг `-M intel` позволит получить код с синтаксисом ассемблера Intel, а флаг `-d macro` обеспечивает дизассемблирование выполняемого кода макроса. Выполните прокрутку по полученному коду до раздела `<main>`.

На рис. 18.2 можно видеть, что код макроса `printf` вставлен в код `main`, начиная с адреса памяти `4004f4` до адреса `400515`, а также с адреса `40052d` до адреса `40054e`. Код макроса `double_it` размещен по адресу `400522`. Программа ассемблирования принимает на себя ответственность при замене инструкции `sal` на `shl` по соображениям улучшения производительности. Как вы уже знаете из главы 16 об инструкциях сдвига, в большинстве случаев такая замена не приводит к каким-либо проблемам. Для эксперимента замените инструкцию `sal` на инструкцию `sar`. Вы увидите, что программа ассемблирования не заменяет `sar` на `shr`, чтобы избежать проблем.

Утилита командной строки `objdump` полезна при исследовании кода, даже если код написан не вами. Вы можете получить огромный объем информации о выполняемом файле, используя `objdump`, но мы не будем углубляться в подробности

в этой книге. Если вы хотите узнать больше, то введите команду `man objdump` в командной строке или найдите требуемую информацию в интернете.

```

00000000004004f0 <main>:
4004f0: 55                push    rbp
4004f1: 48 89 e5          mov     rbp,rsp
4004f4: 48 bf 46 10 60 00 00 movabs  rdi,0x601046
4004fb: 00 00 00          movabs  rsi,0x601038
4004fe: 48 be 38 10 60 00 00 movabs  rsi,0x601038
400505: 00 00 00          mov     rdx,QWORD PTR ds:0x601030
400508: 48 8b 14 25 30 10 60 mov     rdx,QWORD PTR ds:0x601030
40050f: 00                mov     eax,0x0
400510: b8 00 00 00 00    mov     eax,0x0
400515: e8 d6 fe ff ff    call    4003f0 <printf@plt>
40051a: 48 8b 04 25 30 10 60 mov     rax,QWORD PTR ds:0x601030
400521: 00                shl     rax,1
400522: 48 d1 e0          mov     QWORD PTR ds:0x601030,rax
400525: 48 89 04 25 30 10 60 mov     rdi,0x601064
40052c: 00                movabs  rdi,0x601064
400534: 00 00 00          movabs  rsi,0x60104e
400537: 48 be 4e 10 60 00 00 movabs  rsi,0x60104e
40053e: 00 00 00          mov     rdx,QWORD PTR ds:0x601030
400541: 48 8b 14 25 30 10 60 mov     rdx,QWORD PTR ds:0x601030
400548: 00                mov     eax,0x0
400549: b8 00 00 00 00    mov     eax,0x0
40054e: e8 9d fe ff ff    call    4003f0 <printf@plt>
400553: c9                leave   %eax
400554: c3                ret
400555: 66 2e 0f 1f 84 00 00 nop     WORD PTR cs:[rax+rax*1+0x0]
40055c: 00 00 00          nop
40055f: 90                nop

```

Рис. 18.2. Результат выполнения команды `objdump -M intel -d macro`

РЕЗЮМЕ

В этой главе вы узнали:

- когда следует использовать макросы, а когда – функции;
- об однострочных макросах;
- о многострочных макросах;
- о передаче аргументов в многострочные макросы;
- о проблемах отладки в GDB при использовании ассемблерных макросов;
- об использовании утилиты `objdump`.

Глава 19

Ввод и вывод в консоли

Вам уже известно, как выполняется вывод в консоли с использованием системных вызовов или внешней функции `printf`. В этой главе снова будут применяться системные вызовы, но не только для вывода на экран, а еще и для приема ввода с клавиатуры.

ИСПОЛЬЗОВАНИЕ СРЕДСТВ ВВОДА/ВЫВОДА

Можно было бы просто позаимствовать необходимые функции из стандартной библиотеки языка C, но это испортило бы удовольствие от работы с ассемблером. В листинге 19.1 показан пример исходного кода.

Листинг 19.1. Программа *console1.asm*

```
; console1.asm
section .data
    msg1      db      "Hello, World!",10,0
    msg1len   equ     $-msg1
    msg2      db      "Your turn: ",0
    msg2len   equ     $-msg2
    msg3      db      "You answered: ",0
    msg3len   equ     $-msg3
    inputlen  equ     10    ; Длина буфера ввода.
section .bss
    input resb inputlen+1 ; Обеспечение места для завершающего 0.
section .text
    global main
main:
    push rbp
    mov  rbp,rbp
    mov  rsi, msg1      ; Вывод первой строки.
    mov  rdx, msg1len
    call prints
    mov  rsi, msg2      ; Вывод второй строки, без NL.
    mov  rdx, msg2len
    call prints
    mov  rsi, input      ; Адрес буфера ввода inputbuffer.
    mov  rdx, inputlen   ; Длина буфера ввода inputbuffer.
    call reads           ; Ожидание ввода.
    mov  rsi, msg3      ; Вывод третьей строки.
    mov  rdx, msg3len
```

```

    call prints
    mov  rsi, input      ; Вывод содержимого буфера ввода inputbuffer.
    mov  rdx, inputlen   ; Длина буфера ввода inputbuffer.
    call prints
leave
ret
;-----
prints:
push  rbp
mov   rbp, rsp
; rsi содержит адрес строки.
; rdx содержит длину строки.
    mov  rax, 1          ; 1 = запись
    mov  rdi, 1          ; 1 = stdout - стандартный вывод.
    syscall
leave
ret
;-----
reads:
push  rbp
mov   rbp, rsp
; rsi содержит адрес буфера ввода inputbuffer.
; rdi содержит длину буфера ввода inputbuffer.
    mov  rax, 0          ; 0 = чтение
    mov  rdi, 1          ; 1 = stdin - стандартный ввод.
    syscall
leave
ret

```

Это не очень сложная программа, в ней создается буфер ввода с именем `input` для хранения вводимых символов. Кроме того, определяется длина этого буфера в переменной `inputlen`. После вывода нескольких приветственных сообщений вызывается функция `reads`, которая принимает символы, вводимые с клавиатуры, и возвращает их в вызывающую программу после нажатия клавиши **Enter**. Затем вызывающая программа использует функцию `prints` для вывода ранее введенных символов. На рис. 19.1 показан вывод этой программы.

```

jo@ubuntu18:~/Desktop/Book/24 console 1$ make
nasm -f elf64 -g -F dwarf console1.asm -l console1.lst
gcc -o console1 console1.o -no-pie
jo@ubuntu18:~/Desktop/Book/24 console 1$ ./console1
Hello, World!
Your turn: Hi There!
You answered: Hi There!
jo@ubuntu18:~/Desktop/Book/24 console 1$ █

```

Рис. 19.1. Вывод программы *console1.asm*

Но и здесь возникают некоторые проблемы. Для буфера ввода зарезервировано 10 байт. Что произойдет, если будет введено более 10 символов? На рис. 19.2 показан результат в этом случае.

```

jo@ubuntu18:~/Desktop/Book/24 console 1$ ./console1
Hello, World!
Your turn: Hi there, how are you?
You answered: Hi there, jo@ubuntu18:~/Desktop/Book/24 console 1$ how are you?

Command 'how' not found, did you mean:

  command 'show' from deb mailutils-mh
  command 'show' from deb mmh
  command 'show' from deb nmh
  command 'cow' from deb fl-cow
  command 'hoz' from deb hoz
  command 'sow' from deb ruby-hoe
  command 'hot' from deb hopenpgp-tools

Try: sudo apt install <deb name>

jo@ubuntu18:~/Desktop/Book/24 console 1$ █

```

Рис. 19.2. Вывод программы *console1.asm* при вводе слишком большого количества символов

Эта программа принимает только 10 символов и не знает, что делать с остальными «лишними» символами, поэтому возвращает их в операционную систему. Операционная система пытается определить их смысл и интерпретирует эти символы как команды, но не находит соответствующие файлы или имена встроенных команд, поэтому выводит сообщения об ошибках.

Результат неутешителен, но он даже хуже, чем кажется на первый взгляд. Такой способ обработки вывода может стать причиной серьезной уязвимости (бреши) в системе обеспечения безопасности, через которую хакер может взломать программу и получить доступ к операционной системе.

ОБРАБОТКА ПЕРЕПОЛНЕНИЙ

В листинге 19.2 показана другая версия программы, в которой ведется подсчет вводимых символов, а лишние символы просто отбрасываются. Дополнительное условие: разрешен ввод только алфавитных символов в нижнем регистре, от а до z.

Листинг 19.2. Программа *console2.asm*

```

; console2.asm
section .data
    msg1 db "Hello, World!",10,0
    msg2 db "Your turn (only a-z): ",0
    msg3 db "You answered: ",0
    inputlen equ 10 ; Длина буфера ввода inputbuffer.
    NL db 0xa
section .bss
    input resb inputlen+1 ; Обеспечение места для завершающего 0.
section .text
    global main
main:
    push rbp
    mov rbp, rsp
    mov rdi, msg1 ; Вывод первой строки.

```

```

    call prints
    mov rdi, msg2 ; Вывод второй строки, без NL.
    call prints
    mov rdi, input ; Адрес буфера ввода inputbuffer.
    mov rsi, inputlen ; Длина буфера ввода inputbuffer.
    call reads ; Ожидание ввода.
    mov rdi, msg3 ; Вывод третьей строки и добавление введенной строки.
    call prints
    mov rdi, input ; Вывод содержимого буфера ввода inputbuffer.
    call prints
    mov rdi, NL ; Вывод символа перехода на новую строку NL.
    call prints
leave
ret
;-----
prints:
push rbp
mov rbp, rsp
push r12 ; Регистр, сохраняемый вызываемой функцией.
; Подсчет символов.
xor rdx, rdx ; Длина в rdx.
mov r12, rdi
.lengthloop:
    cmp byte [r12], 0
    je .lengthfound
    inc rdx
    inc r12
    jmp .lengthloop
.lengthfound: ; Вывод строки, ее длина в rdx.
    cmp rdx, 0 ; Строка отсутствует (длина 0).
    je .done
    mov rsi, rdi ; rdi содержит адрес строки.
    mov rax, 1 ; 1 = запись.
    mov rdi, 1 ; 1 = stdout - стандартный вывод.
    syscall
.done:
pop r12
leave
ret
;-----
reads:
section .data
section .bss
    .inputchar resb 1
section .text
push rbp
mov rbp, rsp
push r12 ; Регистр, сохраняемый вызываемой функцией.
push r13 ; Регистр, сохраняемый вызываемой функцией.
push r14 ; Регистр, сохраняемый вызываемой функцией.
mov r12, rdi ; Адрес буфера ввода inputbuffer.
mov r13, rsi ; Максимальная длина в r13.
xor r14, r14 ; Счетчик символов.
.readc:
mov rax, 0 ; Чтение.
mov rdi, 1 ; stdin - стандартный ввод.

```



```

    lea    rsi, [.inputchar]      ; Адрес источника ввода.
    mov    rdx, 1                ; Число считываемых символов.
    syscall
    mov    al, [.inputchar]      ; Введен символ NL?
    cmp    al, byte[NL]
    je     .done                 ; NL – конец ввода.
    cmp    al, 97                 ; Код символа меньше а?
    jl     .readc                ; Отбросить символ.
    cmp    al, 122               ; Код символа больше z?
    jg     .readc                ; Отбросить символ.
    inc    r14                   ; Увеличить счетчик символов на 1.
    cmp    r14, r13
    ja     .readc                ; Максимальное заполнение буфера, отбросить лишнее.
    mov    byte [r12], al        ; Сохранить символ в буфере.
    inc    r12                   ; Переместить указатель на следующий символ в буфере.
    jmp    .readc
.done:
    inc    r12
    mov    byte [r12], 0         ; Добавить завершающий 0 в буфер ввода inputbuffer.
    pop    r14                   ; Регистр, сохраняемый вызываемой функцией.
    pop    r13                   ; Регистр, сохраняемый вызываемой функцией.
    pop    r12                   ; Регистр, сохраняемый вызываемой функцией.
leave
ret

```

Функция `prints` изменена: сначала она подсчитывает количество выводимых символов, т. е. счет ведется до тех пор, пока не встретится байт 0. После определения длины `prints` выводит строку с помощью `syscall`.

Функция `reads` ожидает ввода одного символа и проверяет, не является ли он символом перехода на новую строку (NL). Если получен символ перехода на новую строку, то чтение символов с клавиатуры прекращается. Регистр `r14` содержит счетчик вводимых символов. Функция следит за тем, чтобы количество введенных символов не превышало длину буфера `inputlen`. Если это количество меньше длины буфера, то символ добавляется в буфер `input`. Если длина `inputlen` превышена, то символ отбрасывается, но чтение с клавиатуры продолжается. Введено дополнительное требование: ASCII-код символа должен быть не меньше 97 (a) и не больше 122 (z). Это требование обеспечивает прием только алфавитных символов в нижнем регистре. Обратите внимание на сохранение и восстановление регистров, сохраняемых вызываемой функцией (*callee saved*), – регистр `r12` используется в обеих функциях `prints` и `reads`. В рассматриваемом здесь примере отсутствие инструкций сохранения таких (*callee saved*) регистров не должно приводить к проблемам, но легко представить себе ситуацию, когда одна из функций вызывает другую, а вызываемая функция обращается к следующей, – тогда могут возникать проблемы.

На рис. 19.3 показан вывод этой программы.

```
jo@ubuntu18:~/Desktop/Book/24 console 2$ make
nasm -f elf64 -g -F dwarf console2.asm -l console2.lst
gcc -o console2 console2.o -no-pie
jo@ubuntu18:~/Desktop/Book/24 console 2$ ./console2
Hello, World!
Your turn (only a-z): 123a{bcde}fghijklmnop
You answered: abcdefghij
jo@ubuntu18:~/Desktop/Book/24 console 2$ █
```

Рис. 19.3. Вывод программы *console2.asm*

Отладка консольного ввода с использованием SASM затруднена, потому что ввод обеспечивается через системный вызов `syscall`. SASM предоставляет собственные функциональные возможности для ввода/вывода, но мы не пользовались ими, так как необходимо было продемонстрировать, как работает ассемблер и машинный язык, не скрывая подробностей. Если возникли серьезные затруднения при отладке в SASM, то рекомендуется вернуться к старому доброму отладчику GDB.

РЕЗЮМЕ

В этой главе вы узнали:

- о вводе с клавиатуры с использованием системного вызова `syscall`;
- о проверке корректности ввода с клавиатуры;
- об отладке при вводе с клавиатуры, которая может оказаться затрудненной.

Глава 20

Файловый ввод/вывод

При разработке программного обеспечения операции с файлами могут быть сложными. Различные операционные системы предоставляют разнообразные методы управления файлами, и для каждого метода существует обширный список параметров. В этой главе рассматривается файловый ввод/вывод для систем Linux. В главе 43 вы увидите, что файловый ввод/вывод в ОС Windows выполняется совершенно другими методами.

В Linux методы управления файлами сложны и включают создание и открытие файла только для чтения или для чтения/записи, для записи в новый файл или для добавления в существующий файл, а также удаление файлов... не говоря уже о параметрах защиты файлов с разделением на «пользователя-владельца» (user), «группу» (group) и «прочих» (other). При необходимости придется освежить в памяти свои навыки администратора файловой системы Linux и обратиться к руководству системного администратора Linux. В исходном коде примера этой главы определяются только флаги для текущего пользователя (user), но вы можете самостоятельно добавить определения флагов для группы (group) и прочих (other). Если вы не совсем понимаете, о чем идет речь, то прямо сейчас необходимо узнать немного больше об основах управления файлами в Linux.

ИСПОЛЬЗОВАНИЕ СИСТЕМНЫХ ВЫЗОВОВ

Файлы создаются, открываются, закрываются и т. д. через системные вызовы. В этой главе используются многие системные вызовы, поэтому сейчас требуются некоторые объяснения, чтобы упростить понимание. В начальной части исходного кода будут определяться константы, для того чтобы в дальнейшем проще было обращаться к номерам системных вызовов. В приведенном ниже примере исходного кода константы номеров системных вызовов легко узнать по префиксу NR_. Применение таких констант NR_syscall делает исходный код более удобным для чтения. Полный список символьных имен системных вызовов для вашей системы можно найти в файле

`/usr/include/asm/unistd_64.h`

Эти имена будут использоваться в приведенной ниже программе. Следует отметить, что здесь также используется заголовочный файл *unistd_32.h* для обеспечения обратной совместимости с 32-битовыми системными компонентами.

Кроме того, созданы константы для флагов создания, состояния и режимов доступа к файлам. Эти флаги определяют, должен ли файл создаваться или использоваться существующий файл для добавления в него данных, предназначен ли файл только для чтения, только для записи и т. п. Список и краткие описания этих флагов можно найти в файле

/usr/include/asm-generic/fcntl.h

Значения флагов в этом файле приведены в восьмеричном формате (например, `O_CREAT = 00000100`). Значение, начинающееся с `0x`, является шестнадцатеричным, а значение, начинающееся с `0` без `x`, – восьмеричное. Для удобства чтения к восьмеричному числу будет добавляться символ `q`.

При создании файла необходимо обязательно определить права доступа к нему. Напомню, что в Linux определяются права на чтение, запись и выполнение файла для пользователя (владельца), группы и прочих. Общий обзор прав доступа и огромное количество подробностей можно получить с помощью команды

`man 2 open`

Права доступа к файлу также определяются в восьмеричном формате и хорошо знакомы любому системному администратору Linux. Для сохранения логической согласованности мы позаимствуем символические имена, используемые в описанных выше файлах.

Пример исходного кода программы достаточно длинный, но мы будем анализировать программу постепенно, шаг за шагом – такой подход можно применить, используя условное ассемблирование (*conditional assembly*). Это дает возможность постепенного анализа программы по ее последовательным фрагментам.

ОБРАБОТКА ФАЙЛА

В рассматриваемом здесь примере программы выполняются следующие операции:

- 1) создание файла, затем запись данных в этот файл;
- 2) перезапись (замещение) части содержимого того же файла;
- 3) добавление данных в файл;
- 4) запись данных в определенную позицию файла;
- 5) чтение данных из файла;
- 6) чтение данных из определенной позиции файла;
- 7) удаление файла.

В листинге 20.1 показан исходный код программы.

Листинг 20.1. Программа *file.asm*

```

; file.asm
section .data
; Выражения, используемые для условного ассемблирования.
    CREATE      equ    1
    OVERWRITE   equ    1
    APPEND      equ    1
    O_WRITE     equ    1
    READ        equ    1
    O_READ      equ    1
    DELETE      equ    1

; Символические имена системных вызовов.
    NR_read     equ    0
    NR_write    equ    1
    NR_open     equ    2
    NR_close    equ    3
    NR_lseek    equ    8
    NR_create   equ    85
    NR_unlink   equ    87

; Флаги создания и состояния файлов.
    O_CREAT     equ    00000100q
    O_APPEND    equ    00002000q

; Режимы доступа.
    O_RDONLY    equ    0000000q
    O_WRONLY    equ    0000001q
    O_RDWR      equ    0000002q

; Режимы при создании файла (права доступа).
    S_IRUSR     equ    004000q      ; Право на чтение для владельца.
    S_IWUSR     equ    002000q      ; Право на запись для владельца.
    NL          equ    0xa
    buflen      equ    64
    fileName    db      "testfile.txt",0
    FD          dq      0           ; Дескриптор файла.

    text1 db      "1. Hello...to everyone!",NL,0
    len1 dq      $-text1-1         ; Удаление 0.
    text2 db      "2. Here I am!",NL,0
    len2 dq      $-text2-1         ; Удаление 0.
    text3 db      "3. Alife and kicking!",NL,0
    len3 dq      $-text3-1         ; Удаление 0.
    text4 db      "Adios !!!",NL,0
    len4 dq      $-text4-1

    error_Create db "error creating file",NL,0
    error_Close  db "error closing file",NL,0
    error_Write  db "error writing to file",NL,0
    error_Open   db "error opening file",NL,0
    error_Append db "error appending to file",NL,0
    error_Delete db "error deleting file",NL,0
    error_Read   db "error reading file",NL,0
    error_Print  db "error printing string",NL,0
    error_Position db "error positioning in file",NL,0

```

```

    success_Create      db "File created and opened",NL,0
    success_Close       db "File closed",NL,NL,0
    success_Write       db "Written to file",NL,0
    success_Open        db "File opened for R/W",NL,0
    success_Append      db "File opened for appending",NL,0
    success_Delete      db "File deleted",NL,0
    success_Read        db "Reading file",NL,0
    success_Position    db "Positioned in file",NL,0

section .bss
    buffer resb bufferlen
section .text
    global main
main:
    push rbp
    mov  rbp, rsp
%IF CREATE
; СОЗДАНИЕ И ОТКРЫТИЕ ФАЙЛА, ЗАТЕМ ЗАКРЫТИЕ. -----
; Создание и открытие файла.
    mov  rdi, fileName
    call createFile
    mov  qword [FD], rax ; Сохранение дескриптора.
; Запись в файл #1.
    mov  rdi, qword [FD]
    mov  rsi, text1
    mov  rdx, qword [len1]
    call writeFile
; Закрытие файла.
    mov  rdi, qword [FD]
    call closeFile
%ENDIF
%IF OVERWRITE
; ОТКРЫТИЕ И ПЕРЕЗАПИСЬ ФАЙЛА, ЗАТЕМ ЗАКРЫТИЕ. -----
; Открытие файла.
    mov  rdi, fileName
    call openFile
    mov  qword [FD], rax ; Сохранение дескриптора файла.
; Запись в файл #2 - ПЕРЕЗАПИСЬ (ЗАМЕЩЕНИЕ)!
    mov  rdi, qword [FD]
    mov  rsi, text2
    mov  rdx, qword [len2]
    call writeFile
; Закрытие файла.
    mov  rdi, qword [FD]
    call closeFile
%ENDIF
%IF APPEND
; ОТКРЫТИЕ И ДОБАВЛЕНИЕ В ФАЙЛ, ЗАТЕМ ЗАКРЫТИЕ. -----
; Открытие файла для добавления в него данных.
    mov  rdi, fileName
    call appendFile
    mov  qword [FD], rax ; Сохранение дескриптора файла.
; Запись в файл #3 - ДОБАВЛЕНИЕ!
    mov  rdi, qword [FD]
    mov  rsi, text3
    mov  rdx, qword [len3]

```

```
    call writeFile
; Заккрытие файла.
    mov     rdi, qword [FD]
    call    closeFile
%ENDIF
%IF O_WRITE
; ОТКРЫТИЕ И ПЕРЕЗАПИСЬ ПО ПОЗИЦИИ СМЕЩЕНИЯ В ФАЙЛЕ, ЗАТЕМ ЗАКРЫТИЕ. ----
; Открытие файла для записи.
    mov     rdi, fileName
    call    openFile
    mov     qword [FD], rax    ; Сохранение дескриптора файла.
; Позиция в файле, определяемая по смещению.
    mov     rdi, qword[FD]
    mov     rsi, qword[len2]   ; Смещение в заданную позицию.
    mov     rdx, 0
    call    positionFile
; Запись в файл в позицию с заданным смещением.
    mov     rdi, qword[FD]
    mov     rsi, text4
    mov     rdx, qword [len4]
    call    writeFile
; Заккрытие файла.
    mov     rdi, qword [FD]
    call    closeFile
%ENDIF
%IF READ
; ОТКРЫТИЕ И ЧТЕНИЕ ИЗ ФАЙЛА, ЗАТЕМ ЗАКРЫТИЕ. -----
; Открытие файла для чтения.
    mov     rdi, fileName
    call    openFile
    mov     qword [FD], rax    ; Сохранение дескриптора файла.
; Чтение из файла.
    mov     rdi, qword [FD]
    mov     rsi, buffer
    mov     rdx, bufferlen
    call    readFile
    mov     rdi, rax
    call    printString
; Заккрытие файла.
    mov     rdi, qword [FD]
    call    closeFile
%ENDIF
%IF O_READ
; ОТКРЫТИЕ И ЧТЕНИЕ В ПОЗИЦИИ, ЗАДАННОЙ СМЕЩЕНИЕМ В ФАЙЛЕ, ЗАТЕМ ЗАКРЫТИЕ. -----
; Открытие файла для чтения.
    mov     rdi, fileName
    call    openFile
    mov     qword [FD], rax    ; Сохранение дескриптора файла.
; Позиция в файле, определяемая по смещению.
    mov     rdi, qword[FD]
    mov     rsi, qword[len2]   ;skip the first line
    mov     rdx, 0
    call    positionFile
; Чтение из файла (в заданной позиции).
    mov     rdi, qword [FD]
    mov     rsi, buffer
```

```

        mov    rdx, 10          ; Количество считываемых символов.
        call   readFile
        mov    rdi, rax
        call   printString
; Заккрытие файла.
        mov    rdi, qword [FD]
        call   closeFile
%ENDIF
%IF DELETE
; УДАЛЕНИЕ ФАЙЛА. -----
; Удаление файла.  ДЛЯ ИСПОЛЬЗОВАНИЯ УБРАТЬ СИМВОЛЫ КОММЕНТАРИЕВ В СЛЕДУЮЩИХ СТРОКАХ.
        mov    rdi, fileName
        call   deleteFile
%ENDIF
leave
ret
; ФУНКЦИИ ОБРАБОТКИ ФАЙЛА. -----
;-----
global readFile
readFile:
        mov    rax, NR_read
        syscall          ; rax содержит количество считываемых символов.
        cmp    rax, 0
        jl     readerror
        mov    byte [rsi+rax], 0 ; Добавление завершающего нуля.
        mov    rax, rsi
        mov    rdi, success_Read
        push   rax        ; Регистр, сохраняемый вызывающей функцией.
        call   printString
        pop    rax        ; Регистр, сохраняемый вызывающей функцией.
        ret
readerror:
        mov    rdi, error_Read
        call   printString
        ret
;-----
global deleteFile
deleteFile:
        mov    rax, NR_unlink
        syscall
        cmp    rax, 0
        jl     deleteerror
        mov    rdi, success_Delete
        call   printString
        ret
deleteerror:
        mov    rdi, error_Delete
        call   printString
        ret
;-----
global appendFile
appendFile:
        mov    rax, NR_open
        mov    rsi,  O_RDWR|O_APPEND
        syscall
        cmp    rax, 0

```



```
    jl     appenderror
    mov    rdi, success_Append
    push   rax      ; Регистр, сохраняемый вызывающей функцией.
    call   printString
    pop    rax      ; Регистр, сохраняемый вызывающей функцией.
    ret

appenderror:
    mov    rdi, error_Append
    call   printString
    ret

;-----
global openFile
openFile:
    mov    rax, NR_open
    mov    rsi, 0_RDWR
    syscall
    cmp    rax, 0
    jl     openererror
    mov    rdi, success_Open
    push   rax      ; Регистр, сохраняемый вызывающей функцией.
    call   printString
    pop    rax      ; Регистр, сохраняемый вызывающей функцией.
    ret

openererror:
    mov    rdi, error_Open
    call   printString
    ret

;-----
global writeFile
writeFile:
    mov    rax, NR_write
    syscall
    cmp    rax, 0
    jl     writeerror
    mov    rdi, success_Write
    call   printString
    ret

writeerror:
    mov    rdi, error_Write
    call   printString
    ret

;-----
global positionFile
positionFile:
    mov    rax, NR_lseek
    syscall
    cmp    rax, 0
    jl     positionerror
    mov    rdi, success_Position
    call   printString
    ret

positionerror:
    mov    rdi, error_Position
    call   printString
    ret

;-----
```

```

global closeFile
closeFile:
    mov    rax, NR_close
    syscall
    cmp    rax, 0
    jl     closeerror
    mov    rdi, success_Close
    call   printString
    ret
closeerror:
    mov    rdi, error_Close
    call   printString
    ret
;-----
global createFile
createFile:
    mov    rax, NR_create
    mov    rsi, S_IRUSR | S_IWUSR
    syscall
    cmp    rax, 0                ; Дескриптор файла в регистре rax.
    jl     createerror
    mov    rdi, success_Create
    push   rax                   ; Регистр, сохраняемый вызывающей функцией.
    call   printString
    pop    rax                   ; Регистр, сохраняемый вызывающей функцией.
    ret
createerror:
    mov    rdi, error_Create
    call   printString
    ret
; ВЫВОД ИНФОРМАЦИИ О ФАЙЛОВЫХ ОПЕРАЦИЯХ (ОБРАТНОЙ СВЯЗИ).
;-----
global printString
printString:
    ; Счетчик символов.
    mov    r12, rdi
    mov    rdx, 0
strLoop:
    cmp    byte [r12], 0
    je     strDone
    inc    rdx                    ; Длина строки в регистре rdx.
    inc    r12
    jmp    strLoop
strDone:
    cmp    rdx, 0                ; Строка отсутствует (длина 0).
    je     prtDone
    mov    rsi, rdi
    mov    rax, 1
    mov    rdi, 1
    syscall
prtDone:
    ret

```

УСЛОВНОЕ АССЕМБЛИРОВАНИЕ

Поскольку программа весьма велика, для упрощения ее анализа используется условное ассемблирование (*conditional assembly*). С этой целью созданы различные переменные, такие как `CREATE`, `WRITE`, `APPEND` и т. п. Если для такой переменной установлено значение 1, то конкретный фрагмент исходного кода, расположенный между строками `%IF 'ИМЯ_ПЕРЕМЕННОЙ'` и `%ENDIF`, будет ассемблирован. Если для переменной установлено значение 0, то программа ассемблирования пропустит этот фрагмент кода. Строки `%IF` и `%ENDIF` называются директивами препроцессора ассемблера (*assembler preprocessor directives*). Начнем с переменной `CREATE` `equ 1` и установим для всех других переменных значение 0, затем ассемблируем, запустим и проанализируем программу. Далее двигаемся постепенно сверху вниз. Продолжаем с установкой значений `CREATE` `equ 1` и `OVERWRITE` `equ 1` (все прочие переменные равны 0) для второй сборки программы и т. д.

NASM предоставляет большой набор директив препроцессора, здесь мы используем директивы условного ассемблирования. Для определения макросов, как описано в главе 18, также применялись директивы препроцессора. В главе 4 руководства по NASM вы найдете полное описание директив препроцессора.

ИНСТРУКЦИИ ДЛЯ ОБРАБОТКИ ФАЙЛОВ

Начнем с создания файла. Необходимо поместить имя файла в регистр `rdi` и вызвать функцию `createFile`. В функции `createFile` символьная переменная `NR_create` записывается в регистр `rax`, а в регистре `rsi` определяются флаги для создания файла. В рассматриваемом здесь примере пользователю (владельцу) предоставляются права на чтение и запись, затем выполняется системный вызов.

Если по каким-либо причинам файл не может быть создан, то функция `createFile` возвращает отрицательное значение в регистре `rax`, и в этом случае необходимо вывести сообщение об ошибке. Если требуются подробности, то отрицательное значение в `rax` определяет тип возникшей ошибки. Если файл создан, то функция возвращает дескриптор файла в `rax`. В вызывающей программе этот дескриптор файла сохраняется в переменной `fd` для дальнейшей работы с файлом. Очевидно, что необходима аккуратность при сохранении содержимого регистра `rax` перед вызовом функции `printString`. Вызов `printString` уничтожает содержимое `rax`, поэтому необходимо записать его в стек перед вызовом функции. В соответствии с соглашениями о вызовах функций `rax` является регистром, сохраняемым вызывающей функцией (*caller saved*).

Далее в рассматриваемом коде в файл записывается некоторый текст, затем файл закрывается. Обратите внимание: операция создания файла создает новый файл, если файл с таким именем уже существует, то он будет удален.

Выполните сборку и запустите программу с условием `CREATE` `equ 1`, другие переменные условного ассемблирования должны быть равны 0. Затем перейдите в командную строку и проверьте, создан ли файл `testfile.txt` и содержится ли в нем сообщение. Если требуется просмотреть содержимое файла в шестнадцатеричном формате, что иногда полезно, то воспользуйтесь командой `xxd testfile.txt` в командной строке.

Продолжайте постепенно присваивать переменным условного ассемблирования значение 1 и проверяйте по содержимому *testfile.txt*, что происходит.

Следует отметить, что в рассматриваемом здесь примере создаются и используются функции без пролога и эпилога. На рис. 20.1 показан итоговый вывод программы при всех переменных условного ассемблирования, установленных в 1.

```
jo@UbuntuDesktop:~/Desktop/linux64/gcc/25 file$ make
nasm -f elf64 -g -F dwarf file.asm -l file.lst
gcc -o file file.o -no-pie
jo@UbuntuDesktop:~/Desktop/linux64/gcc/25 file$ ./file
File created and opened
Written to file
File closed

File opened for reading/(over)writing/updating
Written to file
File closed

File opened for appending
Written to file
File closed

File opened for reading/(over)writing/updating
Positioned in file
Written to file
File closed

File opened for reading/(over)writing/updating
Reading file
2. Here I am!
Adios !!!
3. Alive and kicking!
File closed

File opened for reading/(over)writing/updating
Positioned in file
Reading file
Adios !!!
File closed

File deleted
jo@UbuntuDesktop:~/Desktop/linux64/gcc/25 file$
```

Рис. 20.1. Итоговый вывод программы *file.asm*

РЕЗЮМЕ

В этой главе вы узнали:

- об операциях создания, открытия, закрытия и удаления файла;
- об операциях записи и добавления данных в файл, а также об операции записи в файл в заданной позиции;
- об операции чтения из файла;
- о различных параметрах для работы с файлом.

Глава 21

Командная строка

Иногда необходимо при запуске программы из командной строки указать аргументы, которые будет использовать эта программа. Это может оказаться полезным при разработке собственных инструментальных средств командной строки. Системные администраторы постоянно применяют инструменты командной строки, потому что, как правило, такие инструменты работают быстрее в руках опытного пользователя.

Доступ к аргументам командной строки

В примере программы в листинге 21.1 показано, как можно получить доступ к аргументам командной строки в ассемблерной программе. Здесь рассматриваются самые простые действия: поиск аргументов и их вывод.

Листинг 21.1. Программа *cmdline.asm*

```
;cmdline.asm
extern printf
section .data
    msg    db    "The command and arguments: ",10,0
    fmt    db    "%s",10,0
section .bss
section .text
    global main
main:
push rbp
mov rbp, rsp
    mov r12, rdi    ; Количество аргументов.
    mov r13, rsi    ; Адрес массива аргументов.
; Вывод заголовка.
    mov rdi, msg
    call printf
    mov r14, 0
; Вывод имени команды и аргументов.
.ploop:
    mov rdi, fmt    ; Цикл прохода по массиву аргументов и их вывода.
    mov rsi, qword [r13+r14*8]
    call printf
    inc r14
    cmp r14, r12    ; Достигнуто максимальное количество аргументов?
    jl .ploop
leave
ret
```

При выполнении этой программы количество аргументов, включая имя самой программы, сохраняется в регистре `rdi`. Регистр `rsi` содержит адрес памяти массива, содержащего адреса аргументов командной строки, при этом первым аргументом является имя самой программы. Использование регистров `rdi` и `rsi` соответствует соглашениям о вызовах функций. Напомню, что мы работаем в системе Linux и используем соглашения о вызовах функций System V AMD64 ABI. На других платформах, например в ОС Windows, применяются иные соглашения о вызовах функций. Информация из этих регистров копируется, потому что `rdi` и `rsi` позже будут использоваться для функции `printf`.

В этом коде существует цикл для прохода по массиву аргументов до тех пор, пока не будет достигнуто максимальное число аргументов. В цикле `.ploop` регистр `r13` указывает на массив аргументов. Регистр `r14` используется как счетчик аргументов. На каждой итерации цикла вычисляется адрес следующего аргумента и записывается в `rsi`. Число 8 в выражении `qword [r13+r14*8]` соответствует длине указываемых адресов: 8 байт \times 8 бит = 64-битовый адрес. Регистр `r14` на каждой итерации цикла сравнивается с регистром `r12`, содержащим общее количество аргументов.

На рис. 21.1 показан вывод этой программы, вызванной с некоторыми случайно заданными аргументами.

```
jo@UbuntuDesktop:~/Desktop/linux64/gcc/26 cmdline$ make
nasm -f elf64 -g -F dwarf cmdline.asm -l cmdline.lst
gcc -o cmdline cmdline.o -no-pie
jo@UbuntuDesktop:~/Desktop/linux64/gcc/26 cmdline$ ./cmdline arg1 arg2 abc 5
The command and arguments:
./cmdline
arg1
arg2
abc
5
jo@UbuntuDesktop:~/Desktop/linux64/gcc/26 cmdline$
```

Рис. 21.1. Вывод программы *cmdline.asm*

ОТЛАДКА ПРОГРАММЫ С АРГУМЕНТАМИ КОМАНДНОЙ СТРОКИ

В настоящее время невозможно использовать SASM для отладки программ с аргументами командной строки, поэтому придется воспользоваться отладчиком GDB. Ниже показан один из способов сделать это:

```
gdb --args ./cmdline arg1 arg2 abc 5
break main
run
info registers rdi rsi rsp
```

С помощью приведенных выше инструкций можно проверить, что регистр `rdi` содержит общее количество аргументов (включая имя самой выполняемой команды), а регистр `rsi` указывает на адрес в старшей (верхней) памяти, даже более старшей, чем указатель на стек, как уже было отмечено в главе 8 (см. рис. 8.7). На рис. 21.2 показан вывод отладчика GDB.

```
(gdb) break main
Breakpoint 1 at 0x4004a0: file cmdline.asm, line 9.
(gdb) run
Starting program: /home/jo/Desktop/linux64/gcc/26 cmdline/cmdline arg1 arg2 abc 5

Breakpoint 1, main () at cmdline.asm:9
9      push rbp
(gdb) info registers rdi rsi rsp
rdi            0x5          5
rsi            0x7fffffffde58    140737488346712
rsp            0x7fffffffdd78    0x7fffffffdd78
(gdb) █
```

Рис. 21.2. Вывод отладчика GDB с программой *cmdline*

На рис. 21.2 видно, что массив с адресами аргументов начинается с адреса 0x7fffffffde58. Продолжим исследования, чтобы добраться до действительных значений аргументов. Адрес первого аргумента можно найти с помощью следующей команды:

```
x/1xg 0x7fffffffde58
```

Здесь выполняется запрос к одному огромному слову (8 байт) в шестнадцатеричном формате по адресу 0x7fffffffde58. Ответ показан на рис. 21.3.

```
(gdb) x/1xg 0x7fffffffde58
0x7fffffffde58: 0x00007ffffffe204
(gdb) █
```

Рис. 21.3. GDB возвращает адрес первого аргумента

Теперь посмотрим, что находится по этому адресу (см. рис. 21.4).

```
x/s 0x7fffffffe204
```

```
(gdb) x/s 0x7fffffffe204
0x7fffffffe204: "/home/jo/Desktop/linux64/gcc/26 cmdline/cmdline"
(gdb) █
```

Рис. 21.4. GDB выводит значение первого аргумента

Разумеется, это первый аргумент, т. е. имя самой выполняемой команды. Чтобы найти второй аргумент, нужно увеличить адрес 0x7fffffffde58 на 8 байт, чтобы получить 0x7fffffffde60, затем определить адрес второго аргумента и т. д. На рис. 21.5 показан результат.

```
(gdb) x/1xg 0x7fffffffde60
0x7fffffffde60: 0x00007ffffffe234
(gdb) x/s 0x7fffffffe234
0x7fffffffe234: "arg1"
(gdb) █
```

Рис. 21.5. GDB выводит значение второго аргумента

Таким способом можно отлаживать и проверять программы с аргументами командной строки.

РЕЗЮМЕ

В этой главе вы узнали, как:

- получить доступ к аргументам командной строки;
- использовать регистры для аргументов командной строки;
- отлаживать программы с аргументами командной строки.

Глава 22

Использование ассемблера в коде C

В предыдущих главах иногда для удобства использовались функции языка C, например стандартная функция вывода `printf` или разработанная нами версия `printb`. В этой главе будет показано, как использовать ассемблерные функции в коде на языке C. При этом сразу же становится очевидным важное значение соглашений о вызовах функций. В этой главе используются соглашения о вызове функций System V AMD64 ABI, потому что мы работаем в системе Linux. Для ОС Windows определены другие соглашения о вызовах функций. Если вы внимательно изучили предыдущие главы и примеры исходного кода в них, то эта глава не вызовет у вас затруднений.

Создание файла исходного кода на языке C

Почти весь ассемблерный код должен быть хорошо знаком вам по предыдущим главам. Новым является только код программы на языке C. Здесь вычисляются площадь и периметр прямоугольника и круга. Затем принимается некоторая строка и реверсируется (порядок букв в ней меняется на обратный), наконец, вычисляется сумма элементов массива, элементы массива удваиваются, и вычисляется сумма удвоенных элементов массива. Рассмотрим различные файлы исходного кода.

Начнем с файла исходного кода на языке C, показанного в листинге 22.1.

Листинг 22.1. Программа *fromc.c*

```
// fromc.c

#include <stdio.h>
#include <string.h>

extern int rarea(int, int);
extern int rcircum(int, int);
extern double carea(double);
extern double ccircum(double);
extern void sreverse(char *, int );
extern void adouble(double [], int );
extern double asum(double [], int );
```

```

int main()
{
    char rstring[64];
    int side1, side2, r_area, r_circum;
    double radius, c_area, c_circum;
    double darray[] = {70.0, 83.2, 91.5, 72.1, 55.5};
    long int len;
    double sum;

    // Вызов ассемблерной функции с аргументами типа int.
    printf("Compute area and circumference of a rectangle\n");
    printf("Enter the length of one side : \n");
    scanf("%d", &side1);
    printf("Enter the length of the other side : \n");
    scanf("%d", &side2);

    r_area = rarea(side1, side2);
    r_circum = rcircum(side1, side2);

    printf("The area of the rectangle = %d\n", r_area);
    printf("The circumference of the rectangle = %d\n\n", r_circum);

    // Вызов ассемблерной функции с аргументом типа double (float).
    printf("Compute area and circumference of a circle\n");
    printf("Enter the radius : \n");
    scanf("%lf", &radius);

    c_area = carea(radius);
    c_circum = ccircum(radius);

    printf("The area of the circle = %lf\n", c_area);
    printf("The circumference of the circle = %lf\n\n", c_circum);

    // Вызов ассемблерной функции со строковым аргументом.
    printf("Reverse a string\n");
    printf("Enter the string : \n");
    scanf("%s", rstring);
    printf("The string is = %s\n", rstring);
    sreverse(rstring, strlen(rstring));
    printf("The reversed string is = %s\n\n", rstring);

    // Вызов ассемблерной функции с аргументом-массивом.
    printf("Some array manipulations\n");
    len = sizeof (darray) / sizeof (double);

    printf("The array has %lu elements\n", len);
    printf("The elements of the array are: \n");
    for (int i=0; i<len; i++){
        printf("Element %d = %lf\n", i, darray[i]);
    }

    sum = asum(darray, len);
    printf("The sum of the elements of this array = %lf\n", sum);

    adouble(darray, len);
    printf("The elements of the doubled array are: \n");
    for (int i=0; i<len; i++){
        printf("Element %d = %lf\n", i, darray[i]);
    }

    sum = asum(darray, len);
    printf("The sum of the elements of this doubled array = %lf\n", sum);
    return 0;
}

```

СОЗДАНИЕ ФАЙЛА ИСХОДНОГО КОДА НА АССЕМБЛЕРЕ

Начнем с объявлений ассемблерных функций. Это внешние функции, и для них объявляются типы данных возвращаемых значений и аргументов.

Программа будет предлагать пользователю ввести большинство используемых данных, за исключением массива, значения в котором для удобства мы определим предварительно.

В листингах с 22.2 по 22.7 показаны соответствующие ассемблерные функции.

Листинг 22.2. Функция *rect.asm*

```
;rect.asm
section .data
section .bss
section .text
global rarea
rarea:
    section .text
        push rbp
        mov  rbp, rsp
        mov  rax, rdi
        imul rsi
        leave
        ret
global rcircum
rcircum:
    section .text
        push rbp
        mov  rbp, rsp
        mov  rax, rdi
        add  rax, rsi
        imul rax, 2
        leave
        ret
```

Листинг 22.3. Функция *circle.asm*

```
;circle.asm
section .data
    pi dq 3.141592654
section .bss
section .text
global carea
carea:
    section .text
        push rbp
        mov  rbp, rsp
        movsd xmm1, qword [pi]
        mulsd xmm0, xmm0 ; Радиус в регистре xmm0.
        mulsd xmm0, xmm1
        leave
        ret
global ccircum
ccircum:
    section .text
        push rbp
```

```

mov    rbp, rsp
movsd  xmm1, qword [pi]
addsd  xmm0, xmm0      ; Радиус в регистре xmm0.
mulsd  xmm0, xmm1
leave
ret

```

Листинг 22.4. Функция *sreverse.asm*

```

;sreverse.asm
section .data
section .bss
section .text
global sreverse
sreverse:
push    rbp
mov     rbp, rsp
pushing:
mov     rcx, rsi
mov     rbx, rdi
mov     r12, 0
pushLoop:
mov     rax, qword [rbx+r12]
push    rax
inc     r12
loop    pushLoop
popping:
mov     rcx, rsi
mov     rbx, rdi
mov     r12, 0
popLoop:
pop     rax
mov     byte [rbx+r12], al
inc     r12
loop    popLoop
mov     rax, rdi
leave
ret

```

Листинг 22.5. Функция *asum.asm*

```

; asum.asm
section .data
section .bss
section .text
global asum
asum:
    section .text
; Вычисление суммы.
mov     rcx, rsi      ; Длина массива.
mov     rbx, rdi      ; Адрес массива.
mov     r12, 0
movsd   xmm0, qword [rbx+r12*8]
dec     rcx           ; На один проход в цикле меньше, так как
                        ; первый элемент уже в регистре xmm0.
sloop:
inc     r12

```

```

        addsd xmm0, qword [rbx+r12*8]
        loop sloop
ret      ; Сумма возвращается в регистре xmm0.

```

Листинг 22.6. Функция *adouble.asm*

```

; adouble.asm
section .data
section .bss
section .text
global adouble
adouble:
    section .text
; Удвоение элементов.
    mov     rcx, rsi      ; Длина массива.
    mov     rbx, rdi      ; Адрес массива.
    mov     r12, 0
aloop:
    movsd   xmm0, qword [rbx+r12*8]    ; Взять элемент.
    addsd   xmm0, xmm0                ; Удвоить его.
    movsd   qword [rbx+r12*8], xmm0    ; Поместить результат обратно в массив.
    inc     r12
    loop    aloop
ret

```

Листинг 22.7. *makefile*

```

fromc: fromc.c rect.o circle.o sreverse.o adouble.o asum.o
    gcc -o fromc fromc.c rect.o circle.o sreverse.o \
        adouble.o asum.o -no-pie
rect.o: rect.asm
    nasm -f elf64 -g -F dwarf rect.asm -l rect.lst
circle.o: circle.asm
    nasm -f elf64 -g -F dwarf circle.asm -l circle.lst
sreverse.o: sreverse.asm
    nasm -f elf64 -g -F dwarf sreverse.asm -l sreverse.lst
adouble.o: adouble.asm
    nasm -f elf64 -g -F dwarf adouble.asm -l adouble.lst
asum.o: asum.asm
    nasm -f elf64 -g -F dwarf asum.asm -l asum.lst

```

В ассемблерном коде нет ничего особенного – просто нужно внимательно следить за типами данных переменных, принимаемых из вызывающей C-программы. Ассемблерные функции принимают аргументы из вызывающей программы и сохраняют их в регистрах в соответствии с соглашениями о вызовах функций. В вызывающую программу результаты возвращаются в регистре `rax` (целочисленное значение) или в регистре `xmm0` (значение с плавающей точкой). Теперь вы можете разрабатывать собственные библиотеки функций для использования в программах на ассемблере или на C, а благодаря соглашениям о вызовах функций не нужно беспокоиться о том, как передаются аргументы. Просто внимательно следите за использованием правильных типов данных.

Обратите внимание: в *makefile* применяется символ обратного следа (`\`) для разделения длинных строк, а для смещения инструкций вправо используется символ табуляции.

На рис. 22.1 показан вывод этой программы.

```
jo@ubuntu18:~/Desktop/Book/27 fromc$ ./fromc
Compute area and circumference of a rectangle
Enter the length of one side :
2
Enter the length of the other side :
3
The area of the rectangle = 6
The circumference of the rectangle = 10

Compute area and circumference of a circle
Enter the radius :
10
The area of the circle = 314.159265
The circumference of the circle = 62.831853

Reverse a string
Enter the string :
abcde
The string is = abcde
The reversed string is = edcba

Double the elements of an array
The array has 5 elements
The elements of the array are:
Element 0 = 70.000000
Element 1 = 83.200000
Element 2 = 91.500000
Element 3 = 72.100000
Element 4 = 55.500000
The sum of the elements of this array = 372.300000
The elements of the doubled array are:
Element 0 = 140.000000
Element 1 = 166.400000
Element 2 = 183.000000
Element 3 = 144.200000
Element 4 = 111.000000
The sum of this doubled array = 744.600000
jo@ubuntu18:~/Desktop/Book/27 fromc$ █
```

Рис. 22.1. Вывод программы *fromc.c*

РЕЗЮМЕ

В этой главе вы узнали:

- как вызывать ассемблерные функции из исходного кода на языке более высокого уровня, в данном случае из исходного кода на языке C;
- о важном значении соглашений о вызовах функций.

Глава 23

Встроенный ассемблер

В этой главе используется язык программирования C для объяснения смысла встроенного (inline) ассемблерного кода. Существует возможность писать инструкции на ассемблере в исходном коде C-программы. В большинстве случаев такой подход не рекомендуется, так как современные компиляторы языка C настолько хорошо спроектированы и реализованы, что вы должны быть весьма высокопрофессиональным программистом на ассемблере, чтобы действительно улучшить производительность кода на C. В действительности применение встроенного ассемблерного кода делает более сложной для компиляторов C или C++ оптимизацию кода с такими ассемблерными вставками.

Кроме того, компилятор языка C не найдет какие-либо ошибки в таких ассемблерных инструкциях – вам придется самостоятельно искать и исправлять их. Более того, доступ к памяти и регистрам, используемым C-программой, может создавать дополнительные потенциальные опасности. Тем не менее во многих статьях в интернете встроенный ассемблерный код используется для описания функциональности на более низком уровне, так что умение читать такой код может оказаться полезным.

Существует два типа встроенного ассемблерного кода: простой и расширенный.

ПРОСТОЙ ВСТРОЕННЫЙ АССЕМБЛЕРНЫЙ КОД

Начнем с примера простого встроенного ассемблерного кода. См. листинги 23.1 и 23.2.

Листинг 23.1. Программа *inline1.c*

```
// inline1.c
#include <stdio.h>

int x=11, y=12, sum, prod;
int subtract(void);
void multiply(void);

int main(void)
{
    printf("The numbers are %d and %d\n",x,y);
    __asm__(
        ".intel_syntax noprefix;"
```

```

        "mov rax,x;"
        "add rax,y;"
        "mov sum,rax"
    );
    printf("The sum is %d.\n",sum);
    printf("The difference is %d.\n",subtract());
    multiply();
    printf("The product is %d.\n",prod);
}

int subtract(void)
{
    __asm__(
        ".intel_syntax noprefix;"
        "mov rax,x;"
        "sub rax,y"           // Возвращаемое значение в rax.
    );
}

void multiply(void)
{
    __asm__(
        ".intel_syntax noprefix;"
        "mov rax,x;"
        "imul rax,y;"
        "mov prod,rax"       // Нет возвращаемого значения, результат в prod.
    );
}

```

Листинг 23.2. *makefile*

```

# makefile inline1.c
inline1: inline1.c
    gcc -o inline1 inline1.c -masm=intel -no-pie

```

Обратите внимание на дополнительный параметр компиляции в *makefile*: `-masm=intel`. Этот параметр обязателен при использовании встроенного ассемблерного кода.

В рассматриваемом здесь примере показан вызов простой (*basic*) встроенной ассемблерной подпрограммы. В основную программу `main` добавляются две переменные, затем вызывается функция вычитания этих двух переменных. Далее вызывается другая функция для умножения тех же переменных. Если требуется доступ к таким переменным в простой встроенной ассемблерной подпрограмме, то необходимо объявить их как глобальные, т. е. вынести их объявление за пределы любой функции. Если такие переменные не объявлены глобальными, то компилятор `gcc` сообщит, что не может найти их. Но глобальные переменные являются потенциальным источником ошибок, таких как конфликты имен. Кроме того, при изменении регистров в ассемблерном коде, возможно, потребуется их сохранение перед вызовом встроенного ассемблерного кода и восстановление их исходных значений при выходе из встроенного ассемблерного кода, иначе возникает опасность аварийного завершения программы. Регистры, изменяемые встроенным ассемблерным кодом, называются затираемыми (*clobber*) регистрами.

В ассемблерной части, заключенной в блоке `__asm__(...)`, первая инструкция определяет, что здесь используется синтаксис ассемблера Intel без префиксов. (Вспомните обсуждение вариантов синтаксиса Intel и AT&T в главе 3.) Затем используются инструкции ассемблера как обычно, завершаемые символом `;` или `\n`. Последнюю ассемблерную инструкцию не обязательно завершать символом `;` или `\n`. Обратите внимание на использование глобальных переменных. Нам повезло, потому что затирание (*clobbering*) регистров не привело к аварийному завершению программы. Чтобы избежать затирания регистров и применения глобальных переменных, необходимо воспользоваться расширенным встроенным ассемблерным кодом, рассматриваемым в следующем разделе.

На рис. 23.1 показан вывод этой программы.

```
jo@UbuntuDesktop:~/Desktop/linux64/gcc/28 inline 1$ make
gcc -o inline1 inline1.c -masm=intel
jo@UbuntuDesktop:~/Desktop/linux64/gcc/28 inline 1$ ./inline1
The numbers are 11 and 12
The sum is 23.
The difference is -1.
The product is 132.
jo@UbuntuDesktop:~/Desktop/linux64/gcc/28 inline 1$ █
```

Рис. 23.1. Вывод программы *inline1.c*

РАСШИРЕННЫЙ ВСТРОЕННЫЙ АССЕМБЛЕРНЫЙ КОД

В листингах 23.3 и 23.4 показан пример расширенного встроенного ассемблерного кода.

Листинг 23.3. Программа *inline2.c*

```
// inline2.c

#include <stdio.h>

int a=12;    // Глобальные переменные.
int b=13;
int bsum;

int main(void)
{
    printf("The global variables are %d and %d\n",a,b);

    __asm__(
        ".intel_syntax noprefix\n"
        "mov rax,a\n"
        "add rax,b\n"
        "mov bsum,rax\n"
        ::: "rax"
    );

    printf("The extended inline sum of global variables is %d.\n\n", bsum);
    int x=14,y=16, esum, eproduct, edif;    // Локальные переменные.
    printf("The local variables are %d and %d\n",x,y);
```

```

__asm__(
    ".intel_syntax noprefix;"
    "mov rax,rdx;"
    "add rax,rcx;"
    : "a"(esum)
    : "d"(x), "c"(y)
    );
printf("The extended inline sum is %d.\n", esum);

__asm__(
    ".intel_syntax noprefix;"
    "mov rbx,rdx;"
    "imul rbx,rcx;"
    "mov rax,rbx;"
    : "a"(eproduct)
    : "d"(x), "c"(y)
    : "rbx"
    );
printf("The extended inline product is %d.\n", eproduct);

__asm__(
    ".intel_syntax noprefix;"
    "mov rax,rdx;"
    "sub rax,rcx;"
    : "a"(edif)
    : "d"(x), "c"(y)
    );
printf("The extended inline asm difference is %d.\n", edif);
}

```

Листинг 23.4. *makefile*

```

# makefile inline2.c
inline2: inline2.c
    gcc -o inline2 inline2.c -masm=intel -no-pie

```

Здесь ассемблерные инструкции выглядят по-другому – для них используется шаблон, показанный ниже:

```

asm (
    ассемблерный код
    : операнды вывода                /* необязательные */
    : операнды ввода                 /* необязательные */
    : список затираемых регистров    /* необязательный */
    );

```

После ассемблерного кода записывается используемая дополнительная, но не обязательная информация. В качестве примера рассмотрим встроенный код вычисления произведения (повторно воспроизведенный ниже):

```

__asm__(
    ".intel_syntax noprefix;"
    "mov rbx,rdx;"
    "imul rbx,rcx;"
    "mov rax,rbx;"

```

```

: "a"(eproduct)
: "d"(x), "c"(y)
: "rbx"
);
printf("The extended inline product is %d.\n", eproduct);

```

Каждая дополнительная (но не обязательная) строка начинается с символа двоеточия (:), при этом непременно должен соблюдаться порядок инструкций. Элементы *a*, *d* и *c* называются регистровыми ограничениями (*register constraints*) – они отображаются в регистры *rax*, *rdx* и *rcx* соответственно. Ниже показано, как регистровые ограничения отображаются в соответствующие регистры.

```

a -> rax, eax, ax, al
b -> rbx, ebx, bx, bl
c -> rcx, ecx, cx, cl
d -> rdx, edx, dx, dl
S -> rsi, esi, si
D -> rdi, edi, di
r -> любой регистр

```

Выражение `: "a"(eproduct)` в первой дополнительной строке означает, что вывод должен находиться в регистре *rax*, а *rax* ссылается на переменную *eproduct*. Регистр *rdx* ссылается на *x*, регистр *rcx* ссылается на *y*, т. е. на переменные ввода.

В завершение следует отметить, что регистр *rbx* считается затираемым в приведенном коде, поэтому необходимо восстанавливать его исходное значение, потому что он был объявлен в списке затираемых (*clobbered*) регистров. В рассматриваемом здесь примере оставление этого регистра в затертом состоянии не приводит к аварийному завершению программы, но такой подход применен в программе только в демонстрационных целях. Намного больше информации о встроенном ассемблерном коде можно найти в интернете, но, как уже было отмечено ранее, необходимо использовать встроенный ассемблер только в особых случаях. Также следует помнить о том, что применение встроенного ассемблерного кода сделает ваш код на языке C менее переносимым. См. рис. 23.2.

```

jo@UbuntuDesktop:~/Desktop/linux64/gcc/29 inline 2$ make
gcc -o inline2 inline2.c -masm=intel
jo@UbuntuDesktop:~/Desktop/linux64/gcc/29 inline 2$ ./inline2
The global variables are 12 and 13
The extended inline asm sum of global variables is 25.

The local variables are 14 and 16
The extended inline asm sum is 30.
The extended inline product is 224.
The extended inline asm difference is -2.
jo@UbuntuDesktop:~/Desktop/linux64/gcc/29 inline 2$ █

```

Рис. 23.2. Вывод программы *inline2.c*

В следующих главах будет рассматриваться использование ассемблера в ОС Windows. Полезно знать о том, что встроенный ассемблерный код не поддерживается для процессоров x64 в Visual Studio, поддержка обеспечивается только для процессоров x86. Но компилятор gcc не имеет такого ограничения.

РЕЗЮМЕ

В этой главе вы узнали об использовании:

- простого встроенного ассемблерного кода;
- расширенного встроенного ассемблерного кода.

Глава 24

Строки

С точки зрения человека строки обычно представляются в виде последовательности символов, образующих слова и фразы, которые мы можем понять. Но в языке ассемблера любой список или массив непрерывных локаций памяти считается строкой вне зависимости от того, может ли понять ее человек. Ассемблер предоставляет несколько мощных инструкций для обработки таких блоков данных эффективным способом. В приведенных ниже примерах используются читаемые символы, но следует помнить, что в действительности ассемблер не заботится о том, чтобы символы были читаемыми. Здесь будет показано, как можно перемещать строки в другие локации, как сканировать и сравнивать строки.

Какими бы мощными ни были эти инструкции обработки строк, более эффективная функциональность будет продемонстрирована при обсуждении SIMD-инструкций в следующих главах. Но сейчас мы начнем с самых простых инструкций.

ОБРАБОТКА СТРОК

В листинге 24.1 показан пример исходного кода.

Листинг 24.1. Программа *move_strings.asm*

```
; move_strings.asm
%macro prnt 2
    mov     rax, 1          ; 1 = запись.
    mov     rdi, 1          ; 1 = в стандартный поток вывода stdout.
    mov     rsi, %1
    mov     rdx, %2
    syscall
    mov rax, 1
    mov rdi, 1
    mov rsi, NL
    mov rdx, 1
    syscall
%endmacro

section .data
    length equ 95
    NL db 0xa
```

```

    string1 db "my_string of ASCII:"
    string2 db 10,"my_string of zeros:"
    string3 db 10,"my_string of ones:"
    string4 db 10,"again my_string of ASCII:"
    string5 db 10,"copy my_string to other_string:"
    string6 db 10,"reverse copy my_string to other_string:"
section .bss
    my_string resb length
    other_string resb length
section .text
    global main
main:
push rbp
mov rbp, rsp
;-----
; Заполнение строки видимыми ascii-символами.
    prnt string1,18
    mov rax,32
    mov rdi,my_string
    mov rcx, length
str_loop1: mov byte[rdi], al      ; Простой метод.
            inc rdi
            inc al
            loop str_loop1
    prnt my_string,length
;-----
; Заполнение строки ascii-символами 0.
    prnt string2,20
    mov rax,48
    mov rdi,my_string
    mov rcx, length
str_loop2: stosb                ; Здесь уже не нужна инструкция inc rdi.
            loop str_loop2
    prnt my_string,length
;-----
; Заполнение строки ascii-символами 1.
    prnt string3,19
    mov rax, 49
    mov rdi,my_string
    mov rcx, length
    rep stosb ; Здесь уже не нужна инструкция inc rdi и не нужен цикл.
    prnt my_string,length
;-----
; Повторное заполнение строки видимыми ascii-символами.
    prnt string4,26
    mov rax,32
    mov rdi,my_string
    mov rcx, length
str_loop3: mov byte[rdi], al      ; Простой метод.
            inc rdi
            inc al
            loop str_loop3
    prnt my_string,length
;-----
; Копирование строки my_string в другую строку other_string.
    prnt string5,32

```

```

mov rsi,my_string      ; Источник в rsi.
mov rdi,other_string   ; Цель в rdi.
mov rcx, length
rep movsb
prnt other_string,length
;-----
; Копирование в обратном порядке строки my_string в другую строку other_string.
prnt string6,40
mov rax, 48             ; Очистка строки other_string.
mov rdi,other_string
mov rcx, length
rep stosb
lea rsi,[my_string+length-4]
lea rdi,[other_string+length]
mov rcx, 27             ; Копирование только 27-1 символов.
std                     ; std устанавливает флаг DF, cld сбрасывает флаг DF.
rep movsb
prnt other_string,length

leave
ret

```

В этой программе применяется макрос (более подробно о макросах см. главу 18) для вывода строк, но можно было бы применить функцию языка C `printf`, которую мы использовали многократно в предыдущих примерах.

Мы начинаем с создания строки, содержащей 95 видимых символов из таблицы ASCII, – код первого символа строки 32 (пробел), код последнего символа 126 (тильда ~). В этих символах нет ничего необычного. Сначала выводится заголовок, потом ASCII-код первого символа записывается в регистр `rax`, а регистр `rdi` указывает на адрес в памяти строки `my_string`. Затем длина формируемой строки передается в регистр `rcx` для использования в цикле. В этом цикле один ASCII-код копируется из регистра `al` в строку `my_string`, затем выполняется переход к следующему ASCII-коду, который записывается по очередному адресу памяти в `my_string` и т. д. После завершения цикла выводится полученная строка. И здесь нет ничего нового.

В следующей части программы все содержимое строки `my_string` заменяется полностью на символы 0 (ASCII-код 48). Длина строки снова записывается в регистр `rcx` для организации цикла. Далее используется инструкция `stosb` для записи символов 1 (ASCII-код 49) в строку `my_string`. Для инструкции `stosb` необходим только начальный адрес строки в регистре `rdi` и записываемый символ в регистре `rax`, и `stosb` автоматически переходит к следующему адресу памяти на каждом шаге цикла. В этом случае можно не заботиться об увеличении содержимого регистра `rdi`.

Далее в программе принимаются дополнительные меры по исключению цикла с использованием регистра `rcx`. Применяется инструкция `rep stosb` для повторения `stosb` заданное количество раз. Количество повторений содержится в регистре `rcx`. Это весьма эффективный способ инициализации памяти.

Продолжаем работать с содержимым памяти. Строго говоря, здесь будет выполняться копирование блоков памяти, а не перемещение их содержимого. Сначала строка снова инициализируется ASCII-кодами видимых символов. Можно было бы оптимизировать эту часть кода, применив макрос или

функцию вместо простого повторения фрагмента исходного кода. Затем начинается копирование строки/блока памяти: из `my_string` в `other_string`. Адрес строки-источника помещается в регистр `rsi`, адрес целевой строки записывается в регистр `rdi`. Это легко запомнить, так как буква *s* в имени регистра `rsi` обозначает *source* (источник), а буква *d* в имени регистра `rdi` – *destination* (цель). Затем выполняется инструкция `rep movsb`, и все сделано. Инструкция `rep` продолжает копирование до тех пор, пока содержимое `rcx` не станет равным 0.

В последней части программы выполняется перемещение (копирование) содержимого памяти в обратном порядке. Эта концепция может показаться несколько более сложной, поэтому ниже объясняются некоторые подробности. При использовании инструкции `movsb` учитывается значение `DF` (флага направления). Если `DF=0`, то значения в регистрах `rsi` и `rdi` увеличиваются на 1, указывая на следующий более старший адрес памяти. Если `DF=1`, то значения `rsi` и `rdi` уменьшаются на 1, указывая на следующий более младший адрес памяти. Это означает, что в рассматриваемом здесь примере при `DF=1` необходимо, чтобы регистр `rsi` указывал на самый старший адрес памяти в копируемом блоке и уменьшался, начиная с этого адреса. При копировании предполагается «проход в обратном порядке», т. е. уменьшение `rsi` и `rdi` на каждом шаге цикла. Будьте внимательны: уменьшаются оба регистра `rsi` и `rdi`, так как невозможно использовать флаг `DF` для увеличения одного регистра и уменьшения другого (т. е. для реверсирования строки). В рассматриваемом здесь примере копируется не вся строка полностью, а только алфавитные символы в нижнем регистре, которые размещаются в старших адресах памяти в целевой строке. Инструкция `lea rsi, [my_string+length-4]` загружает эффективный адрес строки `my_string` в регистр `rsi` и пропускает четыре символа, которые не являются алфавитными. Флаг `DF` можно установить в 1 инструкцией `std` и сбросить в 0 инструкцией `cld`. Затем вызывается мощная инструкция `rep movsb`, и задача выполнена.

Почему в регистр `rcx` записывается значение 27, хотя обрабатываются 26 символов? Оказывается, инструкция `rep` уменьшает `rcx` на 1 перед выполнением любых операций внутри цикла. Это можно проверить с помощью отладчика, например `SASM`. Перед началом отладки закомментируйте все ссылки на макрос `rgnt`, чтобы избежать проблем. Вы увидите, что `SASM` позволяет войти в цикл `rep` и проверить память и регистры. Разумеется, можно также обратиться к руководствам Intel для получения более подробной информации об инструкции `rep` – в подразделе *Operation* вы найдете описание, приблизительно соответствующее приведенному ниже:

```
IF AddressSize = 16
    THEN
        Use CX for CountReg;
        Implicit Source/Dest operand for memory use of SI/DI;
    ELSE IF AddressSize = 64
        THEN Use RCX for CountReg;
        Implicit Source/Dest operand for memory use of RSI/RDI;
    ELSE
        Use ECX for CountReg;
        Implicit Source/Dest operand for memory use of ESI/EDI;
```



```

string1    db "This is the 1st string.",10,0
string2    db "This is the 2nd string.",10,0
strlen2    equ $-string2-2
string21    db "Comparing strings: The strings do not differ.",10,0
string22    db "Comparing strings: The strings differ, "
            db "starting at position: %d.",10,0

string3     db "The quick brown fox jumps over the lazy dog.",0
strlen3     equ $-string3-2
string33    db "Now look at this string: %s",10,0
string4     db "z",0
string44    db "The character '%s' was found at position: %d.",10,0
string45    db "The character '%s' was not found.",10,0
string46    db "Scanning for the character '%s'." ,10,0

section .bss
section .text
    global main
main:
    push    rbp
    mov     rbp, rsp
; Вывод 2 строк.
    xor     rax, rax
    mov     rdi, string1
    call    printf
    mov     rdi, string2
    call    printf
; Сравнение 2 строк. -----
    lea     rdi, [string1]
    lea     rsi, [string2]
    mov     rdx, strlen2
    call    compare1
    cmp     rax, 0
    jnz     not_equal1
; Строки одинаковы, вывод.
    mov     rdi, string21
    call    printf
    jmp     otherversion
; Строки не одинаковы, вывод.
not_equal1:
    mov     rdi, string22
    mov     rsi, rax
    xor     rax, rax
    call    printf
; Сравнение 2 строк, другая версия. -----
otherversion:
    lea     rdi, [string1]
    lea     rsi, [string2]
    mov     rdx, strlen2
    call    compare2
    cmp     rax, 0
    jnz     not_equal2
; Строки одинаковы, вывод.
    mov     rdi, string21
    call    printf
    jmp     scanning
; Строки не одинаковы, вывод.
not_equal2:

```

```

    mov     rdi, string22
    mov     rsi, rax
    xor     rax, rax
    call    printf
; Поиск символа (сканирование) в строке. -----
; Сначала вывод всей строки.
    mov     rdi, string33
    mov     rsi, string3
    xor     rax, rax
    call    printf
; Затем вывод искомого аргумента, которым может быть только 1 символ.
    mov     rdi, string46
    mov     rsi, string4
    xor     rax, rax
    call    printf
scanning:
    lea     rdi, [string3]      ; Строка.
    lea     rsi, [string4]      ; Искомый аргумент.
    mov     rdx, strlen3
    call    cscan
    cmp     rax, 0
    jz      char_not_found
; Символ найден, вывод.
    mov     rdi, string44
    mov     rsi, string4
    mov     rdx, rax
    xor     rax, rax
    call    printf
    jmp     exit
; Символ не найден, вывод.
char_not_found:
    mov     rdi, string45
    mov     rsi, string4
    xor     rax, rax
    call    printf
exit:
leave
ret

; ФУНКЦИИ =====
; Функция сравнения 2 строк. -----
compare1:  mov     rcx, rdx
           cld
cmpg:      cmpsb
           jne     notequal
           loop    cmpg
           xor     rax, rax
           ret
notequal:  mov     rax, strlen2
           dec     rcx           ; Вычисление положения в строке.
           sub     rax, rcx      ; Вычисление положения в строке.
           ret
           xor     rax, rax
           ret
; -----
; Функция сравнения 2 строк. -----

```

```

compare2:  mov     rcx, rdx
           cld
           repe   cmpsb
           je     equal2
           mov     rax, strlen2
           sub     rax, rcx          ; Вычисление положения в строке.
           ret

equal2:    xor     rax, rax
           ret
;-----
; Функция поиска символа в строке (сканирования).
cscan:     mov     rcx, rdx
           lodsb
           cld
           repne  scasb
           jne    char_notfound
           mov     rax, strlen3
           sub     rax, rcx          ; Вычисление положения в строке.
           ret

char_notfound:  xor     rax, rax
           ret

```

Здесь рассматриваются две версии функции сравнения строк. Как и в предыдущей программе, адрес первой строки (источника) записывается в регистр `rsi`, адрес второй строки (целевой) – в регистр `rdi`, а длина строк помещается в регистр `rcx`. Для полной уверенности флаг направления `DF` сбрасывается в 0 инструкцией `cld`. Итак, выполняется проход по строкам в прямом направлении.

Инструкция `cmpsb` сравнивает два байта и устанавливает флаг состояния `ZF` в 1, если сравниваемые байты равны, или сбрасывает `ZF` в 0, если эти два байта не равны.

Использование флага `ZF` может показаться слегка запутанным. Если `ZF=1`, это значит, что при выполнении самой последней предыдущей инструкции был получен результат 0 (байты равны). Если `ZF=0`, это значит, что предыдущая инструкция выполнена с результатом, не равным 0 (байты не равны). Таким образом, необходимо определять, когда флаг `ZF` становится равным 0. Для проверки `ZF` и продолжения выполнения на основе результата этой проверки требуется несколько инструкций условного перехода, как показано ниже:

- `jz` – переход, если результат равен нулю (`ZF=1`);
 - ♦ равнозначная инструкция `je` – переход, если равно (`ZF=1`) (байты равны);
- `jnz` – переход, если результат не равен нулю (`ZF=0`);
 - ♦ равнозначная инструкция `jne` – переход, если равно (`ZF=0`) (байты не равны).

Регистры `rsi` и `rdi` увеличиваются инструкцией `cmpsb`, если флаг `DF` не установлен (равен 0), и уменьшаются, если флаг `DF` установлен (равен 1). Здесь создан цикл, который выполняется до тех пор, пока флаг `ZF` не станет равным 0. Когда `ZF` становится равным 0, происходит выход из цикла и начинается вычисление позиции отличающегося символа на основе значения в регистре `rcx`. Но значение `rcx` изменяется только в конце цикла, до которого дело не доходит, поэтому необходимо скорректировать содержимое `rcx` (уменьшить его на 1). Полученная в результате позиция символа возвращается в основную программу в регистре `rax`.

Во второй версии функции сравнения используется инструкция *gere* – вариант инструкции *гер*. Аббревиатура *gere* означает «repeat while equal» (повторять, пока равно). Как и в первой версии, инструкция *cmpsb* устанавливает флаг ZF по результату сравнения, и ZF=1 означает, что байты равны. Как только *cmpsb* установит флаг ZF равным 0, цикл *gere* завершается, и регистр *gsi* можно использовать для вычисления позиции, в которой обнаружен отличающийся символ. Если строки абсолютно одинаковые, то в регистре *gsi* находится значение 0, а флаг ZF установлен в 1. После завершения цикла *gere* инструкция *je* проверяет флаг ZF на равенство 1. Если ZF равен 1, то строки равны (одинаковы), если ZF равен 0, то строки не равны (не одинаковы). Регистр *gsi* используется для вычисления позиции отличающегося символа, но здесь нет необходимости в коррекции *gsi*, потому что инструкция *gere* увеличивает значение *gsi* в самом начале каждой итерации цикла.

Поиск символа (сканирование) работает аналогично, но вместо инструкции *gere* используется инструкция *gerne* – «repeat while not equal» (повторять, пока не равно). Также применяется инструкция *lods*, которая загружает байт из ячейки памяти с адресом, хранящимся в *rsi*, в регистр *rax*. Инструкция *scasb* сравнивает байт в регистре *al* (младший байт регистра *rax*) с байтом, на который указывает регистр *rdi*, и устанавливает (1=равен) или сбрасывает (0=не равен) флаг ZF соответствующим образом. Инструкция *gerne* проверяет состояние флага и продолжает выполнение, если ZF=0, т. е. если два сравниваемых байта не равны. Если байты равны, то *scasb* устанавливает флаг ZF в 1, цикл *gerne* прекращает выполнение, и регистр *gsi* можно использовать для вычисления позиции найденного байта в строке.

Сканирование строки (поиск символа) работает только с одним символом, заданным как искомый аргумент. Если у вас возник вопрос: как использовать в качестве искомого аргумента строку, то вам придется выполнять поиск (сканирование) поочередно, символ за символом. Но лучше подождать, пока мы доберемся до глав, в которых рассматриваются SIMD-инструкции.

На рис. 24.2 показан вывод этой программы.

```
jo@UbuntuDesktop:~/Desktop/linux64/gcc/31 strings 2$ make
nasm -f elf64 -g -F dwarf strings.asm -l strings.lst
gcc -o strings strings.o
jo@UbuntuDesktop:~/Desktop/linux64/gcc/31 strings 2$ ./strings
This is the 1st string.
This is the 2nd string.
Comparing strings: The strings differ, starting at position: 13.
Now look at this string: The quick brown fox jumps over the lazy dog.
Scanning for the character 'z'.
The character 'z' was found at position: 38.
jo@UbuntuDesktop:~/Desktop/linux64/gcc/31 strings 2$ █
```

Рис. 24.2. Вывод программы *strings.asm*

РЕЗЮМЕ

В этой главе вы узнали, как:

- перемещать и копировать блоки памяти весьма эффективным способом;
- использовать инструкции *movsb* и *гер*;
- сравнивать и выполнять поиск символа (сканирование) в блоках памяти;
- использовать инструкции *cmpsb*, *scasb*, *gere* и *gerne*.

Глава 25

.....

Предъявите ваш идентификатор

Иногда необходимо определить функциональность, предоставляемую некоторым процессором. Например, в программе можно попытаться определить наличие или отсутствие конкретной версии SSE-инструкций. В следующей главе будут использоваться программы с SSE-инструкциями, поэтому необходимо знать, какая именно версия SSE поддерживается процессором. Для проверки характеристик ЦПУ существует специальная инструкция: `cpuid`.

ИСПОЛЬЗОВАНИЕ ИНСТРУКЦИИ `CPUID`

Сначала специальный параметр записывается в регистр `eax`, затем выполняется инструкция `cpuid`, после этого проверяется возвращаемое значение в регистрах `ecx` и `edx`. Вполне очевидно, что инструкция `cpuid` использует 32-битовые регистры.

Объем информации, который можно получить с помощью инструкции `cpuid`, ошеломляет. В руководствах Intel (<https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf>) в томе 2A (Volume 2A) найдите описание инструкции `cpuid`. Там вы увидите несколько таблиц, содержащих описание характеристик, возвращаемых в регистре `ecx` при выполнении инструкции `cpuid` с определенным значением в регистре `eax`. Это только часть информации, которую можно извлечь, в другой таблице описана информация, возвращаемая в регистре `edx`. Внимательно изучите это руководство Intel, чтобы узнать обо всех возможностях.

Рассмотрим пример определения функциональных характеристик SSE-инструкций, которые потребуются нам в следующей главе. В руководстве Intel указано, что можно использовать биты 0, 19 и 20 регистра `ecx` и биты 25 и 26 регистра `edx` для определения версии SSE, реализованной в данном процессоре.

В листинге 25.1 показан пример программы.

Листинг 25.1. Программа *cpu.asm*

```
; cpu.asm
extern printf
section .data
    fmt_no_sse db "This cpu does not support SSE",10,0
    fmt_sse42 db "This cpu supports SSE 4.2",10,0
    fmt_sse41 db "This cpu supports SSE 4.1",10,0
    fmt_ssse3 db "This cpu supports SSSE 3",10,0
    fmt_sse3 db "This cpu supports SSE 3",10,0
    fmt_sse2 db "This cpu supports SSE 2",10,0
    fmt_sse db "This cpu supports SSE",10,0
section .bss
section .text
    global main
main:
    push rbp
    mov rbp, rsp
    call cpu_sse    ; Возвращает 1 в гак, если поддерживается sse, иначе 0.
    leave
    ret

cpu_sse:
    push rbp
    mov rbp, rsp
    xor r12, r12    ; Флаг SSE доступен.
    mov eax, 1      ; Запрос флагов характеристик ЦПУ.
    cpushid

; Проверка поддержки SSE.
    test edx, 2000000h    ; Проверка бита 25 (SSE).
    jz sse2               ; SSE-инструкции доступны.
    mov r12, 1
    xor rax, rax
    mov rdi, fmt_sse
    push rcx              ; Изменяется функцией printf.
    push rdx              ; Сохранение результата cpushid.
    call printf
    pop rdx
    pop rcx

sse2:
    test edx, 4000000h    ; Проверка бита 26 (SSE 2).
    jz sse3               ; Версия SSE 2 доступна.
    mov r12, 1
    xor rax, rax
    mov rdi, fmt_sse2
    push rcx              ; Изменяется функцией printf.
    push rdx              ; Сохранение результата cpushid.
    call printf
    pop rdx
    pop rcx

sse3:
    test ecx, 1           ; Проверка бита 0 (SSE 3).
    jz ssse3              ; Версия SSE 3 доступна.
    mov r12, 1
    xor rax, rax
    mov rdi, fmt_sse3
    push rcx              ; Изменяется функцией printf.
```

```

    call printf
    pop rcx
ssse3:
    test ecx,9h                ; Проверка бита 0 (SSE 3).
    jz sse41                  ; Версия SSE 3 доступна.
    mov r12,1
    xor rax,rax
    mov rdi,fmt_ssse3
    push rcx                  ; Изменяется функцией printf.
    call printf
    pop rcx
sse41:
    test ecx,800000h          ; Проверка бита 19 (SSE 4.1).
    jz sse42                  ; Версия SSE 4.1 доступна.
    mov r12,1
    xor rax,rax
    mov rdi,fmt_sse41
    push rcx                  ; Изменяется функцией printf.
    call printf
    pop rcx
sse42:
    test ecx,1000000h         ; Проверка бита 20 (SSE 4.2).
    jz wrapup                 ; Версия SSE 4.2 доступна.
    mov r12,1
    xor rax,rax
    mov rdi,fmt_sse42
    push rcx                  ; Изменяется функцией printf.
    call printf
    pop rcx
wrapup:
    cmp r12,1
    je sse_ok
    mov rdi,fmt_no_sse
    xor rax,rax
    call printf                ; Вывод сообщения, если SSE-инструкции недоступны.
    jmp the_exit
sse_ok:
    mov rax,r12                ; Возвращает 1, sse-инструкции поддерживаются.
the_exit:
leave
ret

```

Основная программа `main` вызывает только одну функцию `cru_sse`, и если возвращается значение 1, то процессор поддерживает некоторую версию SSE. Если возвращается значение 0, то можно забыть об использовании SSE-инструкций на этом компьютере. В функции `cru_sse` определяется, какие версии SSE поддерживаются. Если записать 1 в регистр `eax` и выполнить инструкцию `cruid`, как описано выше, то результаты будут возвращены в регистрах `ecx` и `edx`.

ИСПОЛЬЗОВАНИЕ ИНСТРУКЦИИ TEST

Регистры `ecx` и `edx` проверяются инструкцией `test`, которая представляет собой битовую логическую операцию `and`, выполняемую с двумя операндами. Можно

было бы воспользоваться инструкцией `cmp`, но `test` обеспечивает более высокую производительность. Разумеется, также можно было бы применить инструкцию `bt` (см. главу 17).

Инструкция `test` устанавливает флаги `SF`, `ZF` и `PF` в соответствии с результатами проверки. В руководстве Intel можно найти описание работы инструкции `test`, которое приведено ниже.

```
TEMP ← SRC1 AND SRC2;
SF ← MSB(TEMP);
IF TEMP = 0
    THEN ZF ← 1;
    ELSE ZF ← 0;
FI:
PF ← BitwiseXNOR(TEMP[0:7]);
CF ← 0;
OF ← 0;
(* AF is undefined *)
```

В рассматриваемом здесь примере самым важным флагом является `ZF`. Если `ZF=0`, то результат ненулевой, т. е. бит `SSE` равен 1, значит, процессор поддерживает эту версию `SSE`. Инструкция `jz` проверяет условие `ZF=1`, и если оно выполнено, то данная версия `SSE` не поддерживается, и происходит переход к следующей части. В противном случае программа выводит подтверждающее сообщение.

В этом примере после выполнения инструкции `cuid` проверяется регистр `edx`. В регистре `edx` 32 бита, но нам необходимо узнать, установлен ли бит 25, означающий, что процессор поддерживает `SSE` (версию 1). Поэтому в инструкции `test` требуется второй операнд, в бите 25 которого установлена 1, а все остальные биты равны 0. Напомню, что самый младший бит имеет индекс 0, а индекс самого старшего бита 31. В двоичном представлении это выглядит так:

```
0000 0010 0000 0000 0000 0000 0000 0000
```

В шестнадцатеричном формате это число имеет следующий вид:

```
20000000
```

Поток выполнения «спускается каскадом» по программе, и если не обнаружена поддержка `SSE`, то в регистре `r12` остается значение 0. Возвращаемое значение здесь не использовалось, но можно было бы проверить его в регистре `rax`, чтобы сделать вывод о поддержке какой-либо версии `SSE`. Или можно изменить программу так, чтобы она возвращала номер самой последней поддерживаемой версии `SSE`.

На рис. 25.1 показан вывод этой программы.

```
jo@UbuntuDesktop:~/Desktop/linux64/gcc/32 cpu_sse$ make
nasm -f elf64 -g -F dwarf cpu_sse.asm -l cpu_sse.lst
gcc -o cpu_sse cpu_sse.o
jo@UbuntuDesktop:~/Desktop/linux64/gcc/32 cpu_sse$ ./cpu_sse
This cpu supports SSE.
This cpu supports SSE 2.
This cpu supports SSE 3.
This cpu supports SSSE 3.
This cpu supports SSE 4.1.
This cpu supports SSE 4.2.
jo@UbuntuDesktop:~/Desktop/linux64/gcc/32 cpu_sse$ █
```

Рис. 25.1. Вывод программы cpu_sse.asm

Можно также написать аналогичную функцию для извлечения другой информации о характеристиках процессора и в зависимости от возвращаемого результата выбирать конкретные функциональные возможности для данного процессора или другие функциональные возможности для другого процессора.

В одной из следующих глав, где рассматривается AVX, также необходимо будет определить, поддерживает ли процессор AVX.

РЕЗЮМЕ

В этой главе вы узнали, как:

- определить, какие функциональные возможности поддерживаются конкретным процессором, с помощью инструкции `cruid`;
- использовать биты при выполнении инструкции `test`.

Глава 26

SIMD

SIMD – это аббревиатура, соответствующая выражению **Single Instruction Stream, Multiple Data** (один поток выполнения инструкций, много потоков данных). Термин SIMD предложил Майкл Дж. Флинн (Michael J. Flynn) для обозначения функциональности, позволяющей выполнять одну инструкцию для нескольких «потоков» данных. Теоретически SIMD может улучшить производительность программ. SIMD – это одна из форм параллельных вычислений, но в некоторых случаях выполнение одной инструкции для различных потоков данных может происходить последовательно в зависимости от функциональных возможностей аппаратного оборудования и конкретных выполняемых инструкций. Более подробно о классификации параллельных архитектур Флинна можно узнать здесь:

<https://ieeexplore.ieee.org/document/5009071/>

и

https://en.wikipedia.org/wiki/Flynn's_taxonomy.

Первой реализацией SIMD был набор инструкций MMX, но никто не знает точного смысла аббревиатуры MMX. Это могло означать **Multi Media Extension** (мультимедийное расширение), или **Multiple Math Extension** (расширение множества математических операций), или **Matrix Math Extension** (расширение математических операций с матрицами). Как бы то ни было, набор инструкций MMX был заменен набором инструкций **Streaming SIMD Extension (SSE)**. В дальнейшем SSE был расширен до набора команд **Advanced Vector Extension (AVX)**. В этой главе приводится краткое введение в SSE как первоначальная основа для дальнейшего изучения, а в одной из следующих глав будет приведено введение в AVX.

СКАЛЯРНЫЕ ДАННЫЕ И УПАКОВАННЫЕ ДАННЫЕ

Процессор, поддерживающий функциональность SSE, имеет 16 дополнительных 128-битовых регистров (от `xmm0` до `xmm15`) и управляющий регистр `mxcsr`. Ранее мы уже пользовались `xmm`-регистрами для вычислений с плавающей точкой, но с помощью этих мощных регистров можно сделать гораздо больше. Регистры `xmm` могут содержать скалярные данные (*scalar data*) или упакованные данные (*packed data*).

Под скалярными данными подразумевается только одно значение. Если число 3.141592654 помещается в регистр `xmm0`, то `xmm0` содержит скалярное значение. Но в `xmm0` также можно записать несколько значений, тогда эти значения определяются как упакованные данные. Ниже перечислены возможные варианты сохранения значений в любом `xmm`-регистре:

- два 64-битовых числа с плавающей точкой двойной точности;
- четыре 32-битовых числа с плавающей точкой обычной (одинарной) точности;
- два 64-битовых целых числа (четверные слова);
- четыре 32-битовых целых числа (двойные слова);
- восемь 16-битовых коротких (short) целых чисел (слова);
- шестнадцать 8-битовых байт или символов.

На рис. 26.1 показаны соответствующие схемы вариантов размещения значений в `xmm`-регистре.

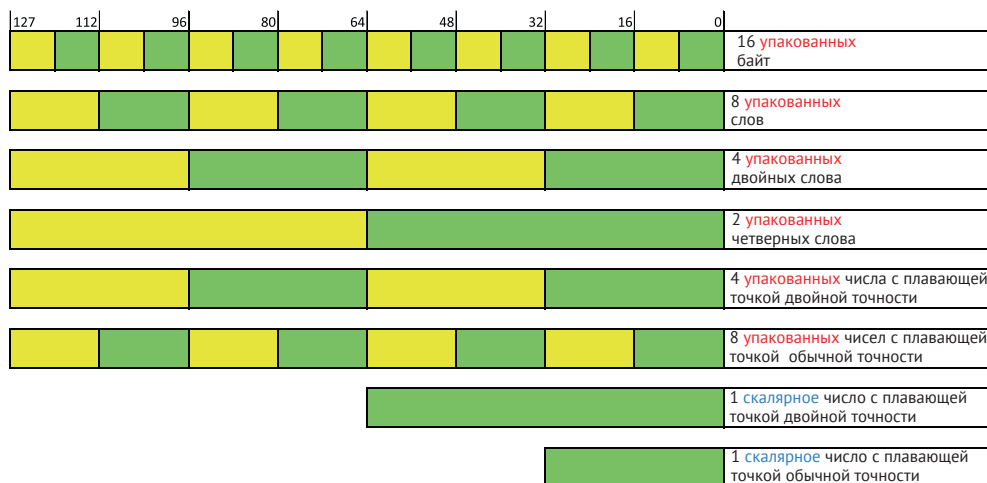


Рис. 26.1. Возможное содержимое любого `xmm`-регистра

Для скалярных значений и для упакованных значений существуют различные ассемблерные инструкции. В руководствах Intel можно найти огромное количество описаний доступных SSE-инструкций. Здесь и в следующих главах рассматривается лишь несколько примеров их применения, чтобы предоставить вам возможность для дальнейшего изучения.

В следующих главах мы будем использовать функциональность AVX. Регистры AVX имеют удвоенный размер по сравнению с регистрами `xmm`. Регистры AVX называются `ymm`-registрами и содержат 256 бит. Существует также расширение AVX-512, предоставляющее регистры AVX-512, содержащие 512 бит, которые называются `zmm`-registрами.

Обладая потенциальными возможностями параллельных вычислений, SIMD-инструкции могут использоваться для повышения скорости вычислений в широком спектре прикладных областей, таких как обработка изображений, звука, сигналов, для векторных и матричных вычислений и т. д. В следующих главах мы рассмотрим применение SIMD для обработки матриц, но не следует волновать-

ся по этому поводу: математическая часть будет ограничена самыми простыми операциями с матрицами. Наша цель – изучение SIMD, а не линейной алгебры.

НЕВЫРОВНЕННЫЕ И ВЫРОВНЕННЫЕ ДАННЫЕ

Данные в памяти могут быть невыровненными или выровненными по определенным адресам, кратным 16, 32 и т. д. Выравнивание данных в памяти может существенно улучшить производительность программы. Причина в следующем: для обработки выровненных упакованных данных SSE-инструкции требуют извлечения 16-байтовых фрагментов памяти за одну операцию – см. схему слева на рис. 26.2. Если данные в памяти не выровнены, то процессору потребуется более одной операции для получения необходимого 16-байтового фрагмента данных, таким образом, выполнение замедляется. Существуют два типа SSE-инструкций: инструкции для выровненных упакованных данных и инструкции для невыровненных упакованных данных. Инструкции для невыровненных упакованных данных могут работать с невыровненными фрагментами памяти, но в общем случае это ухудшает производительность.

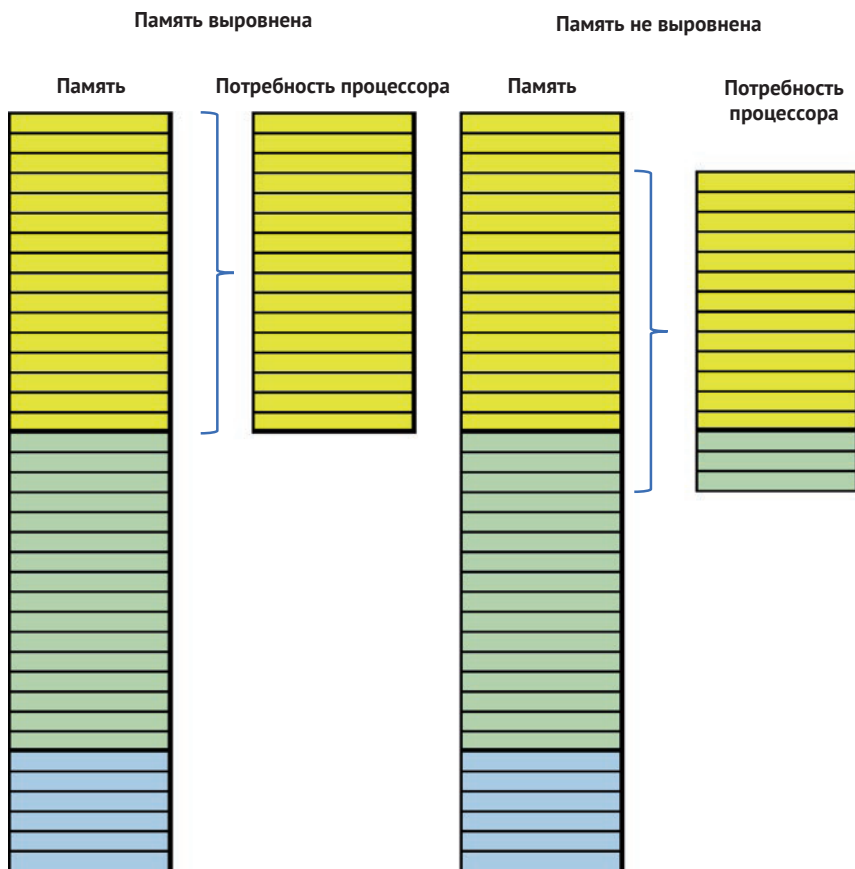


Рис. 26.2. Выравнивание данных

При использовании SSE выравнивание означает, что данные в разделах `section .data` и `section .bss` должны быть выровнены по 16-байтовой границе. В NASM можно воспользоваться директивами ассемблера `align 16` и `alignb 16` перед данными, которые необходимо выравнивать. В следующих главах будут показаны примеры практического применения выравнивания. Для AVX данные должны выравниваться по 32-байтовой границе, а для AVX-512 – по 64-байтовой границе.

РЕЗЮМЕ

В этой главе вы узнали о:

- том, что SSE предоставляет 16 дополнительных 128-битовых регистров;
- различиях между скалярными и упакованными данными;
- важности выравнивания данных.

Глава 27

Наблюдение за регистром MXCSR

Прежде чем начать углубленное изучение программирования с использованием SSE, необходимо понять, как используется регистр управления и состояния SSE для операций с плавающей точкой, который называется `mxcsr`. Это 32-битовый регистр, в котором применяются только младшие 16 бит. В табл. 27.1 описаны используемые биты регистра `mxcsr`.

Таблица 27.1. Биты управления и состояния регистра `mxcsr`

Бит	Мнемоническое обозначение	Описание
0	IE	Ошибка: некорректная (недопустимая) операция
1	DE	Ошибка денормализации
2	ZE	Ошибка: деление на ноль
3	OE	Ошибка переполнения
4	UE	Ошибка: потеря значимости (машинный ноль)
5	PE	Ошибка: потеря точности
6	DAZ	Денормализованные разряды равны нулю
7	IM	Маска некорректной операции
8	DM	Маска операции денормализации
9	ZM	Маска деления на ноль
10	OM	Маска переполнения
11	UM	Маска потери значимости (машинного нуля)
12	PM	Маска точности
13	RC	Управление округлением
14	RC	Управление округлением
15	FZ	Сброс в ноль

Биты с 0 по 5 устанавливаются в 1 при возникновении исключений во время выполнения операций с плавающей точкой, например при делении на ноль или при потере точности (значимых разрядов) при операции с числом с плавающей точкой. Биты с 7 по 12 – это маски, управляющие поведением, когда операция с плавающей точкой устанавливает флаг 1 в битах с 0 по 5. Например, если произошло деление на ноль, программа обычно должна сгенерировать исключение (ошибку) и, возможно, завершиться аварийно. Если флаг маски деления на ноль (ZM) установлен в 1, то программа не завершается, и можно выполнить некоторую инструкцию, чтобы устранить или хотя бы смягчить критический сбой. Все эти маски (биты 7–12) по умолчанию установлены в 1, поэтому исключения для SIMD-операций с плавающей точкой генерироваться не будут. Два бита 13 и 14 управляют округлением, как показано в табл. 27.2.

Таблица 27.2. Биты управления округлением в регистре mxcsr

Биты	Описание
00	Округление до ближайшего числа
01	Округление с недостатком (в меньшую сторону)
10	Округление с избытком (в большую сторону)
11	Усечение (отбрасывание разрядов)

Здесь мы не будем рассматривать в деталях характеристики битов состояния и масок регистра mxcsr, подробное описание см. в руководствах Intel.

РАБОТА С БИТАМИ РЕГИСТРА MXCSR

С битами регистра mxcsr можно работать с помощью инструкций `ldmxcsr` и `stmxcsr`. По умолчанию состояние регистра mxcsr определяется числом `00001F80` или `0001 1111 1000 0000`. Все биты масок установлены в 1, а округление выполняется до ближайшего числа.

В листингах с 27.1 по 27.4 показаны примеры обработки содержимого регистра mxcsr.

Листинг 27.1. Программа *mxcsr.asm*

```
; mxcsr.asm
extern printf
extern print_mxcsr
extern print_hex
section .data
    eleven    dq    11.0
    two       dq    2.0
    three     dq    3.0
    ten       dq    10.0
    zero      dq    0.0
    hex       db    "0x",0
    fmt1      db    10,"Divide, default mxcsr:",10,0
    fmt2      db    10,"Divide by zero, default mxcsr:",10,0
```



```
fmt4      db    10,"Divide, round up:",10,0
fmt5      db    10,"Divide, round down:",10,0
fmt6      db    10,"Divide, truncate:",10,0
f_div     db    "%.1f divided by %.1f is %.16f, in hex: ",0
f_before  db    10,"mxcsr before:",9,0
f_after   db    "mxcsr after:",9,0
```

; Значения регистра mxcsr.

```
default_mxcsr dd 00011111110000000b
round_nearest dd 00011111110000000b
round_down   dd 00111111110000000b
round_up     dd 01011111110000000b
truncate     dd 01111111110000000b
```

section .bss

```
mxcsr_before resd 1
mxcsr_after  resd 1
xmm          resq 1
```

section .text

global main

main:

push rbp

mov rbp, rsp

; Деление.

; Значение mxcsr по умолчанию.

```
mov rdi,fmt1
mov rsi,ten
mov rdx,two
mov ecx,[default_mxcsr]
call apply_mxcsr
```

;-----

; Деление с ошибкой потери точности (значимых разрядов).

; Значение mxcsr по умолчанию.

```
mov rdi,fmt1
mov rsi,ten
mov rdx,three
mov ecx,[default_mxcsr]
call apply_mxcsr
```

; Деление на ноль.

; Значение mxcsr по умолчанию.

```
mov rdi,fmt2
mov rsi,ten
mov rdx,zero
mov ecx,[default_mxcsr]
call apply_mxcsr
```

; Деление с ошибкой потери точности (значимых разрядов).

; Округление в большую сторону.

```
mov rdi,fmt4
mov rsi,ten
mov rdx,three
mov ecx,[round_up]
call apply_mxcsr
```

; Деление с ошибкой потери точности (значимых разрядов).

; Округление в меньшую сторону.

```
mov rdi,fmt5
```

```

    mov    rsi,ten
    mov    rdx,three
    mov    ecx,[round_down]
    call   apply_mxcsr
; Деление с ошибкой потери точности (значимых разрядов).
; Усечение (отбрасывание разрядов).
    mov    rdi,fmt6
    mov    rsi,ten
    mov    rdx,three
    mov    ecx,[truncate]
    call   apply_mxcsr
; -----
; Деление с ошибкой потери точности (значимых разрядов).
; Значение mxcsr по умолчанию.
    mov    rdi,fmt1
    mov    rsi,eleven
    mov    rdx,three
    mov    ecx,[default_mxcsr]
    call   apply_mxcsr
; Деление с ошибкой потери точности (значимых разрядов).
; Округление в большую сторону.
    mov    rdi,fmt4
    mov    rsi,eleven
    mov    rdx,three
    mov    ecx,[round_up]
    call   apply_mxcsr
; Деление с ошибкой потери точности (значимых разрядов).
; Округление в меньшую сторону.
    mov    rdi,fmt5
    mov    rsi,eleven
    mov    rdx,three
    mov    ecx,[round_down]
    call   apply_mxcsr
; Деление с ошибкой потери точности (значимых разрядов).
; Усечение (отбрасывание разрядов).
    mov    rdi,fmt6
    mov    rsi,eleven
    mov    rdx,three
    mov    ecx,[truncate]
    call   apply_mxcsr
leave
ret

; Функция. -----
apply_mxcsr:
push    rbp
mov     rbp,rsi
push    rsi
push    rdx
push    rcx
push    rbp      ; Еще одна операция записи для выравнивания стека.
call    printf
pop     rbp
pop     rcx
pop     rdx
pop     rsi

```

```

    mov     [mxcsr_before],ecx
    ldmxcsr [mxcsr_before]
    movsd   xmm2, [rsi] ; Число с плав. точкой двойной точности в регистр xmm2.
    divsd   xmm2, [rdx] ; Деление xmm2.
    stmxcsr [mxcsr_after] ; Сохранение mxcsr в память.
    movsd   [xmm],xmm2 ; Для использования в функции print_xmm.
    mov     rdi,f_div
    movsd   xmm0, [rsi]
    movsd   xmm1, [rdx]
    call     printf
    call     print_xmm
; Вывод mxcsr.
    mov     rdi,f_before
    call     printf
    mov     rdi, [mxcsr_before]
    call     print_mxcsr
    mov     rdi,f_after
    call     printf
    mov     rdi, [mxcsr_after]
    call     print_mxcsr
leave
ret

; Функция. -----
print_xmm:
push rbp
mov rbp,rsp
mov rdi, hex ; Вывод префикса 0x.
call printf
mov rcx,8
.loop:
xor rdi,rdi
mov dil,[xmm+rcx-1]
push rcx
call print_hex
pop rcx
loop .loop
leave
ret

```

Листинг 27.2. Программа *print_hex.c*

```

// print_hex.c

#include <stdio.h>

void print_hex(unsigned char n){
    if (n < 16) printf("0");
    printf("%x",n);
}

```

Листинг 27.3. Программа *print_mxcsr.c*

```
// print_mxcsr.c

#include <stdio.h>

void print_mxcsr(long long n){
    long long s,c;
    for (c = 15; c >= 0; c--){
        {
            s = n >> c;
            // Вывод пробела после каждого 8-го бита.
            if ((c+1) % 4 == 0) printf(" ");
            if (s & 1)
                printf("1");
            else
                printf("0");
        }
        printf("\n");
    }
}
```

Листинг 27.4. *makefile*

```
mxcsr: mxcsr.o print_mxcsr.o print_hex.o
    gcc -o mxcsr mxcsr.o print_mxcsr.o print_hex.o -no-pie
mxcsr.o: mxcsr.asm
    nasm -f elf64 -g -F dwarf mxcsr.asm -l mxcsr.lst
print_mxcsr: print_mxcsr.c
    gcc -c print_mxcsr.c
print_hex: print_hex.c
    gcc -c print_hex.c
```

В этой программе продемонстрированы различные режимы округления и обработка маскированного деления на ноль. По умолчанию округление выполняется до ближайшего числа. Например, в десятичном формате положительное число, заканчивающееся на .5 и больше, должно округляться до следующего большего числа, а отрицательное число, заканчивающееся на .5 и больше, должно округляться до следующего меньшего (отрицательного) числа. Но в примере программы округление выполняется в шестнадцатеричном формате, а не в десятичном, поэтому не всегда получается тот же результат, что и при округлении десятичных чисел.

На рис. 27.1 показан вывод данной программы.

```

jo@UbuntuDesktop:~/Desktop/linux64/gcc/34 mxcsr$ ./mxcsr

Divide, default mxcsr:
10.0 divided by 2.0 is 5.000000000000000, in hex: 0x4014000000000000
mxcsr before:  0001 1111 1000 0000
mxcsr after:   0001 1111 1000 0000

Divide, default mxcsr:
10.0 divided by 3.0 is 3.333333333333335, in hex: 0x400aaaaaaaaaaaaab
mxcsr before:  0001 1111 1000 0000
mxcsr after:   0001 1111 1010 0000

Divide by zero, default mxcsr:
10.0 divided by 0.0 is inf, in hex: 0x7ff0000000000000
mxcsr before:  0001 1111 1000 0000
mxcsr after:   0001 1111 1000 0100

Divide, round up:
10.0 divided by 3.0 is 3.333333333333335, in hex: 0x400aaaaaaaaaaaaab
mxcsr before:  0101 1111 1000 0000
mxcsr after:   0101 1111 1010 0000

Divide, round down:
10.0 divided by 3.0 is 3.333333333333330, in hex: 0x400aaaaaaaaaaaaaa
mxcsr before:  0011 1111 1000 0000
mxcsr after:   0011 1111 1010 0000

Divide, truncate:
10.0 divided by 3.0 is 3.333333333333330, in hex: 0x400aaaaaaaaaaaaaa
mxcsr before:  0111 1111 1000 0000
mxcsr after:   0111 1111 1010 0000

Divide, default mxcsr:
11.0 divided by 3.0 is 3.666666666666665, in hex: 0x400d555555555555
mxcsr before:  0001 1111 1000 0000
mxcsr after:   0001 1111 1010 0000

Divide, round up:
11.0 divided by 3.0 is 3.6666666666666670, in hex: 0x400d555555555556
mxcsr before:  0101 1111 1000 0000
mxcsr after:   0101 1111 1010 0000

Divide, round down:
11.0 divided by 3.0 is 3.666666666666665, in hex: 0x400d555555555555
mxcsr before:  0011 1111 1000 0000
mxcsr after:   0011 1111 1010 0000

Divide, truncate:
11.0 divided by 3.0 is 3.666666666666665, in hex: 0x400d555555555555
mxcsr before:  0111 1111 1000 0000
mxcsr after:   0111 1111 1010 0000
jo@UbuntuDesktop:~/Desktop/linux64/gcc/34 mxcsr$

```

Рис. 27.1. Вывод программы *mxcsr.asm*

АНАЛИЗ ПРОГРАММЫ

Проанализируем программу *mxcsr.asm*. В ней выполнено несколько операций деления с применением округления. Операции деления производятся в функции `apply_mxcsr`. Перед вызовом этой функции адрес строки заголовка записы-

вается в регистр `rdi`, делимое – в `rsi`, делитель – в `rdx`. Затем требуемое значение регистра `rax` копируется из памяти в регистр `ecx` – при первом вызове функции это значение `rax` по умолчанию. После этого вызывается функция `apply_rax`. В этой функции выводится строка заголовка, но сначала не забываем сохранить необходимые регистры и выровнять стек. Далее значение регистра `ecx` сохраняется в переменной `rax_before`, и это сохраненное значение загружается в регистр `rax` с помощью инструкции `ld_rax`. Инструкция `ld_rax` принимает 32-битовое значение переменной памяти (двойное слово) как операнд. Инструкция `divsd` принимает `rax`-регистр как первый операнд и `rax`-регистр или 64-битовую переменную как второй операнд. После выполнения операции деления содержимое регистра `rax` сохраняется в памяти в переменной `rax_after` с помощью инструкции `st_rax`. Также копируется частное из регистра `rax2` в переменную памяти `rax`, чтобы вывести полученное значение.

Сначала выводится частное в десятичном формате, затем нужно вывести его в шестнадцатеричном формате в той же строке. При использовании функции `printf` в программе на ассемблере невозможно вывести шестнадцатеричное число (по крайней мере, в той версии, которая используется здесь), поэтому необходимо написать функцию, которая выводит числа в шестнадцатеричном формате, – это функция `print_hex`. Функция принимает значение переменной памяти `rax` и поочередно загружает байты по одному в `rdi` в цикле. В том же цикле специально написанная для этого примера C-функция `print_hex` вызывается для каждого байта. Используя в адресе уменьшающийся счетчик цикла `rsi`, мы также позаботились о сохранении правильного порядка байтов – от младшего к старшему: значение с плавающей точкой хранится в памяти именно с таким порядком байтов.

Далее выводятся значения `rax_before` и `rax_after`, чтобы можно было сравнить их. Функция `print_rax` применяется для вывода битов регистра `rax` – она похожа на функции вывода битов, которые мы использовали в предыдущих главах.

Для некоторых читателей эта программа может показаться слишком сложной. В этом случае пройдите по программе с помощью отладчика и наблюдайте за памятью и регистрами.

Проанализируем вывод: можно видеть, что регистр `rax` не изменяется при делении 10 на 2. При делении 10 на 3 получается 3.333. Здесь `rax` сообщает об ошибке потери точности (значимых разрядов) в бите 5. По умолчанию округление выполняется к ближайшему числу, поэтому последняя шестнадцатеричная цифра увеличивается с `a` на `b`. В десятичном формате округление должно выполняться в меньшую сторону, но в шестнадцатеричном представлении цифра `a` больше 8, поэтому необходимо округление в большую сторону к `b`.

Далее выполняется деление на ноль: `rax` сообщает об этом установкой бита 2, но программа не завершается из-за критического сбоя, поскольку установлена маска деления на ноль `ZM`. Результат определен как `inf` (бесконечность) или `0x7ff0000000000000`.

При выполнении следующей операции деления и округлении в большую сторону получается тот же результат, что и при округлении к ближайшему числу. Следующие две операции деления с округлением в меньшую сторону и усечением дают в результате число с последней шестнадцатеричной цифрой `a`.

Чтобы продемонстрировать различия в методах округления, выполняются те же операции деления 11 на 3. В этом варианте деления получается частное с меньшей конечной шестнадцатеричной цифрой. Вы можете сами сравнить поведение при округлении.

Предлагается дополнительное упражнение: очистить (сбросить в 0) маску деления на ноль ZM, пересобрать и запустить программу. Программа завершится аварийно из-за критического сбоя. Маска деления на ноль и другие маски позволяют перехватывать ошибки и переходить к некоторой процедуре коррекции ошибок.

РЕЗЮМЕ

В этой главе вы узнали о:

- предназначении регистра `mxcsr` и его отдельных битов;
- работе с регистром `mxcsr`;
- подробностях различных методов округления чисел.

Глава 28

Выравнивание для SSE

Пора начинать настоящую работу с SSE. Несмотря на то что в нескольких главах мы уже имели дело с SSE, до сих пор эта тема обсуждалась лишь поверхностно. Существует несколько сотен SIMD-инструкций (MMX, SSE, AVX), и для их подробного описания потребуется отдельная книга или даже несколько книг. В этой главе рассматривается несколько примеров, для того чтобы вы поняли, с чего следует начать. Цель данных примеров – позволить читателю определить собственную методику изучения многочисленных SIMD-инструкций по руководствам Intel. В этой главе рассматривается методика выравнивания, которая кратко описывалась в главе 26.

ПРИМЕР БЕЗ ВЫРАВНИВАНИЯ

В листинге 28.1 показана операция сложения векторов с использованием данных, которые не выровнены в памяти.

Листинг 28.1. Программа *sse_unaligned.asm*

```
; sse_unaligned.asm
extern printf
section .data
; С обычной точностью.
    spvector1 dd 1.1
               dd 2.2
               dd 3.3
               dd 4.4
    spvector2 dd 1.1
               dd 2.2
               dd 3.3
               dd 4.4

; С двойной точностью.
    dpvector1 dq 1.1
               dq 2.2
    dpvector2 dq 3.3
               dq 4.4

    fmt1 db "Single Precision Vector 1: %f, %f, %f, %f",10,0
    fmt2 db "Single Precision Vector 2: %f, %f, %f, %f",10,0
    fmt3 db "Sum of Single Precision Vector 1 and Vector 2:"
          db " %f, %f, %f, %f",10,0
```



```
fmt4 db "Double Precision Vector 1: %f, %f",10,0
fmt5 db "Double Precision Vector 2: %f, %f",10,0
fmt6 db "Sum of Double Precision Vector 1 and Vector 2:"
      db " %f, %f",10,0
```

```
section .bss
    spvector_res resd 4
    dpvector_res resq 4
```

```
section .text
    global main
```

```
main:
    push    rbp
    mov     rbp, rsp
```

; Сложение 2 векторов из чисел с плав. точкой с обычной точностью.

```
    mov     rsi, spvector1
    mov     rdi, fmt1
    call    printspf
    mov     rsi, spvector2
    mov     rdi, fmt2
    call    printspf
    movups   xmm0, [spvector1]
    movups   xmm1, [spvector2]
    addps    xmm0, xmm1
    movups   [spvector_res], xmm0
    mov     rsi, spvector_res
    mov     rdi, fmt3
    call    printspf
```

; Сложение 2 векторов из чисел с плав. точкой с двойной точностью.

```
    mov     rsi, dpvector1
    mov     rdi, fmt4
    call    printdcpf
    mov     rsi, dpvector2
    mov     rdi, fmt5
    call    printdcpf
    movupd   xmm0, [dpvector1]
    movupd   xmm1, [dpvector2]
    addpd    xmm0, xmm1
    movupd   [dpvector_res], xmm0
    mov     rsi, dpvector_res
    mov     rdi, fmt6
    call    printdcpf
```

```
leave
ret
```

```
printspf:
    push    rbp
    mov     rbp, rsp
    movss    xmm0, [rsi]
    cvtss2sd xmm0, xmm0
    movss    xmm1, [rsi+4]
    cvtss2sd xmm1, xmm1
    movss    xmm2, [rsi+8]
    cvtss2sd xmm2, xmm2
    movss    xmm3, [rsi+12]
```

```

        cvtss2sd    xmm3,xmm3
        mov         rax,4             ; Четыре числа с плав.точкой.
        call        printf
leave
ret

printdpfp:
push    rbp
mov     rbp,rsb
movsd   xmm0,[rsi]
movsd   xmm1,[rsi+8]
mov     rax,2             ; Два числа с плав.точкой.
call    printf
leave
ret

```

Первая SSE-инструкция `movups` (означающая «move unaligned packed single precision» – перемещение невыровненных упакованных данных с обычной точностью) копирует данные из памяти в регистры `xmm0` и `xmm1`. В результате регистр `xmm0` содержит один вектор с четырьмя значениями с обычной точностью, а регистр `xmm1` содержит второй вектор с четырьмя значениями с обычной точностью. Потом применяется инструкция `addps` (означающая «add packed single precision» – сложение упакованных данных с обычной точностью) для сложения этих векторов, а полученный в результате вектор записывается в регистр `xmm0`, затем передается в память. Далее результат выводится с помощью функции `printf`. В этой функции копируется каждое значение из памяти в `xmm`-регистры с помощью инструкции `movss` (означающей «move scalar single precision» – перемещение скалярного значения с обычной точностью). Поскольку функция `printf` ожидает передачу аргументов с плавающей точкой двойной точности, выполняется преобразование чисел с плавающей точкой обычной точности в числа двойной точности с помощью инструкции `cvtss2sd` (означающей «convert scalar single to scalar double» – преобразование скалярного значения обычной точности в скалярное значение двойной точности).

Далее выполняется сложение двух значений двойной точности. Эта операция аналогична операции сложения чисел обычной точности, но для значений двойной точности используются инструкции `movupd` и `addpd`. Функция `printdpfp` для вывода чисел двойной точности немного проще. Мы работаем с векторами, состоящими только из двух элементов, поэтому сразу же используются значения двойной точности, и нет необходимости в преобразовании элементов векторов.

На рис. 28.1 показан вывод этой программы.

```

jo@UbuntuDesktop:~/Desktop/linux64/gcc/33 sse_unaligned$ make
nasm -f elf64 -g -F dwarf sse_unaligned.asm -l sse_unaligned.lst
gcc -o sse_unaligned sse_unaligned.o
jo@UbuntuDesktop:~/Desktop/linux64/gcc/33 sse_unaligned$ ./sse_unaligned
Single Precision Vector 1: 1.100000, 2.200000, 3.300000, 4.400000
Single Precision Vector 2: 1.100000, 2.200000, 3.300000, 4.400000
Sum of Single Precision Vector 1 and Vector 2: 2.200000, 4.400000, 6.600000, 8.800000
Double Precision Vector 1: 1.100000, 2.200000
Double Precision Vector 2: 3.300000, 4.400000
Sum of Double Precision Vector 1 and Vector 2: 4.400000, 6.600000
jo@UbuntuDesktop:~/Desktop/linux64/gcc/33 sse_unaligned$ █

```

Рис. 28.1. Вывод программы `sse_unaligned.asm`

ПРИМЕР С ВЫРАВНИВАНИЕМ

В листинге 28.2 показана операция сложения векторов.

Листинг 28.2. Программа `sse_aligned.asm`

```

; sse_aligned.asm
extern printf
section .data
    dummy db 13
align 16
    spvector1 dd 1.1
               dd 2.2
               dd 3.3
               dd 4.4
    spvector2 dd 1.1
               dd 2.2
               dd 3.3
               dd 4.4

    dpvector1 dq 1.1
               dq 2.2
    dpvector2 dq 3.3
               dq 4.4

    fmt1 db "Single Precision Vector 1: %f, %f, %f, %f",10,0
    fmt2 db "Single Precision Vector 2: %f, %f, %f, %f",10,0
    fmt3 db "Sum of Single Precision Vector 1 and Vector 2:"
          db " %f, %f, %f, %f",10,0
    fmt4 db "Double Precision Vector 1: %f, %f",10,0
    fmt5 db "Double Precision Vector 2: %f, %f",10,0
    fmt6 db "Sum of Double Precision Vector 1 and Vector 2:"
          db " %f, %f",10,0

section .bss
alignb 16
    spvector_res resd 4
    dpvector_res resq 4
section .text
    global main
main:
    push rbp
    mov rbp, rsp

```

; Сложение 2 векторов из чисел с плав.точкой с обычной точностью.

```

mov    rsi,spvector1
mov    rdi,fmt1
call   printspfp

mov    rsi,spvector2
mov    rdi,fmt2
call   printspfp

movaps xmm0, [spvector1]
addps  xmm0, [spvector2]

movaps [spvector_res], xmm0
mov    rsi,spvector_res
mov    rdi,fmt3
call   printspfp

```

; Сложение 2 векторов из чисел с плав.точкой с двойной точностью.

```

mov    rsi,dpvector1
mov    rdi,fmt4
call   printdpfp

mov    rsi,dpvector2
mov    rdi,fmt5
call   printdpfp

movapd xmm0, [dpvector1]
addpd  xmm0, [dpvector2]

movapd [dpvector_res], xmm0
mov    rsi,dpvector_res
mov    rdi,fmt6
call   printdpfp

```

; Выход.

```

mov    rsp,rbp
pop    rbp    ; Восстановление результата команды push, выполненной в начале.
ret

```

```

printspfp:
push   rbp
mov    rbp,rsp
movss  xmm0, [rsi]
cvtss2sd xmm0,xmm0    ; printf ожидает аргумент двойной точности.
movss  xmm1, [rsi+4]
cvtss2sd xmm1,xmm1
movss  xmm2, [rsi+8]
cvtss2sd xmm2,xmm2
movss  xmm3, [rsi+12]
cvtss2sd xmm3,xmm3
mov    rax,4           ; Четыре числа с плав.точкой.
call   printf
leave
ret

```

```

printdpfp:
push   rbp

```

```

mov    rbp, rsp
movsd  xmm0, [rsi]
movsd  xmm1, [rsi+8]
mov    rax, 2          ; Два числа с плав. точкой.
call   printf
leave
ret

```

В начале этой программы создается переменная `dummy` для полной уверенности в том, что память не выровнена по 16-байтовой границе. Затем используется директива ассемблера NASM `align 16` в разделе `section .data` и директива `alignb 16` в разделе `section .bss`. Необходимо добавлять эти директивы ассемблера перед каждым блоком данных, который требует выравнивания.

SSE-инструкции немного отличаются от инструкций из версии без выравнивания. Инструкция `movaps` (означающая «move aligned packed single precision» – перемещение выровненных упакованных данных обычной точности) копирует данные из памяти в регистр `xmm0`. После этого сразу можно выполнять сложение упакованных чисел в памяти с упакованными значениями в регистре `xmm0`. В этом состоит отличие от версии без выравнивания, где требовалась предварительная запись двух значений в `xmm`-регистры. Если добавить переменную `dummy` в версию без выравнивания и попытаться использовать инструкцию `movaps` вместо `movups` с переменной памяти как вторым операндом, то возникает опасность нарушения сегментации памяти (*segmentation fault*) во время выполнения. Попробуйте проверить это самостоятельно.

Регистр `xmm0` содержит полученный в результате сложения вектор с четырьмя значениями обычной точности. Затем выводится этот результат с помощью функции `printspfpr`. В этой функции каждое значение из памяти копируется в `xmm`-регистры. Поскольку функция `printf` ожидает передачу аргументов с плавающей точкой двойной точности, выполняется преобразование чисел с плавающей точкой обычной точности в числа двойной точности с помощью инструкции `cvtss2sd` (означающей «convert scalar single to scalar double» – преобразование скалярного значения обычной точности в скалярное значение двойной точности).

Далее выполняется сложение двух значений двойной точности. Эта операция аналогична операции сложения чисел обычной точности, но для значений двойной точности используются инструкции `movapd` и `addpd`.

На рис. 28.2 показан вывод примера программы с выравниванием.

```

jo@UbuntuDesktop:~/Desktop/linux64/gcc/34 sse_aligned$ make
nasm -f elf64 -g -F dwarf sse_aligned.asm -l sse_aligned.lst
gcc -o sse_aligned sse_aligned.o
jo@UbuntuDesktop:~/Desktop/linux64/gcc/34 sse_aligned$ ./sse_aligned
Single Precision Vector 1: 1.100000, 2.200000, 3.300000, 4.400000
Single Precision Vector 2: 1.100000, 2.200000, 3.300000, 4.400000
Sum of Single Precision Vector 1 and Vector 2: 2.200000, 4.400000, 6.600000, 8.800000
Double Precision Vector 1: 1.100000, 2.200000
Double Precision Vector 2: 3.300000, 4.400000
Sum of Double Precision Vector 1 and Vector 2: 4.400000, 6.600000
jo@UbuntuDesktop:~/Desktop/linux64/gcc/34 sse_aligned$

```

Рис. 28.2. Вывод программы `sse_aligned.asm`

На рис. 28.3 представлен результат выполнения примера программы без выравнивания с переменной `dummy`, добавленной как второй операнд инструкции `movaps`.

```
jo@UbuntuDesktop:~/Desktop/linux64/gcc/34 sse_unaligned$ make
nasm -f elf64 -g -F dwarf sse_unaligned.asm -l sse_unaligned.lst
gcc -o sse_unaligned sse_unaligned.o -no-pie
jo@UbuntuDesktop:~/Desktop/linux64/gcc/34 sse_unaligned$ ./sse_unaligned
Single Precision Vector 1: 1.100000, 2.200000, 3.300000, 4.400000
Single Precision Vector 2: 1.100000, 2.200000, 3.300000, 4.400000
Segmentation fault (core dumped)
jo@UbuntuDesktop:~/Desktop/linux64/gcc/34 sse_unaligned$ █
```

Рис. 28.3. Критический сбой при нарушении сегментации памяти в программе *sse_unaligned.asm*

РЕЗЮМЕ

В этой главе вы узнали о:

- скалярных и упакованных данных;
- выровненных и невыровненных данных;
- том, как выравнивать данные (в памяти);
- перемещении (копировании) данных и арифметических инструкциях для упакованных данных;
- методах преобразования данных обычной точности в данные двойной точности.

Глава 29

SSE-инструкции для работы с упакованными целыми числами

В предыдущей главе использовались значения с плавающей точкой и инструкции для их обработки. SSE также предоставляет длинный список инструкций для работы с целыми числами, и точно так же, как в предыдущей главе, здесь рассматривается применение лишь нескольких инструкций, чтобы предоставить читателям основу для дальнейшего обучения.

SSE-ИНСТРУКЦИИ ДЛЯ РАБОТЫ С ЦЕЛЫМИ ЧИСЛАМИ

В листинге 29.1 показан пример программы.

Листинг 29.1. Программа *sse_integer.asm*

```
; sse_integer.asm
extern printf
section .data
    dummy    db    13
align 16
    pdivector1 dd    1
                dd    2
                dd    3
                dd    4
    pdivector2 dd    5
                dd    6
                dd    7
                dd    8
    fmt1 db "Packed Integer Vector 1: %d, %d, %d, %d",10,0
    fmt2 db "Packed Integer Vector 2: %d, %d, %d, %d",10,0
    fmt3 db "Sum Vector: %d, %d, %d, %d",10,0
    fmt4 db "Reverse of Sum Vector: %d, %d, %d, %d",10,0
section .bss
alignb 16
    pdivector_res    resd 4
    pdivector_other  resd 4
```

```

section .text
    global main
main:
    push    rbp
    mov     rbp, rsp
    ; Вывод вектора 1.
    mov     rsi, pdivector1
    mov     rdi, fmt1
    call    printpdi
    ; Вывод вектора 2.
    mov     rsi, pdivector2
    mov     rdi, fmt2
    call    printpdi
    ; Сложение 2 векторов с выровненными целыми числами двойной точности.
    movdqa  xmm0, [pdivector1]
    paddb   xmm0, [pdivector2]
    ; Сохранение результата в памяти.
    movdqa  [pdivector_res], xmm0
    ; Вывод вектора, хранящегося в памяти.
    mov     rsi, pdivector_res
    mov     rdi, fmt3
    call    printpdi
    ; Копирование вектора из памяти в регистр xmm3.
    movdqa  xmm3, [pdivector_res]
    ; Извлечение упакованных значений из xmm3.
    pextrd  eax, xmm3, 0
    pextrd  ebx, xmm3, 1
    pextrd  ecx, xmm3, 2
    pextrd  edx, xmm3, 3
    ; Вставка значений в регистр xmm0 в обратном порядке.
    pinsrd  xmm0, eax, 3
    pinsrd  xmm0, ebx, 2
    pinsrd  xmm0, ecx, 1
    pinsrd  xmm0, edx, 0
    ; Вывод реверсированного вектора.
    movdqa  [pdivector_other], xmm0
    mov     rsi, pdivector_other
    mov     rdi, fmt4
    call    printpdi
    ; Выход.
    mov     rsp, rbp
    pop     rbp
    ret

    ; Функция вывода. -----
printpdi:
    push    rbp
    mov     rbp, rsp
    movdqa  xmm0, [rsi]
    ; Извлечение упакованных значений из регистра xmm0.
    pextrd  esi, xmm0, 0
    pextrd  edx, xmm0, 1
    pextrd  ecx, xmm0, 2
    pextrd  r8d, xmm0, 3
    mov     rax, 0
    call    printf
    ; Числа с плав. точкой не используются.
leave
ret

```


Анализ исходного кода

В рассматриваемом здесь примере снова обрабатываются два вектора, на сей раз содержащие целочисленные значения. Для копирования значений в `xmm`-регистр применяется инструкция `movdqa`. Эта инструкция предназначена для работы с выровненными данными. Затем инструкция `padd` суммирует значения в регистрах и помещает результат в `xmm0`. Для использования функции `printf` необходимо извлечь целочисленные значения из `xmm`-регистров и записать их в «обычные» регистры. Следует помнить о соглашениях о вызовах функций, в соответствии с которыми функция `printf` считает, что `xmm`-регистр предназначен для значений с плавающей точкой. Если предварительно не извлечь из такого регистра целочисленные значения, то `printf` будет рассматривать содержимое `xmm`-регистра как значения с плавающей точкой и выводить неправильные результаты. Для вставки и извлечения упакованных целых чисел используются инструкции `pinsrd` и `pextrd` соответственно. Полученный в результате вектор еще и реверсируется, чтобы продемонстрировать, как вставляются значения в упакованный вектор в `xmm`-регистре.

Существуют также версии инструкций `movd`, `padd`, `pinsr` и `pextr` для байтов, слов, двойных и четверных слов.

На рис. 29.1 показан вывод этой программы.

```
jo@UbuntuDesktop:~/Desktop/linux64/gcc/35 sse_integer$ make
nasm -f elf64 -g -F dwarf sse_integer.asm -l sse_integer.lst
gcc -o sse_integer sse_integer.o
jo@UbuntuDesktop:~/Desktop/linux64/gcc/35 sse_integer$ ./sse_integer
Packed Integer Vector 1: 1, 2, 3, 4
Packed Integer Vector 2: 5, 6, 7, 8
Sum of Packed Integer Vector 1 and Vector 2: 6, 8, 10, 12
Reverse of Sum Vector: 12, 10, 8, 6
jo@UbuntuDesktop:~/Desktop/linux64/gcc/35 sse_integer$
```

Рис. 29.1. Вывод программы `sse_integer.asm`

РЕЗЮМЕ

В этой главе вы узнали об:

- упакованных целочисленных данных;
- инструкциях для вставки и извлечения упакованных целочисленных данных;
- инструкциях для копирования и сложения упакованных целочисленных данных.

Глава 30

Обработка строк средствами SSE

В SSE версии 4.2 были введены четыре инструкции сравнения строк: две инструкции для строк с длиной, не определенной явно, и две инструкции для строк с явно заданной длиной. Две из этих четырех инструкций используют маски.

Строки с длиной, не определенной явно, – это строки с завершающим 0. Для строк с явно заданной длиной размер должен определяться некоторыми другими средствами.

В этой главе немного внимания уделено строкам SSE, потому что инструкции сравнения для них несколько более сложны и необычны, особенно при использовании масок. В табл. 30.1 перечислены эти инструкции.

Таблица 30.1. Инструкции сравнения строк SSE

Строка	Инструкция	Аргумент1 (arg1)	Аргумент2 (arg2)	Аргумент3 (arg3)	Вывод
Неявная длина	pcmpistri	xmm	xmm/m128	imm8	Индекс в esx
Неявная длина	pcmpistrm	xmm	xmm/m128	imm8	Маска в xmm0
Явная длина	pcmpestri	xmm	xmm/m128	imm8	Индекс в esx
Явная длина	pcmpestrm	xmm	xmm/m128	imm8	Маска в xmm0

Смысловое значение этих инструкций:

pcmpistri: **p**acked **c**ompare **i**mplicit length **s**trings, **r**eturn **i**ndex – сравнение упакованных строк с неявно заданной длиной, возвращает индекс;

pcmpistrm: **p**acked **c**ompare **i**mplicit length **s**trings, **r**eturn **m**ask – сравнение упакованных строк с неявно заданной длиной, возвращает маску;

pcmpestri: **p**acked **c**ompare **e**xplicit length **s**trings, **r**eturn **i**ndex – сравнение упакованных строк с явно заданной длиной, возвращает индекс;

pcmpestrm: **p**acked **c**ompare **e**xplicit length **s**trings, **r**eturn **m**ask – сравнение упакованных строк с явно заданной длиной, возвращает маску.

Эти инструкции сравнения принимают три аргумента. Аргумент 1 всегда находится в `xmm`-регистре, аргумент 2 может располагаться в `xmm`-регистре или в некоторой локации памяти, аргумент 3 является «непосредственным» (`immediate`) и представлен управляющим байтом (см. `imm8` в руководствах Intel), который определяет, как выполняется инструкция. Управляющий байт играет важную роль, поэтому в следующем разделе он будет описан более подробно.

УПРАВЛЯЮЩИЙ БАЙТ `IMM8`

В табл. 30.2 приведено описание отдельных битов управляющего байта.

Таблица 30.2. Управляющий байт `imm8`

Параметры	Номер бита	Значение бита	Операция	Описание
Формат вывода	7	0	Зарезервирована	Зарезервирован
	6	0	Битовая маска	<code>xmm0</code> содержит <code>IntRes2</code> как битовую маску
		1	Байтовая маска	<code>xmm0</code> содержит <code>IntRes2</code> как байтовую маску
		0	Самый младший индекс	Самый младший индекс, найденный в <code>ecx</code>
Знаковые разряды (ориентация)	5, 4	1	Самый старший индекс	Самый старший индекс, найденный в <code>ecx</code>
		00	+	<code>IntRes2</code> = <code>IntRes1</code>
		01	–	<code>IntRes2</code> = <code>~IntRes1</code>
		10	Маскированный +	<code>IntRes2</code> = <code>IntRes1</code>
Агрегация и сравнение	3, 2	11	Маскированный –	<code>IntRes2</code> = <code>~IntRes1</code>
		00	Равенство любому символу из набора	Совпадение символов
		01	Равенство символу из диапазона	Соответствие символам из заданного диапазона
		10	Равенство в любой позиции	Сравнение строк
Формат данных	1, 0	11	Равенство в определенном порядке	Поиск подстроки
		00	Упакованные беззнаковые байты	
		01	Упакованные беззнаковые слова	
		10	Упакованные знаковые байты	
		11	Упакованные знаковые слова	

Описанные выше инструкции сравнения принимают входные данные (их формат определяется в битах 1 и 0), выполняют операции агрегации и сравнения (биты 3 и 2), которые дают промежуточный результат (совпадение между аргументами `arg1` и `arg2`). В руководствах Intel этот результат обозначен как `IntRes1`. Знаковые разряды (биты 5 и 4) применяются к промежуточному результату `IntRes1` для получения `IntRes2`. Затем `IntRes2` используется для вывода итогового результата в требуемом формате. Отрицание (\sim `IntRes1`) означает вычисление дополнительного кода значения `IntRes1` и запись полученного результата в `IntRes2`. То есть каждый бит, равный 1, преобразуется в 0, а каждый бит, равный 0, преобразуется в 1. Другими словами, это операция логического отрицания (logical NOT). Результат в `IntRes2` может быть сохранен как маска в регистре `xmm0` для инструкций с использованием маски `rcmpistrm` и `rcmpestrm` или как индекс в регистре `ecx` для инструкций `rcmpistri` и `rcmpestri`. Для лучшего понимания будет полезным рассмотрение нескольких примеров.

Ниже приведены примеры содержимого управляющего байта:

00001000 от 0x08:

- 00 – упакованные беззнаковые байты,
- 10 – равенство в любой позиции,
- 00 – положительная ориентация,
- 00 – самый младший индекс в регистре `ecx`

01000100 от 0x44:

- 00 – упакованные беззнаковые байты,
- 01 – равенство в диапазоне,
- 00 – положительная ориентация,
- 01 – регистр `xmm0` содержит байтовую маску

ИСПОЛЬЗОВАНИЕ УПРАВЛЯЮЩЕГО БАЙТА IMM8

В этом разделе показано, как можно устанавливать биты в управляющем байте `imm8` для управления поведением инструкциями обработки упакованных строк. Кроме того, приведены примеры, демонстрирующие воздействие различных вариантов установки управляющих битов.

Биты 0 и 1

Биты 0 и 1 определяют формат исходных данных: источником данных может быть упакованный байт или упакованное слово со знаком или без знака.

Биты 2 и 3

Биты 2 и 3 определяют тип применяемой агрегации. Результат обозначается как `IntRes1` (**I**ntermediate **R**esult **1** – промежуточный результат 1). Из второго операнда берется блок из 16 байт и сравнивается с содержимым первого операнда.

Агрегация может выполняться одним из следующих методов:

- равенство любому символу из набора (00) или поиск символа из заданного набора: это означает поиск в операнде 1 любых символов, заданных в операнде 2. При обнаружении совпадения соответствующий бит в `IntRes1` устанавливается в 1. Пример:

```
operand 1: "this is a joke!!"  
operand 2: "i!"  
IntRes1: 0010010000000011
```

- равенство символу из диапазона (01) или поиск символов из заданного диапазона: это означает поиск в операнде 1 любых символов из диапазона, определяемого операндом 2. При обнаружении совпадения соответствующий бит в IntRes1 устанавливается в 1. Пример:

```
operand 1: "this is a joke!!"  
operand 2: "aj"  
IntRes1: 0010010010100100
```

- равенство в любой позиции (10) или сравнение строк: это означает сравнение каждого символа в операнде 1 с соответствующим символом в операнде 2. При обнаружении совпадения соответствующий бит в IntRes1 устанавливается в 1. Пример:

```
operand 1: "this is a joke!!"  
operand 2: "this is no joke!"  
IntRes1: 1111111100000000
```

- равенство в определенном порядке (11) или поиск подстроки: это означает поиск в операнде 1 строки, определенной в операнде 2. При обнаружении совпадения соответствующий бит в IntRes1 устанавливается в 1. Пример:

```
operand 1: "this is a joke!!"  
operand 2: "is"  
IntRes1: 0010010000000000
```

Биты 4 и 5

Биты 4 и 5 применяют знак ориентации (polarity) и сохраняют результат в IntRes2 по следующим правилам:

- положительная ориентация (00) и (10): IntRes2 равнозначен IntRes1. Пример:

```
IntRes1: 0010010000000011  
IntRes2: 0010010000000011
```

- отрицательная ориентация (01) и (11): IntRes2 является дополнительным кодом или логическим отрицанием IntRes1. Пример:

```
IntRes1: 0010010000000011  
IntRes2: 1101101111111100
```

Бит 6

Бит 6 определяет формат вывода в двух возможных вариантах:

- без использования маски:
 - ♦ 0: индекс, возвращаемый в регистре `ecx`, является самым младшим битом, установленным в 1 в `IntRes2`. Пример:

```
IntRes2: 0010010011000000
```

```
ecx = 6
```

В `IntRes2` самый первый бит, установленный в 1, найден по индексу 6 (отсчет начинается с 0 с правой стороны).

- ♦ 1: индекс, возвращаемый в регистре `ecx`, является самым старшим битом, установленным в 1 в `IntRes2`. Пример:

```
IntRes2: 0010010010100100
```

```
ecx = 13
```

В `IntRes2` самый последний бит, установленный в 1, найден по индексу 13 (отсчет начинается с 0 с правой стороны).

- с использованием маски:
 - ♦ 0: `IntRes2` возвращается как маска в младших битах регистра `xmm0` (с распространением нуля до 128 бит). Пример:

Поиск всех символов 'a' и 'e' в строке

```
string = 'qdacdekkfijlmdoz'
```

затем

```
xmm0: 024h
```

или в двоичном формате 0000000000100100

Обратите внимание: в регистре `xmm0` маска реверсирована.

- ♦ 1: `IntRes2` развертывается в байтовую маску / маску слова в регистре `xmm0`. Пример:

Поиск всех символов 'a' и 'e' в строке

```
string = 'qdacdekkfijlmdoz'
```

затем

```
xmm0: 00000000000000000000ff0000ff0000
```

Обратите внимание: в регистре `xmm0` маска реверсирована.

Зарезервированный бит 7

Бит 7 зарезервирован.

Флаги

Для инструкций обработки строк с неявно заданной длиной флаги используются способом, отличающимся от того, что вы видели в предыдущих главах (см. руководства Intel).

Таблица 30.3. Использование флагов для инструкций обработки строк с неявно заданной длиной

Флаг	Описание
CF	Сбрасывается в 0, если <code>IntRes2</code> равен нулю, иначе устанавливается в 1
ZF	Устанавливается в 1, если любой байт/слово в <code>xmm2/mem128</code> является нулевым, иначе сбрасывается в 0
SF	Устанавливается в 1, если любой байт/слово в <code>xmm1</code> является нулевым, иначе сбрасывается в 0
OF	<code>IntRes2[0]</code>
AF	Сбрасывается в 0
PF	Сбрасывается в 0

Для инструкций обработки строк с явно заданной длиной флаги также используются другими способами, как показано в табл. 30.4.

Таблица 30.4. Использование флагов для инструкций обработки строк с явно заданной длиной

Флаг	Описание
CF	Сбрасывается в 0, если <code>IntRes2</code> равен нулю, иначе устанавливается в 1
ZF	Устанавливается в 1, если абсолютное значение регистра <code>EDX < 16 (8)</code> , иначе сбрасывается в 0
SF	Устанавливается в 1, если абсолютное значение регистра <code>EDX < 16 (8)</code> , иначе сбрасывается в 0
OF	<code>IntRes2[0]</code>
AF	Сбрасывается в 0
PF	Сбрасывается в 0

В примерах следующей главы будет использоваться флаг `CF`, для того чтобы определить, был ли получен какой-либо результат, и флаг `ZF` для определения конца строки.

Эта теоретическая часть может показаться немного сложной, поэтому пора перейти к практике.

РЕЗЮМЕ

В этой главе вы узнали о:

- SSE-инструкциях обработки строк;
- схеме расположения битов в управляющем регистре `xmm8` и их использовании.

Глава 31

Поиск символа в строке

В этой главе мы начнем использовать управляющий байт для того, чтобы выполнить поиск заданного символа в строке.

ОПРЕДЕЛЕНИЕ ДЛИНЫ СТРОКИ

В первом примере определяется длина строки методом поиска завершающего 0.

В листинге 31.1 показан исходный код примера.

Листинг 31.1. Программа *sse_string_length.asm*

```
; sse_string_length.asm
extern printf
section .data
; шаблон          0123456789abcdef0123456789abcdef0123456789abcd e
; шаблон          1234567890123456789012345678901234567890123456 7
    string1 db    "The quick brown fox jumps over the lazy river.",0
    fmt1 db      "This is our string: %s ",10,0
    fmt2 db      "Our string is %d characters long.",10,0
section .bss
section .text
    global main
main:
push    rbp
mov     rbp, rsp
    mov     rdi, fmt1
    mov     rsi, string1
    xor     rax, rax
    call    printf
    mov     rdi, string1
    call    pstrlen
    mov     rdi, fmt2
    mov     rsi, rax
    xor     rax, rax
    call    printf
leave
ret
; Функция вычисления длины строки. -----
pstrlen:
push    rbp
```



```

mov    rbp, rsp
mov     rax, -16          ; Исключение изменений в дальнейшем.
xor     xmm0, xmm0        ; 0 (конец строки).
.not_found:
add     rax, 16           ; Исключение изменений ZF в дальнейшем -
                        ; после вызова rcpistri.
rcpistri xmm0, [rdi + rax], 00001000b ; Равенство заданному символу.
jnz     .not_found        ; 0 найден?
add     rax, rcx          ; rcx содержит индекс завершающего 0.
inc     rax               ; Корректировка индекса 0 относительно начала строки.
leave
ret

```

В начале этой программы добавлены два шаблона в комментариях, для того чтобы упростить подсчет символов. В одном шаблоне используется десятичная нумерация, начинающаяся с 1, в другом шаблоне нумерация шестнадцатеричная, начинающаяся с индекса 0.

```

; Шаблон      1234567890123456789012345678901234567890123456 7
; Шаблон      0123456789abcdef0123456789abcdef0123456789abcd e
string1 db    "The quick brown fox jumps over the lazy river.",0

```

Сначала, как обычно, выводятся строки. Затем вызывается специально написанная для этого примера функция поиска `strlen`. Эта функция выполняет поиск в строке (сканирование) первого вхождения нулевого байта. Инструкция `rcpistri` поочередно анализирует блоки из 16 байт, регистр `rax` используется как счетчик блоков. Если `rcpistri` находит нулевой байт в текущем блоке, то устанавливается флаг ZF, который используется для принятия решения о переходе. Необходимо исключить ситуацию, в которой увеличение значения `rax` воздействует на флаг ZF непосредственно перед выполнением перехода, поэтому требуется инкрементирование флага ZF перед инструкцией `rcpistri`. Именно поэтому мы начинаем с записи значения `-16` в регистр `rax`, теперь можно увеличивать `rax` перед использованием `rcpistri`. Обратите внимание на инструкцию `xor`: это логическая инструкция `or` для `xmm`-регистров. Для SIMD существуют отдельные логические инструкции.

Управляющий байт `imm8` содержит значение `00001000`, которое имеет следующий смысл:

- 00 – упакованные беззнаковые байты;
- 10 – равенство любому символу из заданного набора (в данном случае равенство одному символу);
- 00 – положительная ориентация;
- 0 – самый младший индекс;
- 0 – зарезервирован.

Можно было бы предположить использование метода поиска «равенство в любой позиции» (`equal any`) для поиска любого 0. Но вместо этого применяется метод «равенство любому символу из заданного набора» (`equal each`). Почему?

Необходимо всегда помнить о том, что инструкция `rcpistri` инициализирует регистр `rcx` значением 16, т. е. количеством байтов в блоке. Если найден

совпадающий байт, то `rcmpistri` копирует индекс этого совпадающего байта в регистр `ecx`. Если совпадение не обнаружено, то в `ecx` содержится значение 16.

Заглянем в руководства Intel, точнее в Volume 2B, Section 4.1.6 «Valid/Invalid Override of Comparisons». Этот раздел объясняет, что происходит, когда блок содержит «некорректные» байты, т. е. байты, расположенные после конца строки.

Для описания этой ситуации можно воспользоваться табл. 31.1.

Таблица 31.1. Обработка «некорректных» байтов в блоке

<code>xmm0</code>	Память	Равенство в любой позиции (equal any)	Равенство любому символу из заданного набора (equal each)
Некорректное значение	Некорректное значение	Принудительно вычисленное значение false (ложь)	Принудительно вычисленное значение true (истина)
Некорректное значение	Корректное значение	Принудительно вычисленное значение false (ложь)	Принудительно вычисленное значение false (ложь)

В регистре `xmm0` некорректное значение, потому что при инициализации в него было записано значение 0 байт. При получении 16-байтового блока, содержащего нулевой байт, в случае «равенство в любой позиции» (equal any) инструкция `rcmpistri` определяет, что один из 16 байт содержит 0. В этот момент значения в `xmm0` и в памяти некорректны. Но инструкция `rcmpistri` спроектирована так, чтобы выдавать «принудительно вычисленное значение ложь (force false)» в случае «равенство в любой позиции» (equal any). Поэтому `rcmpistri` считает, что совпадений нет, и возвращает 16 в `ecx`, следовательно, вычисление длины строки дает неправильный результат.

Но при использовании метода «равенство любому символу из заданного набора (equal each)» регистр `xmm0` содержит некорректное значение, как и в описанном выше случае, однако как только `rcmpistri` считывает завершающий нулевой байт в блоке, выдается «принудительно вычисленное значение истина (force true)» в соответствии с спроектированным поведением этой инструкции. Индекс нулевого байта записывается в регистр `ecx`. После этого значение `ecx` можно использовать для правильного вычисления конца строки.

Предупреждение: программа считывает блоки размером 16 байт. Это работает нормально, пока данные, в которых выполняется поиск, находятся в пространстве памяти, выделенном этой программе. Если программа пытается читать данные за границей разрешенной памяти, то происходит аварийное завершение. Этого можно избежать, постоянно отслеживая текущее положение на странице памяти (в большинстве случаев страница – это фрагмент памяти размером 4 Кб), и если вы оказываетесь слишком близко к границе страницы, то необходимо перейти на режим чтения по одному байту последовательно. Такой способ обеспечит защиту от случайного пересечения границы разрешенной памяти и вторжения на страницу памяти другого процесса. Здесь эта методика не реализована, чтобы не усложнять пример программы и ее описание. Но вы должны знать о возможности возникновения такой ситуации.

На рис. 31.1 показан вывод этой программы. Можно видеть, что длина строки учитывает завершающий ноль.

```

jo@UbuntuDesktop:~/Desktop/linux64/gcc/36 sse_string_length$ make
nasm -f elf64 -g -F dwarf sse_string_length.asm -l sse_string_length.lst
gcc -o sse_string_length sse_string_length.o -no-pie
jo@UbuntuDesktop:~/Desktop/linux64/gcc/36 sse_string_length$ ./sse_string_length
This is our string: The quick brown fox jumps over the lazy river.
Our string is 47 characters long.
jo@UbuntuDesktop:~/Desktop/linux64/gcc/36 sse_string_length$ █

```

Рис. 31.1. Вывод программы *sse_string_length.asm*

ПОИСК В СТРОКАХ

Теперь, когда известно, как определить длину строки, можно перейти к выполнению поиска в строках (см. листинг 31.2).

Листинг 31.2. Программа *sse_string_search.asm*

```

; sse_string_search.asm
extern printf
section .data
; Шаблон      123456789012345678901234567890123456789012345 6
; Шаблон      0123456789abcdef0123456789abcdef0123456789abc d
string1 db "the quick brown fox jumps over the lazy river",0
string2 db "e",0
fmt1 db "This is our string: %s ",10,0
fmt2 db "The first '%s' is at position %d.",10,0
fmt3 db "The last '%s' is at position %d.",10,0
fmt4 db "The character '%s' didn't show up!.",10,0
section .bss
section .text
    global main
main:
push rbp
mov rbp,rbp
mov rdi,fmt1
mov rsi,string1
xor rax,rax
call printf
; Поиск первого вхождения.
mov rdi,string1
mov rsi,string2
call pstrscan_f
cmp rax,0
je no_show
mov rdi,fmt2
mov rsi,string2
mov rdx,rax
xor rax,rax
call printf
; Поиск последнего вхождения.
mov rdi,string1
mov rsi,string2
call pstrscan_l
mov rdi,fmt3
mov rsi,string2
mov rdx,rax
xor rax,rax

```

```

        call printf
        jmp exit
no_show:
        mov     rdi, fmt4
        mov     rsi, string2
        xor     rax, rax
        call printf
exit:
leave
ret
;----- Поиск первого вхождения. -----
pstrscan_f:
push rbp
mov rbp, rsp
xor rax, rax
pxor xmm0, xmm0
pinsrb xmm0, [rsi], 0
.block_loop:
        pcmpistri xmm0, [rdi + rax], 00000000b
        jc .found
        jz .none
        add rax, 16
        jmp .block_loop
.found:
        add rax, rcx ; rcx содержит позицию символа.
        inc rax ; Начало отсчета с 1 вместо 0.
leave
ret
.none:
xor rax, rax ; Ничего не найдено, возвращается 0.
leave
ret
;----- Поиск последнего вхождения. -----
pstrscan_l:
push rbp
mov rbp, rsp
push rbx ; Регистр, сохраняемый вызываемой функцией.
push r12 ; Регистр, сохраняемый вызываемой функцией.
xor rax, rax
pxor xmm0, xmm0
pinsrb xmm0, [rsi], 0
xor r12, r12
.block_loop:
        pcmpistri xmm0, [rdi + rax], 01000000b
        setz bl
        jc .found
        jz .done
        add rax, 16
        jmp .block_loop
.found:
        mov r12, rax
        add r12, rcx ; rcx содержит позицию символа.
        inc r12
        cmp bl, 1
        je .done
        add rax, 16
        jmp .block_loop

```

```

pop r12                ; Регистр, сохраняемый вызываемой функцией.
pop rbx                ; Регистр, сохраняемый вызываемой функцией.
leave
get
.done:
    mov    гах,г12
pop r12                ; Регистр, сохраняемый вызываемой функцией.
pop rbx                ; Регистр, сохраняемый вызываемой функцией.
leave
get

```

В начале этой программы добавлены два шаблона в комментариях, для того чтобы упростить подсчет символов в строке.

Здесь `string1` содержит строку, в которой производится поиск, а в `string2` содержится искомым аргумент. Выполняется поиск первого и последнего вхождений искомого аргумента. Сначала выводятся строки, затем вызываются специально созданные для этого примера функции. Для поиска первого вхождения заданного символа и его последнего вхождения написаны отдельные функции. Функция `pstrscan_f` выполняет поиск первого вхождения искомого аргумента. Инstrukция `rcmpistri` поочередно обрабатывает блоки размером 16 байт, в качестве счетчика блоков используется регистр `гах`. Регистр `хmm0` очищается инструкцией `pxog`. Инstrukция `pinsrb` позволяет поместить искомым аргумент в младший байт регистра `хmm0` (байт 0). Для поиска вхождений применяется метод «равенство в любой позиции» (`equal any`), и как только найдено вхождение, в `гсх` содержится индекс совпавшего байта в текущем 16-байтовом блоке. Если в текущем блоке не обнаружено совпадение, то в регистр `гсх` записывается значение 16. С помощью инструкции `jc` проверяется флаг `CF=1`. Если флаг установлен, то совпадение найдено, выполняется сложение значения `гсх` со значением `гах`, в котором содержится количество байтов, уже просмотренных в предыдущих блоках, затем значение `гах` возвращается и корректируется для отсчета, начиная с 1 вместо 0.

Если `CF=0`, то с помощью инструкции `jz` определяется, находимся ли мы в последнем блоке. Инstrukция `rcmpistri` устанавливает `ZF=1`, если обнаружен нулевой байт, и регистр `гах` очищается, потому что совпадение не найдено. В этом случае функция возвращает 0.

Разумеется, здесь нет никаких проверок на ошибки: если строка не завершается нулевым байтом, то вы можете получить ошибочные результаты. Попробуйте удалить 0 в конце строки и посмотрите, что получится.

Функция `pstrscan_l` выполняет поиск последнего вхождения искомого аргумента. Это более сложная операция, чем поиск первого вхождения и выход. Требуется прочитать все 16-байтовые блоки и отследить самое последнее вхождение искомого символа в блоке. Но даже когда вхождение обнаружено, необходимо продолжать выполнение цикла до тех пор, пока не встретится завершающий нулевой байт. Для постоянного наблюдения за завершающим нулевым байтом регистр `bl` устанавливается в 1 сразу, как только обнаружен нулевой байт. Регистр `г12` используется для записи индекса самого последнего найденного совпадения. См. рис. 31.2.

```
jo@UbuntuDesktop:~/Desktop/linux64/gcc/36 sse_string_search$ make
nasm -f elf64 -g -F dwarf sse_string_search.asm -l sse_string_search.lst
gcc -o sse_string_search sse_string_search.o -no-pie
jo@UbuntuDesktop:~/Desktop/linux64/gcc/36 sse_string_search$ ./sse_string_search
This is our string: the quick brown fox jumps over the lazy river
The first 'e' is at position 3.
The last 'e' is at position 44.
jo@UbuntuDesktop:~/Desktop/linux64/gcc/36 sse_string_search$ █
```

Рис. 31.2. Вывод программы *sse_string_search.asm*

РЕЗЮМЕ

В этой главе вы узнали, как:

- использовать инструкцию `pscmpistri` для поиска символов и определения длины строки;
- интерпретировать результат выполнения инструкции `pscmpistri` с помощью различных значений управляющего байта `imm8`.

Глава 32

Сравнение строк

В предыдущей главе мы работали со строками с неявно заданной длиной, т. е. со строками, завершающимися нулевым байтом. В этой главе рассматривается сравнение строк с неявно заданной длиной и строк с явно заданной длиной.

СТРОКИ С НЕЯВНО ЗАДАННОЙ ДЛИНОЙ

Вместо поиска совпадений мы будем искать символы, которые отличаются друг от друга. В листинге 32.1 показан пример исходного кода.

Листинг 32.1. Программа *sse_string2_imp.asm*

```
; sse_string2_imp.asm
; Сравнение строк с неявно заданной длиной.
extern printf
section .data
    string1    db    "the quick brown fox jumps over the lazy"
                db    " river",10,0
    string2    db    "the quick brown fox jumps over the lazy"
                db    " river",10,0
    string3    db    "the quick brown fox jumps over the lazy"
                db    " dog",10,0
    fmt1       db    "Strings 1 and 2 are equal.",10,0
    fmt11      db    "Strings 1 and 2 differ at position %i.",10,0
    fmt2       db    "Strings 2 and 3 are equal.",10,0
    fmt22      db    "Strings 2 and 3 differ at position %i.",10,0
section .bss
section .text
    global main
main:
    push    rbp
    mov     rbp, rsp
; Сначала выводятся строки.
    mov     rdi, string1
    xor     rax, rax
    call    printf
    mov     rdi, string2
    xor     rax, rax
    call    printf
    mov     rdi, string3
    xor     rax, rax
    call    printf
```

```

; Сравнение строк 1 и 2.
    mov     rdi, string1
    mov     rsi, string2
    call    pstrcmp
    mov     rdi,fmt1
    cmp     rax,0
    je      eql1          ; Строки равны.
    mov     rdi,fmt11     ; Строки не равны.
eql1:
    mov     rsi, rax
    xor     rax,rax
    call    printf
; Сравнение строк 2 и 3.
    mov     rdi, string2
    mov     rsi, string3
    call    pstrcmp
    mov     rdi,fmt2
    cmp     rax,0
    je      eql2          ; Строки равны.
    mov     rdi,fmt22     ; Строки не равны.
eql2:
    mov     rsi, rax
    xor     rax,rax
    call    printf
; Выход.
leave
ret
; Сравнение строк. -----
pstrcmp:
push     rbp
mov      rbp,rsp
xor      rax, rax          ;
xor      rbx, rbx          ;
.loop:   movdqu    xmm1, [rdi + rbx]
         rscmpistri xmm1, [rsi + rbx], 0x18 ; Метод equal each | отрицательная ориентация.
         jc        .differ
         jz        .equal
         add       rbx, 16
         jmp       .loop
.differ:
    mov     rax,rbx
    add     rax,rcx          ; Позиция отличающегося символа.
    inc     rax              ; Потому что индекс начинается с 0.
.equal:
leave
ret

```

Как обычно, сначала выводятся сами строки, затем вызывается функция `pstrcmp` для сравнения строк. Самая важная информация содержится в функции `pstrcmp`. Управляющий байт содержит значение `0x18` или `00011000`, т. е. справа налево: упакованные целочисленные байты, метод «равенство любому символу из заданного набора» (`equal each`), отрицательная ориентация. Регистр `ecx` содержит индекс первого вхождения. Инструкция `rscmpistri` позволяет воспользоваться флагами – соответствующее описание можно найти в руководствах Intel:

- CFlag – сбрасывается в 0, если IntRes2 равен 0, иначе устанавливается в 1;
- ZFlag – устанавливается в 1, если любой байт/слово в xmm2/mem128 равен нулю, иначе сбрасывается в 0;
- SFlag – устанавливается в 1, если любой байт/слово в xmm1 равен нулю, иначе сбрасывается в 0;
- OFlag – IntRes2[0];
- AFlag – сбрасывается в 0;
- PFlag – сбрасывается в 0.

В рассматриваемом здесь примере инструкция `pscmpistri` записывает 1 для каждого обнаруженного совпадения в соответствующую позицию промежуточного результата `IntRes1`. Если найден отличающийся байт, то в соответствующую позицию `IntRes1` записывается ноль. Затем формируется промежуточный результат `IntRes2` и применяется отрицательная ориентация к `IntRes1`. `IntRes2` будет содержать 1 по индексу отличающегося байта (результат отрицательной ориентации), поэтому `IntRes2` не равен нулю, и флаг CF устанавливается в 1. После этого цикл будет прерван, и функция `pstrcmp` возвращает в регистрах позицию отличающегося символа. Если флаг CF не установлен в 1, но инструкция `pscmpistri` обнаружила завершающий ноль, то функция возвращает в регистрах значение 0.

На рис. 32.1 показан вывод этой программы.

```
jo@ubuntu18:~/Desktop/Book/37 sse_string2_imp$ ./sse_string2
the quick brown fox jumps over the lazy river
the quick brown fox jumps over the lazy river
the quick brown fox jumps over the lazy dog
Strings 1 and 2 are equal.
Strings 2 and 3 differ at position 41.
jo@ubuntu18:~/Desktop/Book/37 sse_string2_imp$ █
```

Рис. 32.1. Вывод программы *sse_string2_imp.asm*

СТРОКИ С ЯВНО ЗАДАННОЙ ДЛИНОЙ

В основном мы используем строки с неявно заданной длиной, но в листинге 32.2 показан пример обработки строк с явно заданной длиной.

Листинг 32.2. Программа *sse_string3_exp.asm*

```
; sse_string3_exp.asm
; Сравнение строк с явно заданной длиной.
extern printf
section .data
    string1      db      "the quick brown fox jumps over the "
                  db      "lazy river"
    string1Len   equ $ - string1
    string2      db      "the quick brown fox jumps over the "
                  db      "lazy river"
    string2Len   equ $ - string2
    dummy       db      "confuse the world"
    string3      db      "the quick brown fox jumps over the "
                  db      "lazy dog"
    string3Len   equ $ - string3
```

```

    fmt1 db "Strings 1 and 2 are equal.",10,0
    fmt11 db "Strings 1 and 2 differ at position %i.",10,0
    fmt2 db "Strings 2 and 3 are equal.",10,0
    fmt22 db "Strings 2 and 3 differ at position %i.",10,0

section .bss
    buffer resb 64
section .text
    global main
main:
push    rbp
mov     rbp, rsp
; Сравнение строк 1 и 2.
    mov     rdi, string1
    mov     rsi, string2
    mov     rdx, string1Len
    mov     rcx, string2Len
    call    pstrcmp
    push    rax ; Запись результата в стек для дальнейшего использования.
; Вывод строк 1 и 2 и результата.
; -----
; Сначала создается строка с символом перехода на новую строку
; и завершающим 0.
; string1
    mov     rsi, string1
    mov     rdi, buffer
    mov     rcx, string1Len
    rep     movsb
    mov     byte[rdi], 10 ; Добавление NL в буфер.
    inc     rdi ; Добавление завершающего 0 в буфер.
    mov     byte[rdi], 0
; Вывод.
    mov     rdi, buffer
    xor     rax, rax
    call    printf
; string2
    mov     rsi, string2
    mov     rdi, buffer
    mov     rcx, string2Len
    rep     movsb
    mov     byte[rdi], 10 ; Добавление NL в буфер.
    inc     rdi ; Добавление завершающего 0 в буфер.
    mov     byte[rdi], 0
; Вывод.
    mov     rdi, buffer
    xor     rax, rax
    call    printf
; -----
; Теперь вывод результата сравнения.
    pop     rax ; Восстановление возвращаемого значения.
    mov     rdi, fmt1
    cmp     rax, 0
    je      eql1
    mov     rdi, fmt11
eql1:
    mov     rsi, rax

```

```
    xor     rax,rax
    call    printf
;-----
;-----
; Сравнение строк 2 и 3.
    mov     rdi, string2
    mov     rsi, string3
    mov     rdx, string2Len
    mov     rcx, string3Len
    call    pstrcmp
    push    rax
; Вывод строки string3 и результата.
;-----
; Сначала создается строка с символом перехода на новую строку
; и завершающим 0.
; string3
    mov     rsi,string3
    mov     rdi,buffer
    mov     rcx,string3Len
    rep     movsb
    mov     byte[rdi],10    ; Добавление NL в буфер.
    inc     rdi             ; Добавление завершающего 0 в буфер.
    mov     byte[rdi],0
; Вывод.
    mov     rdi, buffer
    xor     rax,rax
    call    printf
;-----
; Теперь вывод результата сравнения.
    pop     rax             ; Восстановление возвращаемого значения.
    mov     rdi,fmt2
    cmp     rax,0
    je      eql2
    mov     rdi,fmt22
eql2:
    mov     rsi, rax
    xor     rax,rax
    call    printf
; Выход.
leave
ret
;-----
pstrcmp:
push    rbp
mov     rbp,rsp
    xor     rbx, rbx
    mov     rax,rdx         ;rax содержит длину 1-й строки.
    mov     rdx,rcx         ;rdx содержит длину 2-й строки.
    xor     rcx,rcx         ;rcx как индекс.
.loop:
    movdqu  xmm1, [rdi + rbx]
    rcpstrsi xmm1, [rsi + rbx], 0x18    ; Метод equal each | отр. ориентация.
    jc      .differ
    jz      .equal
    add     rbx, 16
    sub     rax,16
```

```

        sub    rdx,16
        jmp    .loop
.differ:
        mov    rax,rbx
        add    rax,rcx    ; rcx содержит позицию отличающегося символа.
        inc    rax        ; Потому что отсчет начинается с 0.
        jmp    exit
.equal:
        xor    rax,rax
exit:
leave
ret

```

Здесь можно видеть, что обработка строк с явно заданной длиной иногда может становиться достаточно сложной. Так зачем же использовать такие строки? Строки с явно заданной длиной используют многие протоколы обмена информацией. Или, возможно, в каком-либо приложении потребуется использование данных с нулевыми значениями. Так или иначе, необходимо обеспечить представление информации о длине строк. В рассматриваемом здесь примере длина строк вычислялась по локациям памяти в разделе `section .data`. Но функция `printf` ожидает передачи строк, завершающихся нулевым байтом. Поэтому после демонстрации сравнения строк с явно заданной длиной строки переформатируются в буфере: добавляется символ перехода на новую строку и завершающий ноль непосредственно в буфер, после чего отредактированный буфер передается в функцию `printf`.

Теперь рассмотрим функцию сравнения `rstrcmp`. Длина первой строки записывается в регистр `rax`, длина второй строки – в `rdx`. Затем начинается цикл: адрес 16-байтового блока памяти загружается в регистр `xmm1`, и выполняется инструкция `pcmpstrsi` с предварительно установленным значением `0x18` в управляющем байте. Далее проверяем флаги: их описание можно найти в руководствах Intel:

- `CF`flag – сбрасывается в 0, если `IntRes2` равен нулю, иначе устанавливается в 1;
- `ZF`flag – устанавливается в 1, если абсолютное значение `EDX` меньше 16 (8), иначе сбрасывается в 0;
- `SF`flag – устанавливается в 1, если абсолютное значение `EAX` меньше 16 (8), иначе сбрасывается в 0;
- `OF`flag – `IntRes2[0]`;
- `AF`flag – сбрасывается в 0;
- `PF`flag – сбрасывается в 0.

Обратите внимание: инструкции `pcmpstrsi` и `pcmpstrsi` используют флаги `ZF` и `CF` по-разному. Вместо флага `ZF`, сигнализирующего о появлении завершающего нуля, на каждой итерации цикла уменьшаются значения регистров `rax` и `rdx`, и когда значение в одном из этих регистров становится меньше 16, цикл завершается.

На рис. 32.2 показан вывод этой программы.

```
jo@ubuntu18:~/Desktop/Book/38_0 sse_string3_exp$ ./sse_string3
the quick brown fox jumps over the lazy river
the quick brown fox jumps over the lazy river
Strings 1 and 2 are equal.
the quick brown fox jumps over the lazy dog
Strings 2 and 3 differ at position 41.
jo@ubuntu18:~/Desktop/Book/38_0 sse_string3_exp$ █
```

Рис. 32.2. Вывод программы *sse_string3_exp.asm*

РЕЗЮМЕ

В этой главе вы узнали:

- о строках с неявно и явно заданной длиной;
- о смысле отрицательной ориентации;
- об использовании флагов.

Глава 33

Перемешиваем данные

Применяя инструкции обработки строк без использования масок, мы получаем в свое распоряжение несколько возможных вариантов. Можно найти первое или последнее вхождение символа, но поиск всех вхождений является более трудной задачей. Можно сравнивать строки и находить различие, но поиск всех различий не так-то прост. К счастью, существуют еще и инструкции обработки строк, которые используют маски, что придает им гораздо большую мощь. Но, прежде чем начать углубленное изучение инструкций с применением масок, необходимо рассмотреть операции перемешивания.

ОСНОВНЫЕ ПРИНЦИПЫ ОПЕРАЦИЙ ПЕРЕМЕШИВАНИЯ

Перемешивание (shuffling; или тасование, перетасовка) означает перестановку в произвольном порядке упакованных значений. Перестановка может выполняться в одном и том же `xmm`-регистре, или из одного `xmm`-регистра в другой `xmm`-регистр, или из 128-битового блока памяти в `xmm`-регистр.

В листинге 33.1 показан пример исходного кода.

Листинг 33.1. Программа *shuffle.asm*

```
; shuffle.asm
extern printf
section .data
    fmt0 db "These are the numbers in memory: ",10,0
    fmt00 db "This is xmm0: ",10,0
    fmt1 db "%d ",0
    fmt2 db "Shuffle-broadcast double word %i:",10,0
    fmt3 db "%d %d %d %d",10,0
    fmt4 db "Shuffle-reverse double words:",10,0
    fmt5 db "Shuffle-reverse packed bytes in xmm0:",10,0
    fmt6 db "Shuffle-rotate left:",10,0
    fmt7 db "Shuffle-rotate right:",10,0
    fmt8 db "%c%c%c%c%c%c%c%c%c%c%c%c%c%c",10,0
    fmt9 db "Packed bytes in xmm0:",10,0
    NL db 10,0

    number1 dd 1
    number2 dd 2
    number3 dd 3
    number4 dd 4
```

```

char db "abcdefghijklmnp"
bytereverse db 15,14,13,12,11,10,9,8,7,6,5,4,3,2,1,0

section .bss
section .text
    global main
main:
push    rbp
mov     rbp, rsp
    sub     rsp, 32 ; Пространство в стеке для исходного xmm0
                    ; и для измененного xmm0.
; ПЕРЕМЕШИВАНИЕ ДВОЙНЫХ СЛОВ.
; Сначала вывод чисел в обратном порядке.
    mov     rdi, fmt0
    call    printf
    mov     rdi, fmt1
    mov     rsi, [number4]
    xor     rax, rax
    call    printf
    mov     rdi, fmt1
    mov     rsi, [number3]
    xor     rax, rax
    call    printf
    mov     rdi, fmt1
    mov     rsi, [number2]
    xor     rax, rax
    call    printf
    mov     rdi, fmt1
    mov     rsi, [number1]
    xor     rax, rax
    call    printf
    mov     rdi, NL
    call    printf
; Наполнение регистра xmm0 числами.
    pxor     xmm0, xmm0
    pinsrd   xmm0, dword[number1], 0
    pinsrd   xmm0, dword[number2], 1
    pinsrd   xmm0, dword[number3], 2
    pinsrd   xmm0, dword[number4], 3
    movdqu   [rbp-16], xmm0 ; Сохранение xmm0 для дальнейшего использования.
    mov     rdi, fmt00
    call     printf ; Вывод заголовка.
    movdqu   xmm0, [rbp-16] ; Восстановление xmm0 после printf.
    call     print_xmm0d ; Вывод xmm0.
    movdqu   xmm0, [rbp-16] ; Восстановление xmm0 после printf.

; ПЕРЕМЕШИВАНИЕ В ПРОИЗВОЛЬНОМ ПОРЯДКЕ.
; Перемешивание: случайная перестановка младшего двойного слова (индекс 0).
    movdqu   xmm0, [rbp-16] ; Восстановление xmm0.
    pshufd   xmm0, xmm0, 00000000b ; Перемешивание.
    mov     rdi, fmt2
    mov     rsi, 0 ; Вывод заголовка.
    movdqu   [rbp-32], xmm0 ; printf уничтожает xmm0.
    call     printf
    movdqu   xmm0, [rbp-32] ; Восстановление xmm0 после printf.
    call     print_xmm0d ; Вывод содержимого xmm0.

```

```

; Перемешивание: случайная перестановка двойного слова с индексом 1.
movdqu    xmm0,[rbp-16]      ; Восстановление xmm0.
pshufd    xmm0,xmm0,01010101b ; Перемешивание.
mov       rdi,fmt2
mov       rsi, 1              ; Вывод заголовка.
movdqu    [rbp-32],xmm0      ; printf уничтожает xmm0.
call      printf
movdqu    xmm0,[rbp-32]      ; Восстановление xmm0 после printf.
call      print_xmm0d        ; Вывод содержимого xmm0.
; Перемешивание: случайная перестановка двойного слова с индексом 2.
movdqu    xmm0,[rbp-16]      ; Восстановление xmm0.
pshufd    xmm0,xmm0,10101010b ; Перемешивание.
mov       rdi,fmt2
mov       rsi, 2              ; Вывод заголовка.
movdqu    [rbp-32],xmm0      ; printf уничтожает xmm0.
call      printf
movdqu    xmm0,[rbp-32]      ; Восстановление xmm0 после printf.
call      print_xmm0d        ; Вывод содержимого xmm0.
; Перемешивание: случайная перестановка двойного слова с индексом 3.
movdqu    xmm0,[rbp-16]      ; Восстановление xmm0.
pshufd    xmm0,xmm0,11111111b ; Перемешивание.
mov       rdi,fmt2
mov       rsi, 3              ; Вывод заголовка.
movdqu    [rbp-32],xmm0      ; printf уничтожает xmm0.
call      printf
movdqu    xmm0,[rbp-32]      ; Восстановление xmm0 после printf.
call      print_xmm0d        ; Вывод содержимого xmm0.

; ПЕРЕМЕШИВАНИЕ: ПЕРЕСТАНОВКА В ОБРАТНОМ ПОРЯДКЕ.
; Реверсирование двойных слов.
movdqu    xmm0,[rbp-16]      ; Восстановление xmm0.
pshufd    xmm0,xmm0,00011011b ; Перемешивание.
mov       rdi,fmt4
mov       rsi, 1              ; Вывод заголовка.
movdqu    [rbp-32],xmm0      ; printf уничтожает xmm0.
call      printf
movdqu    xmm0,[rbp-32]      ; Восстановление xmm0 после printf.
call      print_xmm0d        ; Вывод содержимого xmm0.

; ПЕРЕМЕШИВАНИЕ: ВРАЩЕНИЕ.
; Вращение влево.
movdqu    xmm0,[rbp-16]      ; Восстановление xmm0.
pshufd    xmm0,xmm0,10010011b ; Перемешивание.
mov       rdi,fmt6
mov       rsi, 1              ; Вывод заголовка.
movdqu    [rbp-32],xmm0      ; printf уничтожает xmm0.
call      printf
movdqu    xmm0,[rbp-32]      ; Восстановление xmm0 после printf.
call      print_xmm0d        ; Вывод содержимого xmm0.
; Вращение вправо.
movdqu    xmm0,[rbp-16]      ; Восстановление xmm0.
pshufd    xmm0,xmm0,00111001b ; Перемешивание.
mov       rdi,fmt7
mov       rsi, 1              ; Вывод заголовка.
movdqu    [rbp-32],xmm0      ; printf уничтожает xmm0.
call      printf
movdqu    xmm0,[rbp-32]      ; Восстановление xmm0 после printf.
call      print_xmm0d        ; Вывод содержимого xmm0.

```


; ПЕРЕМЕШИВАНИЕ БАЙТОВ.

```
mov     rdi, fmt9
call    printf           ; Вывод заголовка.
movdqu  xmm0,[char]      ; Загрузка символа в xmm0.
movdqu  [rbp-32],xmm0     ; printf уничтожает xmm0.
call    print_xmm0b      ; Вывод байтов из xmm0.
movdqu  xmm0,[rbp-32]    ; Восстановление xmm0 после printf.
movdqu  xmm1,[byte reverse] ; Загрузка маски.
pshufb  xmm0,xmm1        ; Перемешивание байтов.
mov     rdi,fmt5         ; Вывод заголовка.
movdqu  [rbp-32],xmm0     ; printf уничтожает xmm0.
call    printf
movdqu  xmm0,[rbp-32]    ; Восстановление xmm0 после printf.
call    print_xmm0b      ; Вывод содержимого xmm0.

leave
ret
```

; Функция для вывода двойных слов.-----

```
print_xmm0d:
push    rbp
mov     rbp,rsi
mov     rdi, fmt3
xor     rax,rax
pextrd  esi, xmm0,3      ; Извлечение двойных слов в обратном порядке,
pextrd  edx, xmm0,2      ; порядок байтов от младшего к старшему (прямой).
pextrd  ecx, xmm0,1
pextrd  r8d, xmm0,0
call    printf

leave
ret
```

; Функция для вывода байтов.-----

```
print_xmm0b:
push    rbp
mov     rbp,rsi
mov     rdi, fmt8
xor     rax,rax
pextrb  esi, xmm0,0      ; В обратном порядке, прямой порядок байтов.
pextrb  edx, xmm0,1      ; Сначала используются регистры,
pextrb  ecx, xmm0,2      ; затем стек.
pextrb  r8d, xmm0,3
pextrb  r9d, xmm0,4
pextrb  eax, xmm0,15
push    rax
pextrb  eax, xmm0,14
push    rax
pextrb  eax, xmm0,13
push    rax
pextrb  eax, xmm0,12
push    rax
pextrb  eax, xmm0,11
push    rax
pextrb  eax, xmm0,10
push    rax
pextrb  eax, xmm0,9
push    rax
```

```

    pextrb    eax, xmm0, 8
    push     rax
    pextrb    eax, xmm0, 7
    push     rax
    pextrb    eax, xmm0, 6
    push     rax
    pextrb    eax, xmm0, 5
    push     rax
    xor      rax, rax
    call     printf
leave
ret

```

Сначала резервируется пространство в стеке для переменных размером 128 байт. Это пространство необходимо для записи `xmm`-регистров в стек. Для `xmm`-регистров нельзя использовать стандартные инструкции `push/pop`, поэтому непременно требуется применение адресов памяти для копирования `xmm`-регистров в стек и для извлечения их из стека. Мы используем регистр `rbp`, указатель базового адреса, как точку отсчета.

Далее выводятся числа, которые будут применяться как упакованные значения. Затем эти числа загружаются в регистр `xmm0` как двойные слова с помощью инструкции `pinsrd` (означающей «packed insert double» – вставка упакованных двойных слов). Содержимое регистра `xmm0` сохраняется в стеке как локальная переменная с помощью инструкции `movdqu [rbp-16], xmm0`. (Для этой локальной переменной было зарезервировано пространство в стеке в начале программы.) При каждом выполнении функции `printf` содержимое регистра `xmm0` будет изменяться независимо от наших намерений. Поэтому придется сохранять и восстанавливать исходное значение регистра `xmm0` при необходимости. Инструкция `movdqu` используется для перемещения невыровненных упакованных целочисленных значений. Для более наглядной визуализации результатов перемешивания при выводе учитывается прямой порядок байтов (от младшего к старшему). Это позволяет увидеть содержимое регистра `xmm0` так, как вы могли бы наблюдать его в отладчике, например в `SASM`.

Для операции перемешивания требуется целевой операнд, операнд-источник и маска перемешивания. Маска определяется в виде непосредственного 8-битового значения. В следующих разделах будут рассматриваться некоторые полезные примеры операций перемешивания и соответствующие маски:

- перемешивание в случайном порядке;
- перемешивание в обратном порядке;
- перемешивание вращением.

ПЕРЕМЕШИВАНИЕ В СЛУЧАЙНОМ ПОРЯДКЕ

Графическая схема поможет лучше понять этот тип перемешивания. На рис. 33.1 показаны четыре примера перемешивания в случайном порядке.

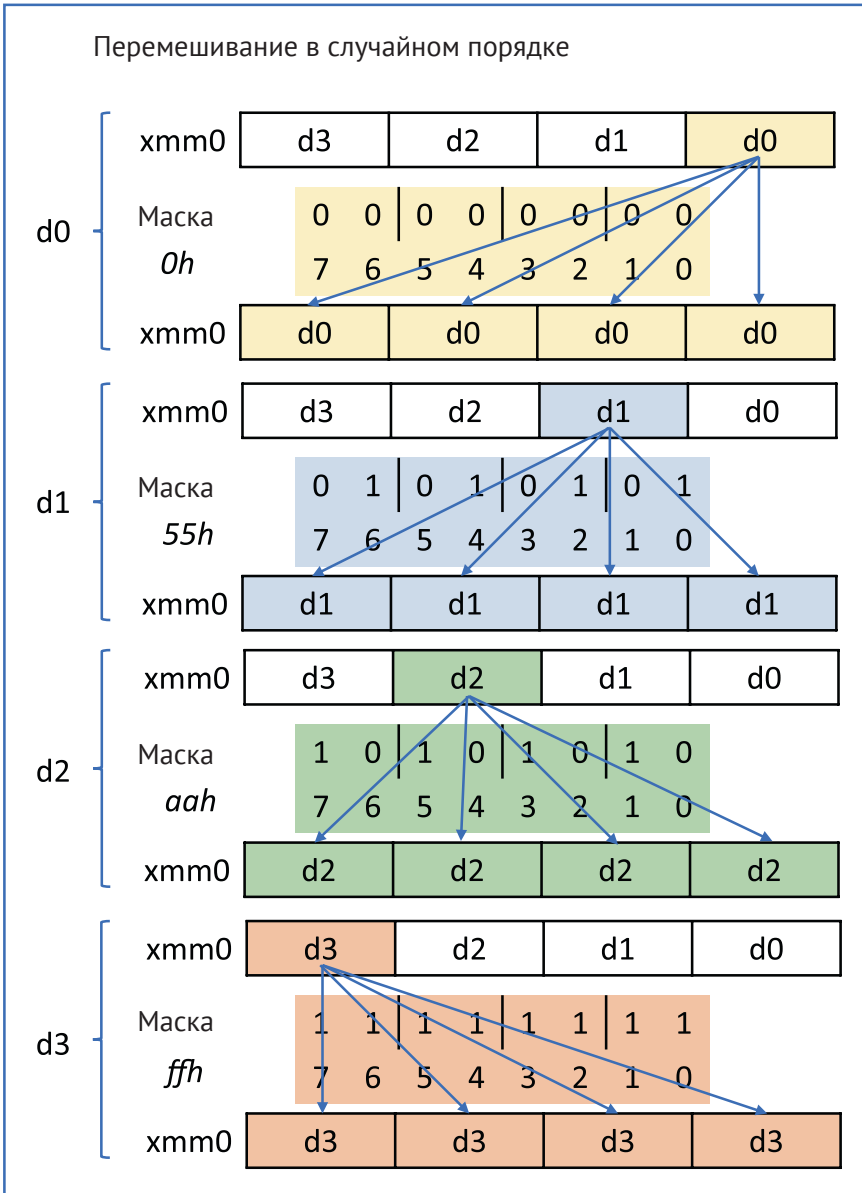


Рис. 33.1. Перемешивание в случайном порядке

На рис. 33.1 источником и целью является регистр `xmm0`. Младшее двойное слово `d0` определено в маске как `00b`. Второе младшее двойное слово `d1` определено как `01b`. Третье двойное слово `d2` определяется как `10b`, а четвертое `d3` – как `11b`. Двоичная маска `10101010b` или `aah` в шестнадцатеричном формате работает следующим образом: `d2` (`10b`) помещается в четыре целевые позиции упакованных двойных слов. Маска `11111111b` должна помещать `d3` (`11b`) в четыре целевые позиции упакованных двойных слов.

При изучении исходного кода вы увидите следующую простую инструкцию перемешивания:

```
pshufd xmm0,xmm0,10101010b
```

В рассматриваемом здесь примере выполняется распространение («рассеивание») третьего младшего элемента в регистре `xmm0`. Поскольку функция `printf` изменяет регистр `xmm0`, необходимо сохранять его содержимое в памяти перед вызовом `printf`. В действительности требуется выполнить больший объем работы для защиты содержимого `xmm0` по сравнению с самой операцией перемешивания.

Разумеется, вы не ограничены четырьмя масками, представленными в этом примере. Можно создать любую 8-битовую маску и перемешивать данные как угодно.

ПЕРЕМЕШИВАНИЕ В ОБРАТНОМ ПОРЯДКЕ

На рис. 33.2 показана общая схема перемешивания в обратном порядке.

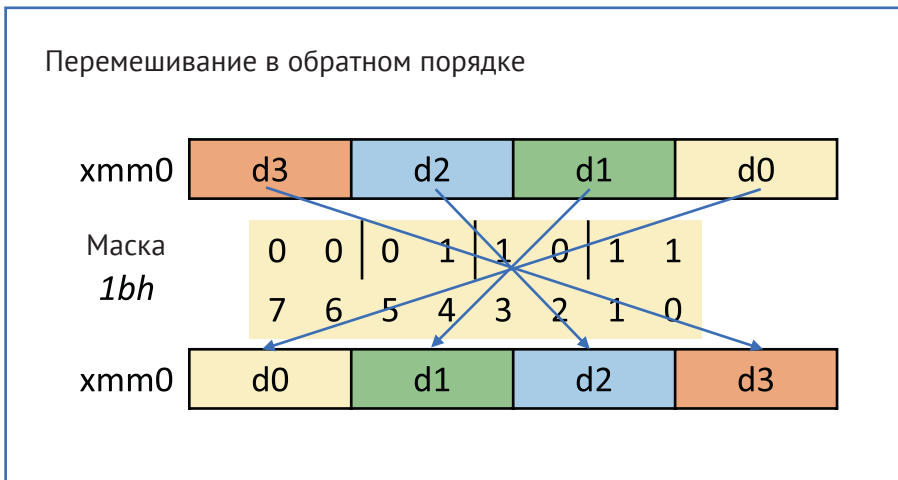


Рис. 33.2. Перемешивание в обратном порядке

Здесь определена маска `00011011b` или `1bh` – она интерпретируется следующим образом:

- 11 (значение в `d3`) перемещается в позицию 0;
- 01 (значение в `d2`) перемещается в позицию 1;
- 10 (значение в `d1`) перемещается в позицию 2;
- 00 (значение в `d0`) перемещается в позицию 3.

В исходном коде примера можно видеть, что операция перемешивания в обратном порядке весьма просто кодируется на языке ассемблера, как показано ниже:

```
pshufd xmm0,xmm0,1bh
```

ПЕРЕМЕШИВАНИЕ ВРАЩЕНИЕМ

Существуют две версии перемешивания вращением: вращение влево и вращение вправо. Это всего лишь вопрос определения правильной маски как последнего аргумента инструкции перемешивания. На рис. 33.3 показана общая схема перемешивания вращением.

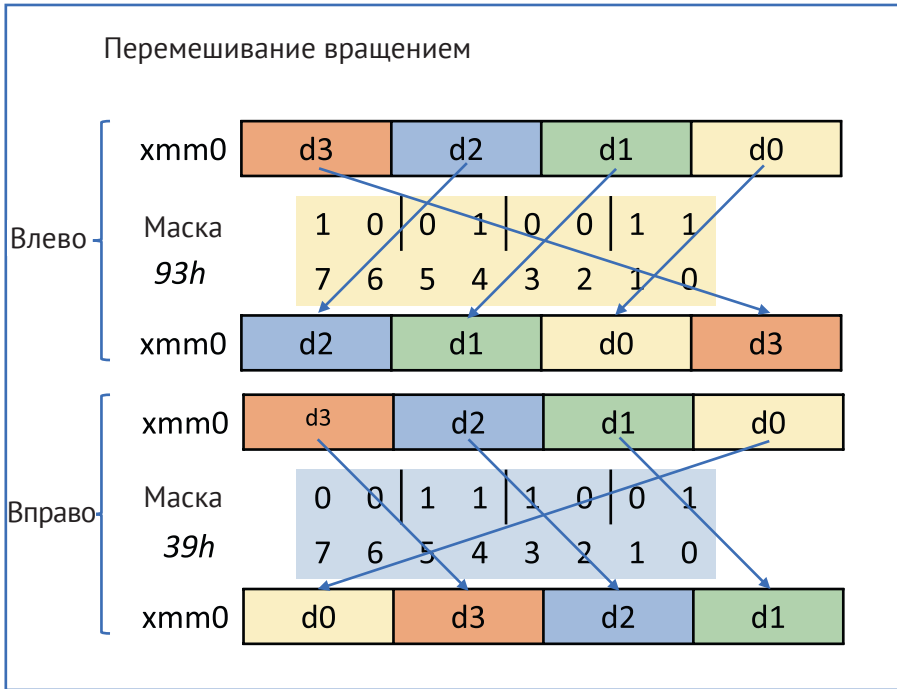


Рис. 33.3. Перемешивание вращением

На языке ассемблера перемешивание вращением выполняется следующими инструкциями:

```
pshufd xmm0,xmm0,93h
pshufd xmm0,xmm0,39h
```

ПЕРЕМЕШИВАНИЕ БАЙТОВ

Двойные слова можно перемешивать с помощью инструкции `pshufd`, а одинарные слова – инструкцией `pshufw`. Также можно перемешивать старшие слова и младшие слова с помощью инструкций `pshufhw` и `pshufld` соответственно. Подробности можно узнать в руководствах Intel. Все перечисленные выше инструкции используют операнд-источник, целевой операнд и маску, определенную непосредственным значением. Представление маски как непосредственного значения имеет свои ограничения: такой способ не обладает гибкостью, поскольку приходится определять маску во время ассемблирования, а не во время выполнения.

Существует решение этой проблемы: перемешивание байтов.

Перемешивание байтов выполняется с помощью инструкции `pshufb`. Эта инструкция принимает только два операнда: операнд целевого `xmm`-регистра и маску, хранящуюся в `xmm`-регистре или в 128-битовом блоке памяти. В приведенном выше исходном коде (см. листинг 33.1) строка `'char'` реверсируется с помощью инструкции `pshufb`. Маска определена в блоке памяти `bytereverse` в разделе `section .data`. Эта маска требует, чтобы байт 15 переместился в позицию 0, байт 14 – в позицию 1 и т. д. Перемешиваемая строка копируется в регистр `xmm0`, а маска помещается в регистр `xmm1`, так что инструкция перемешивания выглядит следующим образом:

```
pshufb xmm0, xmm1
```

Потом происходит магическое превращение. Следует помнить о том, что маска находится во втором операнде, источник совпадает с целью, поэтому результат сохраняется в первом операнде.

Здесь имеется важное преимущество: нет необходимости определять маску как непосредственное значение во время ассемблирования. Маска может быть сформирована в регистре `xmm1` как результат вычислений во время выполнения.

На рис. 33.4 показан итоговый вывод примера программы из листинга 33.1.

```
jo@UbuntuDesktop:~/Desktop/linux64/gcc/38_1 shuffle$ make
nasm -f elf64 -g -F dwarf shuffle.asm -l shuffle.lst
gcc -o shuffle shuffle.o -no-pie
jo@UbuntuDesktop:~/Desktop/linux64/gcc/38_1 shuffle$ ./shuffle
These are the numbers in memory:
4 3 2 1
This is xmm0:
4 3 2 1
Shuffle-broadcast double word 0:
1 1 1 1
Shuffle-broadcast double word 1:
2 2 2 2
Shuffle-broadcast double word 2:
3 3 3 3
Shuffle-broadcast double word 3:
4 4 4 4
Shuffle-reverse double words:
1 2 3 4
Shuffle-rotate left:
3 2 1 4
Shuffle-rotate right:
1 4 3 2
Packed bytes in xmm0:
abcdefghijklmno
Shuffle-reverse packed bytes in xmm0:
ponmlkjihgfedcba
jo@UbuntuDesktop:~/Desktop/linux64/gcc/38_1 shuffle$ █
```

Рис. 33.4. Вывод программы `shuffle.asm`

РЕЗЮМЕ

В этой главе вы узнали:

- об использовании инструкций перемешивания;
- о масках перемешивания;
- о масках, определяемых во время выполнения;
- о том, как использовать стек для работы с xmm-регистрами.

Глава 34

SSE-инструкции: маски строк

Теперь, когда мы знаем, как выполняются операции перемешивания, можно перейти к рассмотрению масок строк.

Напомним, что SSE предоставляет две инструкции обработки строк, которые используют маски: `psrpistrm` и `psrpestrm`. Мы будем использовать инструкции обработки строк с явно заданной длиной. Сначала практическое применение масок кажется сложным, но как только вы поймете сущность этой операции, сразу же убедитесь в том, насколько мощным инструментом может стать маскирование.

Поиск символов

В листингах 34.1, 34.2 и 34.3 показан исходный код примера.

Листинг 34.1. Программа *sse_string4.asm*

```
; sse_string4.asm
; Поиск символа.
extern print16b
extern printf
section .data
    string1    db    "qdacdekkfijlmdoza"
                db    "becdfgdklkmdddaf"
                db    "ffffffffdedeee",10,0
    string2    db    "e",0
    string3    db    "a",0
    fmt        db    "Find all the characters '%s' "
                db    "and '%s' in:",10,0
    fmt_oc     db    "I found %ld characters '%s'"
                db    "and '%s'",10,0
    NL         db    10,0
section .bss
section .text
    global main
main:
    push    rbp
    mov     rbp,rbp

; Вывод искомых символов.
    mov     rdi, fmt
    mov     rsi, string2
```



```

    mov     rdx, string3
    xor     rax, rax
    call    printf
; Вывод целевой строки.
    mov     rdi, string1
    xor     rax, rax
    call    printf
; Поиск в строке и вывод маски.
    mov     rdi, string1
    mov     rsi, string2
    mov     rdx, string3
    call    pcharsrch
; Вывод числа вхождений искомых символов в строке string2.
    mov     rdi, fmt_oc
    mov     rsi, rax
    mov     rdx, string2
    mov     rcx, string3
    call    printf
; Выход.
leave
ret

;-----
; Функция поиска в строке и вывода маски.
pcharsrch:                ; Поиск упакованного символа.
push     rbp
mov      rbp, rsp
sub      rsp, 16          ; Обеспечение пространства в стеке для сохранения xmm1.
xor      r12, r12         ; Для вычисления общего числа вхождений.
xor      rcx, rcx         ; Для оповещения о конце строки.
xor      rbx, rbx         ; Для вычисления адреса.
mov      rax, -16         ; Для счетчика байтов, чтобы избежать установки флага.
; Формирование значения xmm1, загрузка искомых символов.
xor      xmm1, xmm1       ; Очистка регистра xmm1.
pinsrb   xmm1, byte[rsi], 0 ; Первый символ по индексу 0.
pinsrb   xmm1, byte[rdx], 1 ; Второй символ по индексу 1.
.loop:
add      rax, 16          ; Чтобы избежать установки флага ZF.
mov      rsi, 16          ; Если нет завершающего 0, то вывод 16 байт.
movdqu   xmm2, [rdi+rbx]  ; Загрузка 16 байт строки в xmm2.
pcmpestri xmm1, xmm2, 40h ; Метод 'equal each' и 'байт маски в xmm0'.
setz     cl              ; Если найден завершающий 0.
; Если найден завершающий 0, то определить позицию.
cmp      cl, 0
je       .gotoprint       ; Завершающий 0 не найден.
; Завершающий нулевой байт найден.
; Осталось менее 16 байт.
; rdi содержит адрес строки.
; rbx содержит число байтов в блоках, обработанных
; до настоящего момента.
add      rdi, rbx         ; Адрес оставшейся части строки.
push     rcx              ; Регистр, сохр. вызывающей функцией (cl используется).
call     pstrlen          ; В rax возвращается длина.
pop      rcx              ; Регистр, сохраняемый вызывающей функцией.
dec      rax              ; Длина без завершающего 0.
mov      rsi, rax         ; Длина оставшейся маски байтов.

```

```

; Вывод маски.
.gotoprint:
    call print_mask
; Продолжение выполнения для вычисления общего числа совпадений.
    popcnt    r13d,r13d    ; Подсчет числа битов, равных 1.
    add       r12d,r13d    ; Сохранение числа вхождений в r12d.
    or        cl,cl        ; Завершающий 0 обнаружен?
    jnz       .exit
    add       rbx,16        ; Подготовка следующих 16 байт.
    jmp       .loop
.exit:
    mov       rdi,NL        ; Добавление символа перехода на новую строку.
    call      printf
    mov       rax,r12        ; Количество вхождений.
leave
ret

;-----
; Функция поиска завершающего 0.
pstrlen:
push    rbp
mov     rbp,rsb
sub     rsp,16            ; Для сохранения xmm0.
movdqu  [rbp-16],xmm0     ; Запись в стек xmm0.
mov     rax, -16          ; Чтобы далее не устанавливать флаг.
xor     xmm0, xmm0        ; Поиск 0 (конца строки).
.loop:  add     rax, 16      ; Чтобы не устанавливать флаг ZF.
        pcmpistri    xmm0, [rdi + rax], 0x08 ; Метод 'equal each'.
        jnz         .loop    ; Завершающий 0 найден?
        add         rax, rcx   ; rax = число уже обработанных байтов.
                                ; rcx = байты, обработанные в завершающем цикле.
        movdqu      xmm0,[rbp-16] ; Восстановление xmm0 из стека.
leave
ret

;-----
; Функция вывода маски.
; xmm0 содержит маску.
; rsi содержит число битов, которые нужно вывести (16 или меньше).
print_mask:
push    rbp
mov     rbp,rsb
sub     rsp,16            ; Для сохранения xmm0.
call    reverse_xmm0      ; Прямой порядок байтов (от мл. к ст.).
movmskb r13d,xmm0         ; Запись байта маски в r13d.
movdqu  [rbp-16],xmm1     ; Сохранение в стеке xmm1 перед вызовом printf.
push    rdi               ; rdi содержит string1.
mov     edi,r13d          ; Содержит маску, которую нужно вывести.
push    rdx               ; Содержит маску.
push    rcx               ; Содержит флаг конца строки.
call    print16b
pop     rcx
pop     rdx
pop     rdi
movdqu  xmm1,[rbp-16] ; Восстановление xmm1 из стека.
leave
ret

```

```

;-----
; Функция реверсирования, перемешивание xmm0.
reverse_xmm0:
section .data
;mask for reversing
        .bytereverse db 15,14,13,12,11,10,9,8,7,6,5,4,3,2,1,0
section .text
push    rbp
mov     rbp, rsp
sub     rsp, 16
movdqu  [rbp-16], xmm2
movdqu  xmm2, [.bytereverse]      ; Загрузка маски в xmm2.
pshufb  xmm0, xmm2               ; Выполнение перемешивания.
movdqu  xmm2, [rbp-16]           ; Восстановление xmm2 из стека.
leave   ; Возврат перемешанного значения в xmm0.
ret

```

Листинг 34.2. Функция *print16b.c*

```

// print16b.c
#include <stdio.h>
#include <string.h>

void print16b(long long n, int length){
    long long s,c;
    int i=0;
    for (c = 15; c >= 16-length; c--){
        {
            s = n >> c;
            if (s & 1)
                printf("1");
            else
                printf("0");
        }
    }
}

```

Листинг 34.3. *makefile*

```

sse_string4: sse_string4.o print16b.o
    gcc -o sse_string4 sse_string4.o print16b.o -no-pie
sse_string4.o: sse_string4.asm
    nasm -f elf64 -g -F dwarf sse_string4.asm -l sse_string4.lst
printb: print16b.c
    gcc -c print16b.c

```

Основная часть программы достаточно проста, но, как и в предыдущих примерах, сложность программы заключается в том, что необходимо вывести некоторый результат на экран. Можно было бы исключить части, отвечающие за вывод, и воспользоваться отладчиком для наблюдения результатов в регистрах и в памяти. Но преодоление трудностей, связанных с выводом, само по себе занимательно, не так ли?

На рис. 34.1 показан вывод этой программы.

```
jo@ubuntu18:~/Desktop/Book/39 sse_string4$ ./sse_string4
Find all the characters 'e' and 'a' in:
qdacdekkfijlmdozabecdfgdklmdddaaaaaaaaaaadedeee
0010010000000000101000000000000100000000010111
I found 9 characters 'e' and 'a'
jo@ubuntu18:~/Desktop/Book/39 sse_string4$ █
```

Рис. 34.1. Вывод программы *sse_string4.asm*

В рассматриваемом здесь примере программы производится поиск двух символов в строке. Предоставлена строка *string1*, в которой выполняется поиск символа 'e', хранящегося в строке *string2*, и символа 'a', находящегося в строке *string3*.

В этой программе используется несколько функций. Сначала рассмотрим функцию *reverse_xmm0*. Эта функция в качестве аргумента принимает содержимое регистра *xmm0* и изменяет порядок байтов на обратный, используя инструкцию перемешивания. Для этого требуется возможность вывода *xmm0*, начиная с самых младших байтов, т. е. вывода в формате с прямым порядком байтов (от младшего к старшему). Именно поэтому так подробно рассматривались операции перемешивания в предыдущей главе.

Также имеется функция вычисления длины строки *psrln*. Она необходима потому, что чтение выполняется 16-байтовыми блоками. Самый последний блок, вероятно, не будет содержать полные 16 байт, поэтому для этого последнего блока нужно определить позицию завершающего нулевого байта. Это позволит вывести маску, которая имеет ту же длину, что и строка, в которой выполняется поиск.

Специализированная функция *rchargch* принимает три строки как аргументы, и в ней производится основная работа. Сначала выполняются некоторые подготовительные операции, такие как инициализация регистров. Регистр *xmm1* используется как маска – искомые символы сохраняются в *xmm1* с помощью инструкции *pinsrb* (*packed insert bytes* – вставка упакованных байтов). Затем начинается цикл с копированием на каждой итерации 16 байт из строки *string1* в регистр *xmm2* и с поиском заданных символов или завершающего нулевого байта. Применяется инструкция с использованием маски *pcmpistrm* (**p**acked **c**ompare **i**mplicit length **s**tring with a **m**ask – сравнение упакованной строки с явно заданной длиной с использованием маски). Инструкция *pcmpistrm* в качестве третьего операнда принимает непосредственное значение управляющего байта (*imm8*), определяющего, что нужно сделать: в данном случае применяется метод «поиск любого символа из заданного набора (*equal any*)» и «байт маски в регистре *xmm0*». Таким образом, выполняется поиск «любого» символа, который «равен» заданным искомым строкам. Для каждого совпавшего символа в *xmm2* устанавливается бит (в 1) в регистре *xmm0*, который соответствует позиции символа, для которого обнаружено совпадение (в *xmm2*). Инструкция *pcmpistrm* не работает с регистром *xmm0* как с операндом, но этот регистр используется неявно. Возвращаемая маска всегда будет храниться в регистре *xmm0*.

Инструкция *pcmpistri* отличается тем, что возвращает индекс единичного бита, т. е. позиции совпадения символов в регистре *ecx*. А инструкция *pcmpistrm* возвращает все позиции совпадения в регистре *xmm0* для 16-байтового блока.

Это позволяет значительно сократить количество шагов для выполнения поиска всех совпадающих символов.

Можно использовать битовую маску или байтовую маску для `xmm0` (для этого необходимо установить или сбросить бит 6 в управляющем байте). В рассматриваемом здесь примере использовалась байтовая маска, чтобы можно было более просто наблюдать за регистром `xmm0` в отладчике – два значения `ff` в `xmm0` соответствуют байту, в котором все биты установлены в 1.

После обследования первого 16-байтового блока проверяется, найден ли завершающий нулевой блок, и результат этой проверки сохраняется в регистре `cl` для дальнейшего использования. Требуемый вывод маски, хранящейся в регистре `xmm0`, выполняется с помощью функции `print_mask`. Обратите внимание: в отладчике эта байтовая маска реверсирована в регистре `xmm0`, чтобы соответствовать формату с прямым порядком байтов (от младшего к старшему). Поэтому перед выводом необходимо реверсировать маску – это делается в функции `reverse_xmm0`. Затем вызывается C-функция `print16b` для вывода реверсированной маски. Но регистр `xmm0` нельзя передать как аргумент в `print16b`, так как внутри эта функция использует стандартную функцию `printf`, которая интерпретирует `xmm0` как значение с плавающей точкой, а не как байтовую маску. Поэтому перед вызовом `print16b` байтовая маска передается из `xmm0` в `r13d` с помощью инструкции `movmskb` (означающей «move byte mask» – перемещение байтовой маски). В дальнейшем `r13d` используется как счетчик, а для вывода его значение копируется в регистр `edi`. Значение `xmm1` сохраняется в стеке для дальнейшего использования.

Вызывается C-функция `print16b` для вывода маски. Эта функция в качестве аргументов принимает `edi` (маска) и `rsi` (длина, передаваемая вызывающей функцией).

Сразу после возвращения в функцию `rcharsrch` подсчитывается количество единиц в `r13d` с помощью инструкции `popcnt` и обновляется счетчик в `r12d`. Также определяется, нужно ли выйти из цикла, потому что в блоке байтов был обнаружен завершающий нулевой байт.

Перед вызовом `print_mask`, если найден завершающий нулевой байт, истинная длина последнего блока определяется с помощью функции `strlen`. Начальный адрес этого блока вычисляется сложением значения `gbx`, содержащего количество уже проверенных байтов из предыдущих блоков, с `rdi`, где хранится адрес строки `string1`. Длина строки, возвращаемая в `rax`, используется для вычисления количества оставшихся байтов маски в `xmm0`, которые передаются в `rsi` для вывода.

Организация вывода представляет собой весьма интересное занятие, согласитесь?

Не воспринимайте часть программы, в которой осуществляется вывод, как ошеломляюще сложную. Сосредоточьтесь на том, как работают маски, – это главная цель текущей главы.

Что можно сделать с маской, возвращаемой инструкцией `pcmpistrm`? Например, можно использовать итоговую маску для подсчета всех вхождений искомого аргумента или для поиска всех вхождений и замены их на что-либо другое, создавая собственную версию функции поиска с заменой.

Теперь рассмотрим другой вариант поиска.

ПОИСК СИМВОЛОВ ИЗ ЗАДАННОГО ДИАПАЗОНА

Диапазон может содержать любое количество искомых символов, например все символы верхнего регистра, все символы между а и к, все символы, представляющие цифры, и т. д.

В листинге 34.4 показан пример поиска в строке символов верхнего регистра.

Листинг 34.4. Программа *sse_string5.asm*

```
; sse_string5.asm
; Поиск символов из заданного диапазона.
extern print16b
extern printf
section .data
    string1      db    "eeAecdckkFijlmeoZa"
                  db    "bcefgeKlkmeDad"
                  db    "fdsafadfaseeE",10,0
    startrange   db    "A",10,0          ; Поиск символов верхнего регистра.
    stoprange    db    "Z",10,0
    NL           db    10,0
    fmt          db    "Find the uppercase letters in:",10,0
    fmt_oc       db    "I found %ld uppercase letters",10,0
section .bss
section .text
    global main
main:
    push    rbp
    mov     rbp,rbp
; Сначала выводится исходная строка.
    mov     rdi,fmt          ; Заголовок.
    xor     rax,rax
    call    printf
    mov     rdi,string1      ; Строка.
    xor     rax,rax
    call    printf
; Поиск в строке.
    mov     rdi,string1
    mov     rsi,startrange
    mov     rdx,stoprange
    call    prangesrch
; Вывод количества найденных вхождений.
    mov     rdi,fmt_oc
    mov     rsi,rax
    xor     rax,rax
    call    printf
leave
ret

;-----
; Функция поиска и вывода маски.
prangesrch:          ; Упакованный диапазон символов для поиска.
    push    rbp
    mov     rbp,rbp
    sub     rsp,16          ; Пространство в стеке для сохранения xmm1.
    xor     r12,r12         ; Для числа найденных вхождений.
    xor     rcx,rcx         ; Для оповещения о конце строки.
```

```

    xog    rbx,rbx    ; Для вычисления адреса.
    mov    rax,-16    ; Чтобы избежать установки флага ZF.
; Формирование значения xmm1.
    pxor   xmm1,xmm1    ; Обеспечение очистки xmm1.
    pinsrb xmm1,byte[rsi],0 ; startrange по индексу 0.
    pinsrb xmm1,byte[rdx],1 ; stoprange по индексу 1.
.loop:
    add     rax,16
    mov     rsi,16      ; Если нет завершающего 0, вывод 16 байт.
    movdqu  xmm2,[rdi+rbx]
    pcmpestri  xmm1,xmm2,01000100b ; Метод equal each|байт.маска в xmm0.
    setz    cl          ; Обнаружен завершающий 0.
; Если найден завершающий 0, то определить позицию.
    cmp     cl,0
    je      .gotoprint  ; Завершающий 0 не найден.
    ; Завершающий нулевой байт найден.
    ; Осталось меньше 16 байт.
    ; rdi содержит адрес строки.
    ; rbx содержит число байтов в блоках, обработанных ранее.
    add     rdi,rbx     ; Взять только хвост этой строки.
    push    rcx          ; Регистр, сохр. вызывающей функцией (используется cl).
    call    pstrlen      ; Определение позиции завершающего 0.
    pop     rcx          ; Регистр, сохраняемый вызывающей функцией.
    dec     rax          ; Длина без завершающего 0.
    mov     rsi,rax      ; Количество байтов в хвосте строки.

; Вывод маски.
.gotoprint:
    call    print_mask
; Продолжать вычисление общего числа найденных совпадений.
    porcnt r13d, r13d    ; Подсчет числа битов, равных 1.
    add     r12d, r13d    ; Сохранение числа вхождений в r12.
    or      cl,cl        ; Завершающий 0 обнаружен?
    jnz     .exit
    add     rbx,16        ; Подготовка следующего блока.
    jmp     .loop
.exit:
    mov     rdi, NL
    call    printf
    mov     rax, r12      ; Возврат числа найденных вхождений.
leave
ret

;-----
pstrlen:
push    rbp
mov     rbp,rbp
    sub     rsp,16        ; Для сохранения в стеке xmm0.
    movdqu  [rbp-16],xmm0 ; Сохранение в стеке xmm0.
    mov     rax, -16      ; Чтобы далее избежать установки флага ZF.
    pxor    xmm0, xmm0    ; Поиск 0 (конца строки).
.loop:
    add     rax, 16        ; Чтобы избежать установки ZF, если rax = 0 после pcmpestri.
    pcmpestri  xmm0, [rdi + rax], 0x08 ; Метод 'equal each'.
    jnz     .loop          ; 0 найден?
    add     rax, rcx        ; rax = число уже обработанных байтов.

```

```

; gscx = число байтов, обработанных
;          на последней итерации цикла.
movdqu    xmm0,[rbp-16] ; Восстановление из стека xmm0.
leave
ret

;-----
; Функция вывода маски.
; xmm0 содержит маску.
; rsi содержит число выводимых битов (16 или меньше).
print_mask:
push rbp
mov rbp, rsp
sub rsp, 16 ; Для сохранения в стеке xmm0.
call reverse_xmm0 ; Прямой порядок байтов (от мл. к ст.).
movmskb r13d, xmm0 ; Запись байтовой маски в r13d.
movdqu [rbp-16], xmm1 ; Сохранение в стеке xmm1 перед вызовом printf.
push rdi ; rdi содержит string1.
mov edi, r13d ; Содержит выводимую маску.
push rdx ; Содержит маску.
push rcx ; Содержит флаг конца строки.
call print16b
pop rcx
pop rdx
pop rdi
movdqu xmm1, [rbp-16] ; Восстановление из стека xmm1.
leave
ret

;-----
; Функция реверсирования, перемешивания xmm0.
reverse_xmm0:
section .data
;mask for reversing
    .bytereverse db 15,14,13,12,11,10,9,8,7,6,5,4,3,2,1,0
section .text
push rbp
mov rbp, rsp
sub rsp, 16
movdqu [rbp-16], xmm2
movdqu xmm2, [.bytereverse] ; Загрузка маски в xmm2.
pshufb xmm0, xmm2 ; Выполнение перемешивания.
movdqu xmm2, [rbp-16] ; Восстановление из стека xmm2.
leave ; Возврат перемешанного значения xmm0.
ret

```

Эта программа делает почти то же самое, что и предыдущая (листинг 34.1), просто в данном примере строки string2 и string3 получили более осмысленные имена. Здесь наиболее важно изменение управляющего байта так, чтобы в инструкцию `pscmpstrm` передавалось значение `01000100b`, означающее «поиск каждого символа из заданного диапазона» (equal range) и «байтовая маска в регистре `xmm0`».

Организация вывода точно такая же, как в предыдущем разделе.

На рис. 34.2 показан вывод этой программы.


```
jo@ubuntu18:~/Desktop/Book/40 sse_string5$ ./sse_string5
Find the uppercase letters in:
eeAecdkkFijlmeoZabcefgeKlkmeDadfdsafadfaseeE
0010000001000000010000000010000010000000000000001
I found 6 uppercase letters
jo@ubuntu18:~/Desktop/Book/40 sse_string5$ █
```

Рис. 34.2. Вывод программы *sse_string5.asm*

Теперь рассмотрим еще один пример.

Поиск подстроки

В листинге 34.5 показан исходный код примера.

Листинг 34.5. Программа *sse_string6.asm*

```
; sse_string6.asm
; Поиск подстроки.
extern print16b
extern printf
section .data
    string1    db    "a quick pink dinosaur jumps over the "
               db    "lazy river and the lazy dinosaur "
               db    "doesn't mind",10,0
    string2    db    "dinosaur",0
    NL         db    10,0
    fmt        db    "Find the substring '%s' in:",10,0
    fmt_oc     db    "I found %ld %ss",10,0

section .bss
section .text
    global main
main:
    push    rbp
    mov     rbp,rbp
; Сначала выводятся строки.
    mov     rdi,fmt
    mov     rsi,string2
    xor     rax,rax
    call    printf
    mov     rdi,string1
    xor     rax,rax
    call    printf
; Поиск в строке.
    mov     rdi,string1
    mov     rsi,string2
    call    psubstringsrch
; Вывод числа найденных вхождений заданной подстроки.
    mov     rdi,fmt_oc
    mov     rsi,rax
    mov     rdx,string2
    call    printf
leave
ret
```

```

;-----
; Функция поиска подстроки и вывода маски.
psubstringsrch:      ; Поиск упакованной подстроки.
push  rbp
mov   rbp,rsp
    sub    rsp,16      ; Пространство для сохранения в стеке xmm1.
    xor    r12,r12     ; Вычисление общего числа вхождений.
    xor    rcx,rcx     ; Для оповещения о конце строки.
    xor    rbx,rbx     ; Для вычисления адреса.
    mov    rax,-16     ; Чтобы избежать установки флага ZF.
; Формирование значения xmm1, загрузка подстроки.
    xor    xmm1,xmm1
    movdqu xmm1,[rsi]
.loop:
    add    rax,16      ; Чтобы избежать установки флага ZF.
    mov    rsi,16      ; Если нет завершающего 0, вывод 16 байт.
    movdqu xmm2,[rdi+rbx]
    pcmpstrm xmm1,xmm2,01001100b ; Метод 'equal ordered'|'байтовая маска в xmm0'.
    setz   cl          ; Обнаружен завершающий 0.

; Если найден завершающий 0, то определить позицию.
    cmp    cl,0
    je     .gotoprint  ; Завершающий 0 не найден.
    ; Найден завершающий нулевой байт.
    ; Осталось меньше 16 байт.
    ; rdi содержит адрес строки.
    ; rbx содержит число байтов в блоках, обработанных ранее.
    add    rdi,rbx     ; Взять только хвост строки.
    push   rcx          ; Регистр, сохр. вызывающей функцией (используется cl).
    call   pstrlen     ; В rax возвращается позиция завершающего 0.
    push   rcx          ; Регистр, сохр. вызывающей функцией (используется cl).
    dec    rax          ; Длина без завершающего 0.
    mov    rsi,rax     ; Длина оставшихся байтов.

; Вывод маски.
.gotoprint:
    call   print_mask
; Продолжение вычисления общего числа найденных совпадений.
    popcnt r13d,r13d    ; Подсчет числа битов, равных 1.
    add    r12d,r13d    ; Сохранение числа найденных вхождений в r12.
    or     cl,cl        ; Завершающий 0 обнаружен?
    jnz    .exit
    add    rbx,16       ; Подготовка следующего блока.
    jmp    .loop
.exit:
    mov    rdi,NL
    call   printf
    mov    rax,r12     ; Возврат числа найденных вхождений.
leave
ret

;-----
pstrlen:
push  rbp
mov   rbp,rsp

```

```

    sub    rsp,16          ; Для сохранения в стеке xmm0.
    movdqu [rbp-16],xmm0   ; Сохранение в стеке xmm0.
    mov    rax, -16        ; Чтобы далее избежать установки флага ZF.
    рхог   xmm0, xmm0      ; Поиск 0 (конца строки).
.loop:
    add    rax, 16 ; Чтобы избежать установки флага ZF, если rax = 0 после рсмрпstrі.
    рсмрпstrі xmm0, [rdi + rax], 0x08 ; Метод 'equal each'
    jnz    .loop ; 0 найден?
    add    rax, rсх ; rax = число уже обработанных байтов.
                ; rсх = число байтов, обработанных на последней итерации цикла.
    movdqu xmm0,[rbp-16] ; Восстановление из стека xmm0.
leave
ret

```

```

;-----
; Функция вывода маски.
; xmm0 содержит маску.
; rsi содержит число выводимых битов (16 или меньше).
print_mask:
push    rbp
mov     rbp,rsp
    sub    rsp,16          ; Для сохранения в стеке xmm0.
    call   reverse_xmm0    ; Прямой порядок байтов (от младшего к старшему).
    рmovmskb r13d,xmm0     ; Запись байтовой маски в edx.
    movdqu [rbp-16],xmm1   ; Сохранение в стеке xmm1 перед вызовом printf.
    push   rdi             ; rdi содержит string1.
    mov    edi,r13d        ; Содержит выводимую маску.
    push   rdx             ; Содержит маску.
    push   rсх             ; Содержит флаг конца строки.
    call   print16b
    pop    rсх
    pop    rdx
    pop    rdi
    movdqu xmm1,[rbp-16] ; Восстановление из стека xmm1.
leave
ret

```

```

;-----
; Функция реверсирования, перемешивания xmm0.
reverse_xmm0:
section .data
; Маска для реверсирования.
    .bytereverse db 15,14,13,12,11,10,9,8,7,6,5,4,3,2,1,0
section .text
push    rbp
mov     rbp,rsp
    sub    rsp,16
    movdqu [rbp-16],xmm2
    movdqu xmm2,[.bytereverse] ; Загрузка маски в xmm2.
    pshufb xmm0,xmm2          ; Выполнение перемешивания.
    movdqu xmm2,[rbp-16]      ; Восстановление из стека xmm2.
leave
ret
; Возврат перемешанного значения xmm0.

```


Глава 35

AVX

Advanced Vector Extensions (AVX) – это расширение системы команд SSE для процессоров x86. Система SSE предлагает 16 xmm-регистров размером 128 бит каждый, тогда как система AVX предоставляет 16 ymm-регистров размером 256 бит каждый. Нижняя (младшая) половина каждого ymm-регистра в действительности представляет собой соответствующий xmm-регистр. AVX-512 – это дальнейшее расширение, предоставляющее 32 zmm-регистра размером 512 бит каждый.

В дополнение к ymm-регистрам система AVX расширяет инструкции SSE и предоставляет набор дополнительных новых инструкций. После изучения глав этой книги, посвященных SSE, вы не будете испытывать затруднения при освоении многочисленных инструкций SSE и AVX.

В этой главе сначала будет показано, как определить версию AVX, поддерживаемую конкретным процессором, затем рассматривается пример программы с использованием AVX.

ПРОВЕРКА ПОДДЕРЖКИ AVX

В листинге 35.1 показан исходный код программы, определяющей, поддерживает ли процессор расширение AVX.

Листинг 35.1. Программа *cpu_avx.asm*

```
; cpu_avx.asm
extern printf
section .data
    fmt_noavx    db    "This cpu does not support AVX.",10,0
    fmt_avx      db    "This cpu supports AVX.",10,0
    fmt_noavx2   db    "This cpu does not support AVX2.",10,0
    fmt_avx2     db    "This cpu supports AVX2.",10,0
    fmt_noavx512 db    "This cpu does not support AVX-512.",10,0
    fmt_avx512   db    "This cpu supports AVX-512.",10,0
section .bss
section .text
    global main
main:
    push    rbp
    mov     rbp,rbp
    call    cpu_sse    ; Возврат 1 в eax, если AVX поддерживается, иначе 0.
```

```

leave
ret

cpu_sse:
push rbp
mov rbp, rsp
; Проверка поддержки avx.
    mov     eax, 1          ; Запрос флагов характеристик ЦП.
    cpushid
    mov     eax, 28         ; Проверка бита 28 в ecx.
    bt      ecx, eax
    jnc     no_avx
    xor     rax, rax
    mov     rdi, fmt_avx
    call    printf
; Проверка поддержки avx2.
    mov     eax, 7          ; Запрос флагов характеристик ЦП.
    mov     ecx, 0
    cpushid
    mov     eax, 5          ; Проверка бита 5 в ebx
    bt      ebx, eax
    jnc     the_exit
    xor     rax, rax
    mov     rdi, fmt_avx2
    call    printf
; Проверка поддержки основных функций avx512.
    mov     eax, 7          ; Запрос флагов характеристик ЦП.
    mov     ecx, 0
    cpushid
    mov     eax, 16         ; Проверка бита 16 в ebx
    bt      ebx, eax
    jnc     no_avx512
    xor     rax, rax
    mov     rdi, fmt_avx512
    call    printf
    jmp     the_exit
no_avx:
    mov     rdi, fmt_noavx
    xor     rax, rax
    call    printf          ; Вывод сообщения, если AVX не поддерживается.
    xor     rax, rax        ; Возврат 0, AVX не поддерживается.
    jmp     the_exit        ; Выход.
no_avx2:
    mov     rdi, fmt_noavx2
    xor     rax, rax
    call    printf          ; Вывод сообщения, если AVX не поддерживается.
    xor     rax, rax        ; Возврат 0, AVX не поддерживается.
    jmp     the_exit        ; Выход.
no_avx512:
    mov     rdi, fmt_noavx512
    xor     rax, rax
    call    printf          ; Вывод сообщения, если AVX не поддерживается.
    xor     rax, rax        ; Возврат 0, AVX не поддерживается.
    jmp     the_exit        ; Выход.
the_exit:
leave
ret

```

Эта программа похожа на программу, которая использовалась для проверки поддержки SSE, но здесь необходимо проверять флаги AVX. В этом нет ничего особенного, а более подробные сведения об используемых регистрах и о том, какую информацию можно извлечь из них, можно найти в руководствах Intel, Volume 2, в разделе, описывающем `cpuid`.

На рис. 35.1 показан вывод этой программы.

```
jo@ubuntu18:~/Desktop/Book/42 cpu_avx$ ./cpu_avx
This cpu supports AVX.
This cpu supports AVX2.
This cpu does not support AVX-512.
jo@ubuntu18:~/Desktop/Book/42 cpu_avx$ █
```

Рис. 35.1. Вывод программы *cpu_avx.asm*

ПРИМЕР ПРОГРАММЫ С ИСПОЛЬЗОВАНИЕМ AVX

В листинге 35.2 показан измененный соответствующим образом исходный код примера SSE-программы без выравнивания данных из главы 28.

Листинг 35.2. Программа *avx_unaligned.asm*

```
; avx_unaligned.asm
extern printf
section .data
    spvector1    dd    1.1
                  dd    2.1
                  dd    3.1
                  dd    4.1
                  dd    5.1
                  dd    6.1
                  dd    7.1
                  dd    8.1
    spvector2    dd    1.2
                  dd    1.2
                  dd    3.2
                  dd    4.2
                  dd    5.2
                  dd    6.2
                  dd    7.2
                  dd    8.2
    dpvector1    dq    1.1
                  dq    2.2
                  dq    3.3
                  dq    4.4
    dpvector2    dq    5.5
                  dq    6.6
                  dq    7.7
                  dq    8.8

    fmt1    db    "Single Precision Vector 1:",10,0
    fmt2    db    10,"Single Precision Vector 2:",10,0
    fmt3    db    10,"Sum of Single Precision Vector 1 and Vector 2:",10,0
    fmt4    db    10,"Double Precision Vector 1:",10,0
```

```

fmt5    db    10,"Double Precision Vector 2:",10,0
fmt6    db    10,"Sum of Double Precision Vector 1 and Vector 2:",10,0

section .bss
    spvector_res    resd    8
    dpvector_res    resq    4
section .text
    global main
main:
push    rbp
mov     rbp,rbp
; ВЕКТОРЫ С ПЛАВАЮЩЕЙ ТОЧКОЙ ОБЫЧНОЙ ТОЧНОСТИ.
; Загрузка vector1 в регистр xmm0.
    vmovups    xmm0, [spvector1]
; Извлечение из xmm0.
    vextractf128    xmm2,xmm0,0    ; Первая часть xmm0.
    vextractf128    xmm2,xmm0,1    ; Вторая часть xmm0.
; Загрузка vector2 в регистр xmm1.
    vmovups    xmm1, [spvector2]
; Извлечение из xmm1.
    vextractf128    xmm2,xmm1,0
    vextractf128    xmm2,xmm1,1
; Сложение 2 векторов с плавающей точкой обычной точности.
    vaddps    xmm2,xmm0,xmm1
    vmovups    [spvector_res],xmm2
; Вывод векторов.
    mov     rdi,fmt1
    call    printf
    mov     rsi,spvector1
    call    printspfpv
    mov     rdi,fmt2
    call    printf
    mov     rsi,spvector2
    call    printspfpv
    mov     rdi,fmt3
    call    printf
    mov     rsi,spvector_res
    call    printspfpv

; ВЕКТОРЫ С ПЛАВАЮЩЕЙ ТОЧКОЙ ДВОЙНОЙ ТОЧНОСТИ.
; Загрузка vector1 в регистр xmm0.
    vmovups    xmm0, [dpvector1]
; Извлечение из xmm0.
    vextractf128    xmm2,xmm0,0    ; Первая часть xmm0.
    vextractf128    xmm2,xmm0,1    ; Вторая часть xmm0.
; Загрузка vector2 в регистр xmm1.
    vmovups    xmm1, [dpvector2]
; Извлечение из xmm1.
    vextractf128    xmm2,xmm1,0
    vextractf128    xmm2,xmm1,1
; Сложение 2 векторов с плавающей точкой двойной точности.
    vaddpd    xmm2,xmm0,xmm1
    vmovupd    [dpvector_res],xmm2
; Вывод векторов.
    mov     rdi,fmt4
    call    printf
    mov     rsi,dpvector1

```



```
    call    printdpfpv
    mov     rdi,fmt5
    call    printf
    mov     rsi,dpvector2
    call    printdpfpv
    mov     rdi,fmt6
    call    printf
    mov     rsi,dpvector_res
    call    printdpfpv
leave
ret
```

```
printspfvp:
section .data
    .NL    db    10,0
    .fmt1   db    "%.1f, ",0
section .text
push    rbp
mov     rbp,rsp
    push   rcx
    push   rbx
    mov    rcx,8
    mov    rbx,0
    mov    rax,1
.loop:
    movss  xmm0,[rsi+rbx]
    cvtss2sd xmm0,xmm0
    mov     rdi,.fmt1
    push    rsi
    push    rcx
    call    printf
    pop     rcx
    pop     rsi
    add     rbx,4
    loop    .loop
    xor     rax,rax
    mov     rdi,.NL
    call    printf
    pop     rbx
    pop     rcx
leave
ret
```

```
printdpfpv:
section .data
    .NL    db    10,0
    .fmt    db    "%.1f, %.1f, %.1f, %.1f",0
section .text
push    rbp
mov     rbp,rsp
    mov    rdi,.fmt
    mov    rax,4      ; Четыре числа с плавающей точкой.
    call    printf
    mov    rdi,.NL
    call    printf
leave
ret
```

В этой программе используются 256-битовые умм-регистры и несколько новых инструкций. Например, инструкция `vmovups` применяется для записи невыровненных данных в умм-регистр. Воспользуемся SASM для просмотра содержимого регистров. После выполнения инструкций `vmovups` регистр `ymm0` содержит следующие значения:

```
{0x4083333340466666400666663f8cccd, 0x4101999a40e3333340c3333340a33333}
```

А вот как эти значения выглядят после преобразования в десятичный формат:

```
{4.1 3.1 2.1 1.1 , 8.1 7.1 6.1 5.1}
```

Посмотрите, где хранятся эти значения, – это может показаться не совсем понятным.

Исключительно в демонстрационных целях данные извлекаются из умм-регистра, и используется инструкция `vextractf128` для записи упакованных значений с плавающей точкой из регистра `ymm0` в регистр `xmm2`, частями по 128 бит. Можно было бы воспользоваться инструкцией `extractps` для последнего извлечения значений с плавающей точкой и сохранения их в регистрах общего назначения.

Используются новые инструкции с тремя операндами, как показано ниже:

```
vaddps ymm2,ymm0,ymm1
```

Выполняется сложение регистров `ymm1` и `ymm0`, а результат сохраняется в регистре `ymm2`.

Функции вывода просто загружают значения из памяти в `xmm`-регистр, выполняют преобразование чисел с обычной точностью в числа с двойной точностью, когда это необходимо, затем вызывают функцию `printf`.

На рис. 35.2 показан вывод этой программы.

```
jo@ubuntu18:~/Desktop/Book/43 avx_unaligned$ ./avx_unaligned
Single Precision Vector 1:
1.1, 2.1, 3.1, 4.1, 5.1, 6.1, 7.1, 8.1,

Single Precision Vector 2:
1.2, 1.2, 3.2, 4.2, 5.2, 6.2, 7.2, 8.2,

Sum of Single Precision Vector 1 and Vector 2:
2.3, 3.3, 6.3, 8.3, 10.3, 12.3, 14.3, 16.3,

Double Precision Vector 1:
1.1, 2.2, 3.3, 4.4

Double Precision Vector 2:
5.5, 6.6, 7.7, 8.8

Sum of Double Precision Vector 1 and Vector 2:
6.6, 8.8, 11.0, 13.2
jo@ubuntu18:~/Desktop/Book/43 avx_unaligned$
```

Рис. 35.2. Вывод программы `avx_unaligned.asm`

РЕЗЮМЕ

В этой главе вы узнали о том:

- как определить, поддерживается ли AVX конкретным процессором;
- что AVX использует 16 256-битовых `ymm`-регистров;
- что 128-битовые `xmm`-регистры соответствуют младшим частям `ymm`-регистров;
- как извлекать значения из `ymm`-регистров.

Глава 36

Операции с матрицами с использованием AVX

Вместо общего обзора многочисленных инструкций AVX, возможно, вызывающих определенный интерес, рассмотрим некоторые операции с матрицами, использующие AVX. Это длинная глава с несколькими страницами исходного кода, большая часть которого уже знакома читателю, но в этом коде будут представлены некоторые новые инструкции.

В примерах данной главы демонстрируется выполнение операций умножения и обращения (инверсии) матриц. В следующей главе будет показано, как транспонировать матрицу.

ПРИМЕР ИСХОДНОГО КОДА ДЛЯ ОПЕРАЦИЙ С МАТРИЦАМИ

В листинге 36.1 показан исходный код примера.

Листинг 36.1. Программа *matrix4x4.asm*

```
; matrix4x4.asm
extern printf

section .data
    fmt0 db 10,"4x4 DOUBLE PRECISION FLOATING POINT MATRICES",10,0
    fmt1 db 10,"This is matrixA:",10,0
    fmt2 db 10,"This is matrixB:",10,0
    fmt3 db 10,"This is matrixA x matrixB:",10,0
    fmt4 db 10,"This is matrixC:",10,0
    fmt5 db 10,"This is the inverse of matrixC:",10,0
    fmt6 db 10,"Proof: matrixC x inverse =",10,0
    fmt7 db 10,"This is matrixS:",10,0
    fmt8 db 10,"This is the inverse of matrixS:",10,0
    fmt9 db 10,"Proof: matrixS x inverse =",10,0
    fmt10 db 10,"This matrix is singular!",10,10,0
    align 32
    matrixA dq 1., 3., 5., 7.
             dq 9., 11., 13., 15.
             dq 17., 19., 21., 23.
             dq 25., 27., 29., 31.
    matrixB dq 2., 4., 6., 8.
```

```

dq    10., 12., 14., 16.
dq    18., 20., 22., 24.
dq    26., 28., 30., 32.
matrixC dq    2.,      11.,      21.,      37.
dq    3.,      13.,      23.,      41.
dq    5.,      17.,      29.,      43.
dq    7.,      19.,      31.,      47.
matrixS dq    1.,      2.,      3.,      4.
dq    5.,      6.,      7.,      8.
dq    9.,      10.,     11.,     12.
dq    13.,     14.,     15.,     16.

```

```

section .bss
    alignb 32
    product resq 16
    inverse resq 16
section .text
    global main
main:
    push rbp
    mov  rbp, rsp
; Вывод заголовка.
    mov  rdi, fmt0
    call printf
; Вывод матрицы matrixA.
    mov  rdi, fmt1
    call printf
    mov  rsi, matrixA
    call printm4x4
; Вывод матрицы matrixB.
    mov  rdi, fmt2
    call printf
    mov  rsi, matrixB
    call printm4x4
; Вычисление произведения матриц matrixA x matrixB.
    mov  rdi, matrixA
    mov  rsi, matrixB
    mov  rdx, product
    call multi4x4
; Вывод произведения (результата умножения).
    mov  rdi, fmt3
    call printf
    mov  rsi, product
    call printm4x4
; Вывод матрицы matrixC.
    mov  rdi, fmt4
    call printf
    mov  rsi, matrixC
    call printm4x4
; Выполнение обращения (инверсии) матрицы matrixC.
    mov  rdi, matrixC
    mov  rsi, inverse
    call inverse4x4
    cmp  rax, 1
    je   singular
; Вывод обращенной матрицы.

```

```

    mov    rdi,fmt5
    call   printf
    mov    rsi,inverse
    call   printm4x4
; Подтверждение операции умножения matrixC и обращение.
    mov    rsi,matrixC
    mov    rdi,inverse
    mov    rdx,product
    call   multi4x4
; Вывод подтверждения.
    mov    rdi,fmt6
    call   printf
    mov    rsi,product
    call   printm4x4

; Сингулярная (вырожденная) матрица.
; Вывод matrixS.
    mov    rdi,fmt7
    call   printf
    mov    rsi,matrixS
    call   printm4x4
; Выполнение обращения (инверсии) матрицы matrixS.
    mov    rdi,matrixS
    mov    rsi,inverse
    call   inverse4x4
    cmp    rax,1
    je     singular
; Вывод обращенной матрицы.
    mov    rdi,fmt8
    call   printf
    mov    rsi,inverse
    call   printm4x4
; Подтверждение операции умножения matrixS и обращение.
    mov    rsi,matrixS
    mov    rdi,inverse
    mov    rdx,product
    call   multi4x4
; Вывод подтверждения.
    mov    rdi,fmt9
    call   printf
    mov    rsi,product
    call   printm4x4
    jmp    exit
singular:
; Вывод сообщения об ошибке.
    mov    rdi,fmt10
    call   printf
exit:
leave
ret

inverse4x4:
section .data
    align 32
    .identity    dq      1., 0., 0., 0.
                  dq      0., 1., 0., 0.

```

```

        dq      0., 0., 1., 0.
        dq      0., 0., 0., 1.
.minus_mask dq      8000000000000000h
.size      dq      4 ; Матрицы 4x4.
.one       dq      1.0
.two       dq      2.0
.three     dq      3.0
.four      dq      4.0
section .bss
alignb 32
.matrix1 resq 16 ; Промежуточная матрица.
.matrix2 resq 16 ; Промежуточная матрица.
.matrix3 resq 16 ; Промежуточная матрица.
.matrix4 resq 16 ; Промежуточная матрица.
.matrixI resq 16
.mxcsr resd 1 ; Используется для проверки деления на ноль.

section .text
push rbp
mov rbp, rsp
push rsi ; Сохранение адреса обращенной матрицы.
vzeroall ; Очистка всех умм-регистров.

; Вычисление промежуточных матриц.
; Вычисление промежуточной матрицы matrix2.
; rdi содержит адрес исходной матрицы.
mov rsi, rdi
mov rdx, .matrix2
push rdi
call multi4x4
pop rdi

; Вычисление промежуточной матрицы matrix3.
mov rsi, .matrix2
mov rdx, .matrix3
push rdi
call multi4x4
pop rdi

; Вычисление промежуточной матрицы matrix4.
mov rsi, .matrix3
mov rdx, .matrix4
push rdi
call multi4x4
pop rdi

; Вычисление следов матриц.
; Вычисление следа матрицы trace1.
mov rsi, [.size]
call vtrace
movsd xmm8, xmm0 ; След 1 в xmm8.
; Вычисление следа матрицы trace2.
push rdi ; Сохранение адреса исходной матрицы.
mov rdi, .matrix2
mov rsi, [.size]
call vtrace
movsd xmm9, xmm0 ; След 2 в xmm9.

```

```

; Вычисление следа матрицы trace3.
    mov     rdi,.matrix3
    mov     rsi,[.size]
    call    vtrace
    movsd   xmm10,xmm0 ; След 3 в xmm10.
; Вычисление следа матрицы trace4.
    mov     rdi,.matrix4
    mov     rsi,[.size]
    call    vtrace
    movsd   xmm11,xmm0 ; trace 4 in xmm11

; Вычисление коэффициентов.
; Вычисление коэффициента p1.
; p1 = -s1
    vxorpd   xmm12,xmm8,[.minus_mask] ; p1 в xmm12.
; Вычисление коэффициента p2.
; p2 = -1/2 * (p1 * s1 + s2)
    movsd     xmm13,xmm12 ; Копирование p1 в xmm13.
    vfmadd213sd xmm13,xmm8,xmm9 ; xmm13=xmm13*xmm8+xmm9
    vxorpd     xmm13,xmm13,[.minus_mask]
    divsd      xmm13,[.two] ; Деление на 2 и p2 в xmm13.
; Вычисление коэффициента p3.
; p3 = -1/3 * (p2 * s1 + p1 * s2 + s3)
    movsd     xmm14,xmm12 ; Копирование p1 в xmm14.
    vfmadd213sd xmm14,xmm9,xmm10 ; p1*s2+s3;xmm14=xmm14*xmm9+xmm10
    vfmadd231sd xmm14,xmm13,xmm8 ; xmm14+p2*s1;xmm14=xmm14+xmm13*xmm8
    vxorpd     xmm14,xmm14,[.minus_mask]
    divsd      xmm14,[.three] ; p3 в xmm14.
; Вычисление коэффициента p4.
; p4 = -1/4 * (p3 * s1 + p2 * s2 + p1 * s3 + s4)
    movsd     xmm15,xmm12 ; Копирование p1 в xmm15.
    vfmadd213sd xmm15,xmm10,xmm11 ; p1*s3+s4;xmm15=xmm15*xmm10+xmm11
    vfmadd231sd xmm15,xmm13,xmm9 ; xmm15+p2*s2;xmm15=xmm15+xmm13*xmm9
    vfmadd231sd xmm15,xmm14,xmm8 ; xmm15+p3*s1;xmm15=xmm15+xmm14*xmm8
    vxorpd     xmm15,xmm15,[.minus_mask]
    divsd      xmm15,[.four] ; p4 в xmm15.
; Умножение матриц с соответствующим коэффициентом.
    mov     rcx,[.size]
    xor     rax,rax
    vbroadcastsd ymm1,xmm12 ; p1
    vbroadcastsd ymm2,xmm13 ; p2
    vbroadcastsd ymm3,xmm14 ; p3
    pop rdi ; Восстановление адреса исходной матрицы.

.loop1:
    vmovapd     ymm0,[rdi+rax]
    vmulpd      ymm0,ymm0,ymm2
    vmovapd     ymm0,[.matrix1+rax],ymm0
    vmovapd     ymm0,[.matrix2+rax]
    vmulpd      ymm0,ymm0,ymm1
    vmovapd     ymm0,[.matrix2+rax],ymm0
    vmovapd     ymm0,[.identity+rax]
    vmulpd      ymm0,ymm0,ymm3
    vmovapd     ymm0,[.matrixI+rax],ymm0
    add         rax,32
    loop        .loop1

```



```

; Сложение четырех матриц и умножение на -1/p4.
    mov     rcx,[.size]
    xor     rax,rax
; Вычисление -1/p4.
    movsd   xmm0, [.one]
    vdivsd   xmm0,xmm0,xmm15    ; 1/p4
; Проверка деления на ноль.
    stmxcsr  [.mxcsr]
    and     dword[.mxcsr],4
    jnz     .singular

; Нет деления на ноль.
    pop     rsi                ; Восстановление адреса обращенной матрицы.
    vxorpd   xmm0,xmm0,[.minus_mask] ; -1/p4
    vbroadcastsd ymm2,xmm0

; Цикл по строкам.
.loop2:
    ; Сложение строк.
    vmovapd   ymm0,[.matrix1+rax]
    vaddpd     ymm0, ymm0, [.matrix2+rax]
    vaddpd     ymm0, ymm0, [.matrix3+rax]
    vaddpd     ymm0, ymm0, [.matrixI+rax]
    vmulpd     ymm0,ymm0,ymm2    ; Умножение строки на -1/p4.
    vmovapd   [rsi+rax],ymm0
    add       rax,32
    loop      .loop2
    xor       rax,rax ; Возврат 0, ошибок нет.
leave
ret

.singular:
    mov       rax,1 ; Возврат 1, вырожденная (сингулярная) матрица.
leave
ret

;-----
; Вычисление следа матрицы.
vtrace:
push  rbp
mov   rbp,rsi
; Формирование матрицы в памяти.
    vmovapd   ymm0, [rdi]
    vmovapd   ymm1, [rdi+32]
    vmovapd   ymm2, [rdi+64]
    vmovapd   ymm3, [rdi+96]
    vblendpd   ymm0,ymm0,ymm1,0010b
    vblendpd   ymm0,ymm0,ymm2,0100b
    vblendpd   ymm0,ymm0,ymm3,1000b
    vhaddpd    ymm0,ymm0,ymm0
    vpermpd    ymm0,ymm0,00100111b
    haddpd     xmm0,xmm0
leave
ret

;-----

```

```

printm4x4:
section .data
    .fmt db      "%f",9,"%f",9, "%f",9,"%f",10,0
section .text
push    rbp
mov     rbp,rsp
push    rbx      ; Регистр, сохраняемый вызываемой функцией.
push    r15      ; Регистр, сохраняемый вызываемой функцией.
        mov rdi,.fmt
        mov rcx,4
        xor rbx,rbx      ; Счетчик строк.
.loop:
        movsd xmm0, [rsi+rbx]
        movsd xmm1, [rsi+rbx+8]
        movsd xmm2, [rsi+rbx+16]
        movsd xmm3, [rsi+rbx+24]
        mov    rax,4      ; Четыре значения с плавающей точкой.
        push   rcx      ; Регистр, сохраняемый вызывающей функцией.
        push   rsi      ; Регистр, сохраняемый вызывающей функцией.
        push   rdi      ; Регистр, сохраняемый вызывающей функцией.
        ; Выравнивание стека, если необходимо.
        xor    r15,r15
        test   rsp,0xf    ; Последний байт равен 8 (стек не выровнен)?
        setnz  r15b      ; Установить флаг, если стек не выровнен.
        shl    r15,3      ; Умножение на 8.
        sub    rsp,r15    ; Вычитание 0 или 8.
        call   printf
        add    rsp,r15    ; Прибавление 0 или 8 для восстановления rsp.
        pop    rdi
        pop    rsi
        pop    rcx
        add    rbx,32     ; Следующая строка.
        loop   .loop
pop     r15
pop     rbx
leave
ret

;-----
multi4x4:
push    rbp
mov     rbp,rsp
xor     rax,rax
mov     rcx,4
vzeroall      ; Обнуление всех умм-регистров.
.loop:
        vmovapd    ymm0, [rsi]
        vbroadcastsd ymm1,[rdi+rax]
        vfmadd231pd ymm12,ymm1,ymm0
        vbroadcastsd ymm1,[rdi+32+rax]
        vfmadd231pd ymm13,ymm1,ymm0
        vbroadcastsd ymm1,[rdi+64+rax]
        vfmadd231pd ymm14,ymm1,ymm0
        vbroadcastsd ymm1,[rdi+96+rax]
        vfmadd231pd ymm15,ymm1,ymm0
        add    rax,8      ; Один элемент содержит 8 байт, 64 бит.

```

```

    add rsi,32    ; Каждая строка содержит 32 байта, 256 бит.
    loop .loop
; Перемещение результата в память, строка за строкой.
    vmovapd      [rdx], ymm12
    vmovapd      [rdx+32], ymm13
    vmovapd      [rdx+64], ymm14
    vmovapd      [rdx+96], ymm15
    хог          гах,гах    ; Возврат значения.
leave
ret

```

Самая интересная часть кода находится в функциях. Основная функция `main` предназначена для инициализации программы, вызова функций и вывода результатов. В рассматриваемом здесь примере используются матрицы 4×4, содержащие значения с плавающей точкой двойной точности. Обратите внимание на выравнивание матриц по 32-байтовой границе: применяя систему команд AVX, мы используем `ymm`-регистры размером 32 байта. В следующих разделах анализируются отдельные функции этой программы.

Вывод матрицы: PRINTM4x4

Матрица считывается построчно, строки помещаются в четыре `xmm`-регистра, затем содержимое этих регистров записывается в стек. Эти регистры изменяет функция `printf`, поэтому необходимо сохранить их. После этого стек выравнивается по 16-байтовой границе. При обычной операции `rsp` будет выровнен по 8-байтовой границе. Для выравнивания стека по 16-байтовой границе невозможно использовать прием с применением инструкции `and`, описанный в главе 16. Причина в том, что при использовании инструкции `and` неизвестно заранее, будет изменяться `rsp` или нет. Кроме того, необходимо скорректировать указатель стека, потому что после вызова функции `printf` извлекаются ранее сохраненные в стеке регистры. Если `rsp` был изменен, то необходимо вернуть его предыдущее значение перед извлечением, иначе из стека будут восстановлены некорректные значения. Если `rsp` не был изменен, то корректировка указателя стека не нужна.

Для проверки выравнивания стека используется инструкция `test` и значение `0xf`. Если самая последняя шестнадцатеричная цифра в `rsp` равна 0, то указатель стека `rsp` выровнен по 16-байтовой границе. Если последняя цифра отличается от 0, то самый последний полубайт содержит, как минимум, один бит, установленный в 1. Инструкция `test` похожа на инструкцию `and`. Если самый последний полубайт `rsp` содержит один или несколько битов, установленных в 1, то результат сравнения будет ненулевым, поэтому флаг `ZF` будет очищен (сброшен в 0). Инструкция `setnz` (`set if non-zero`) считывает флаг `ZF`, и если он не установлен, то `setnz` записывает значение `0000 0001` в `r15b`. Это означает, что `rsp` не выровнен по 16-байтовой границе, поэтому выполняется вычитание 8 для выравнивания. Для `r15b` три раза выполняется операция сдвига влево, чтобы получить десятичное значение 8 и произвести вычитание. После завершения работы функции `printf` восстанавливается корректный адрес стека посредством прибавления `r15` к `rsp`, т. е. значения 8, если стек выравнивался, или значения 0, если стек не выравнивался. Таким образом, стек возвращается в то состояние, в котором он был перед выравниванием, после чего можно восстанавливать корректные значения регистров.

УМНОЖЕНИЕ МАТРИЦ: MULTI4x4

В рассматриваемом здесь примере исходного кода и в приведенном ниже описании используются следующие две матрицы:

$$A = \begin{bmatrix} 1 & 3 & 5 & 7 \\ 9 & 11 & 13 & 15 \\ 17 & 19 & 21 & 23 \\ 25 & 27 & 29 & 31 \end{bmatrix}; \quad B = \begin{bmatrix} 2 & 4 & 6 & 8 \\ 10 & 12 & 14 & 16 \\ 18 & 20 & 22 & 24 \\ 26 & 28 & 30 & 32 \end{bmatrix}$$

Если вы изучали линейную алгебру, то, вероятно, знаете, что умножение матриц выполняется следующим образом: для получения элемента c_{11} итоговой матрицы $C = AB$ требуется вычисление по формуле:

$$a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31} + a_{14}b_{41}.$$

В рассматриваемом здесь примере вычисление по этой формуле выглядит так:

$$1 \times 2 + 3 \times 10 + 5 \times 18 + 7 \times 26 = 304.$$

Другой элемент c_{32} должен вычисляться, как показано ниже:

$$a_{31}b_{12} + a_{32}b_{22} + a_{33}b_{32} + a_{34}b_{42}.$$

С учетом конкретных значений элементов матриц A и B получаем:

$$17 \times 4 + 19 \times 12 + 21 \times 20 + 23 \times 28 = 1360.$$

Это вполне подходит для вычислений вручную, но мы будем использовать метод, более подходящий для компьютера. Воспользуемся умм-регистрами для хранения вычисляемых сумм и для обновления сумм в последующих циклах. Здесь мы используем всю мощь инструкций AVX.

Сначала производится очистка всех умм-регистров с помощью инструкции `vzeroall`. Затем выполняется цикл четыре раза, по одной итерации для каждой строки матрицы `matrixB`. Строка из четырех значений двойной точности загружается из `matrixB` в регистр `ymm0`. Далее значение из последовательно выбираемых столбцов матрицы `matrixA` распределяется по частям регистра `ymm1`. Регистры служат счетчиком столбцов, при этом значения в столбцах располагаются со смещением 0, 32, 64 и 96. Распределение (`broadcast` – «рассеяние, распыление») означает, что все четыре учетверенных слова (`quadwords`) (размером 8 байт каждое) в регистре будут содержать соответствующее значение. После этого значения в `ymm1` умножаются на значения `ymm0` и суммируются в регистре `ymm12`. Умножение и суммирование выполняется одной инструкцией `vfmadd231pd`, которая означает «`vector fused multiply add packed double`» (умножение и суммирование объединенного вектора из упакованных значений двойной точности). Набор цифр 231 показывает, как используются регистры. Существует несколько вариантов инструкции `vfmadd` (132, 213, 231), кроме того, имеются варианты для

значений с двойной и обычной (одиначной) точностью. Здесь используется вариант 231, который означает умножение второго операнда на третий операнд и сложение с первым операндом, а результат помещается в первый операнд. Эти действия выполняются для каждого значения столбца матрицы `matrixA`, затем итерация продолжается; загружается следующая строка матрицы `matrixB`, и вычисления повторяются.

Пройдите по программе по шагам с помощью отладчика, который вы предпочитаете использовать. Наблюдайте за тем, как в регистрах `ymm12`, `ymm13`, `ymm14` и `ymm15` сохраняются текущие результаты суммирования и формируется итоговое произведение. Вероятнее всего, ваш отладчик будет показывать значения в `ymm`-регистрах в шестнадцатеричном формате и с прямым порядком байтов (от младшего к старшему). Чтобы вам было проще разобраться, в табл. 36.1–36.5 показаны все подробности того, что происходит на каждом шаге цикла.

Таблица 36.1. Начальные значения элементов умножаемых матриц

rdi					rsi				
32 байта					32 байта				
	8 байт	8 байт	8 байт	8 байт		8 байт	8 байт	8 байт	8 байт
0–31	1	3	5	7	0–31	2	4	6	8
32–63	9	11	13	15	32–63	10	12	14	16
64–95	17	19	21	23	64–95	18	20	22	24
96–127	25	27	29	31	96–127	26	28	30	32

Первая итерация цикла показана в табл. 36.2.

Таблица 36.2. Первая итерация цикла

Инструкция	Регистр	Значения			
<code>vmovapd ymm0, [rsi]</code>	<code>ymm0</code>	2	4	6	8
<code>vbroadcastsd ymm1, [rdi+0]</code>	<code>ymm1</code>	1	1	1	1
<code>vfmadd231pd ymm12, ymm1, ymm0</code>	<code>ymm12</code>	2	4	6	8
<code>vbroadcastsd ymm1, [rdi+32+0]</code>	<code>ymm1</code>	9	9	9	9
<code>vfmadd231pd ymm13, ymm1, ymm0</code>	<code>ymm13</code>	18	36	54	72
<code>vbroadcastsd ymm1, [rdi+64+0]</code>	<code>ymm1</code>	17	17	17	17
<code>vfmadd231pd ymm14, ymm1, ymm0</code>	<code>ymm14</code>	34	68	102	136
<code>vbroadcastsd ymm1, [rdi+96+0]</code>	<code>ymm1</code>	25	25	25	25
<code>vfmadd231pd ymm15, ymm1, ymm0</code>	<code>ymm15</code>	50	100	150	200

Вторая итерация цикла показана в табл. 36.3.

Таблица 36.3. Вторая итерация цикла

Инструкция	Регистр	Значения			
vmovapd ymm0, [rsi+32]	ymm0	10	12	14	16
vbroadcastsd ymm1, [rdi+8]	ymm1	3	3	3	3
vfmadd231pd ymm12, ymm1, ymm0	ymm12	32	40	48	56
vbroadcastsd ymm1, [rdi+32+8]	ymm1	11	11	11	11
vfmadd231pd ymm13, ymm1, ymm0	ymm13	128	168	208	248
vbroadcastsd ymm1, [rdi+64+8]	ymm1	19	19	19	19
vfmadd231pd ymm14, ymm1, ymm0	ymm14	224	296	368	440
vbroadcastsd ymm1, [rdi+96+8]	ymm1	27	27	27	27
vfmadd231pd ymm15, ymm1, ymm0	ymm15	320	424	528	632

Третья итерация цикла показана в табл. 36.4.

Таблица 36.4. Третья итерация цикла

Инструкция	Регистр	Значения			
vmovapd ymm0, [rsi+32+32]	ymm0	18	20	22	24
vbroadcastsd ymm1, [rdi+8+8]	ymm1	5	5	5	5
vfmadd231pd ymm12, ymm1, ymm0	ymm12	122	140	158	176
vbroadcastsd ymm1, [rdi+32+8+8]	ymm1	13	13	13	13
vfmadd231pd ymm13, ymm1, ymm0	ymm13	362	428	494	560
vbroadcastsd ymm1, [rdi+64+8+8]	ymm1	21	21	21	21
vfmadd231pd ymm14, ymm1, ymm0	ymm14	602	716	830	944
vbroadcastsd ymm1, [rdi+96+8+8]	ymm1	29	29	29	29
vfmadd231pd ymm15, ymm1, ymm0	ymm15	842	1004	1166	1328

Четвертая (и последняя) итерация цикла показана в табл. 36.5.

Таблица 36.5. Четвертая (последняя) итерация цикла

Инструкция	Регистр	Значения			
vmovapd ymm0, [rsi+32+32+32]	ymm0	26	28	30	32
vbroadcastsd ymm1, [rdi+8+8+8]	ymm1	7	7	7	7
vfmadd231pd ymm12, ymm1, ymm0	ymm12	304	336	368	400
vbroadcastsd ymm1, [rdi+32+8+8+8]	ymm1	15	15	15	15
vfmadd231pd ymm13, ymm1, ymm0	ymm13	752	848	944	1040
vbroadcastsd ymm1, [rdi+64+8+8+8]	ymm1	23	23	23	23
vfmadd231pd ymm14, ymm1, ymm0	ymm14	1200	1360	1520	1680
vbroadcastsd ymm1, [rdi+96+8+8+8]	ymm1	31	31	31	31
vfmadd231pd ymm15, ymm1, ymm0	ymm15	1648	1872	2096	2320

ОБРАЩЕНИЕ МАТРИЦЫ: INVERSE4x4

Математики разработали группу алгоритмов для эффективного вычисления обращения матрицы. Целью этого раздела является не предоставление полной программы обращения матрицы со всеми атрибутами и деталями, а простая демонстрация использования системы инструкций AVX.

В рассматриваемом здесь примере будет использоваться метод, основанный на теореме Гамильтона–Кэли (*Cayley–Hamilton theorem*) о характеристических многочленах. Более подробную информацию о характеристических многочленах можно найти здесь: <http://www.mcs.csueastbay.edu/~malek/Class/Characteristic.pdf>.

Теорема Гамильтона–Кэли

По теореме Гамильтона–Кэли для матрицы A существует следующий характеристический многочлен:

$$A^n + p_1 A^{n-1} + \dots + p_{n-1} A + p_n I = 0,$$

где A^n – это A в степени n . Например, A^3 равно AAA , т. е. матрица A умножается сама на себя три раза. Значения p – это определяемые коэффициенты, I – единичная матрица (матрица тождественности), 0 – нулевая матрица.

Если приведенную выше формулу умножить на A^{-1} , разделить на $-p_n$ и преобразовать (перегруппировать члены), то получим следующую формулу обращения матрицы:

$$(1/-p_n)[A^{n-1} + p_1 A^{n-2} + \dots + p_{n-2} A + p_{n-1} I] = A^{-1}.$$

Таким образом, для вычисления обращения матрицы A необходимо выполнить несколько операций умножения матриц, а также требуется метод определения коэффициентов p .

Для матрицы A размером 4×4 получаем следующую формулу:

$$(1/-p_4)[A^3 + p_1 A^2 + p_2 A + p_3 I] = A^{-1}.$$

Алгоритм Фаддеева–Леверье

Для вычисления коэффициентов p используется алгоритм Фаддеева–Леверье, также описанный здесь: <http://www.mcs.csueastbay.edu/~malek/Class/Characteristic.pdf>. Сначала определяются следы матриц, т. е. суммы элементов главных диагоналей (от верхнего левого угла к нижнему правому). Пусть s_n – след матрицы A^n .

Для матрицы A размером 4×4 вычисляются следы степеней матрицы A , как показано ниже:

s_1 для A ,
 s_2 для AA ,
 s_3 для AAA ,
 s_4 для $AAAA$.

Тогда по алгоритму Фаддеева–Леверье получим следующие формулы:

$$\begin{aligned}
 p_1 &= -s_1, \\
 p_2 &= -1/2(p_1s_1 + s_2), \\
 p_3 &= -1/3(p_2s_1 + p_1s_2 + s_3), \\
 p_4 &= -1/4(p_3s_1 + p_2s_2 + p_1s_3 + s_4).
 \end{aligned}$$

Достаточно просто, не так ли? Если не принимать во внимание более сложные операции умножения матриц для вычисления следов, разумеется.

Исходный код

В предлагаемой функции `inverse4x4` существует отдельный раздел `section .data`, в котором определяется единичная матрица и некоторые переменные, используемые в дальнейшем. Сначала вычисляются степени матрицы и сохраняются в переменных `matrix2`, `matrix3` и `matrix4`. Переменная `matrix1` пока не используется. Затем вызывается функция `vttrace` для вычисления следа каждой (степени) матрицы. В функции `vttrace` сначала формируется требуемая матрица в `ymm`-регистрах (`ymm0`, `ymm1`, `ymm2`, `ymm3`), при этом каждый регистр содержит строку матрицы. Далее применяется инструкция `vblendpd` с четырьмя операндами: два операнда-источника, один целевой операнд и управляющая маска. Необходимо извлечь диагональные элементы из строк 2, 3, 4 и поместить их как упакованные значения в регистр `ymm0` по индексам локаций 1, 2 и 3. В локации с индексом 0 сохраняется элемент следа `ymm0`.

Маска определяет, какие упакованные значения выбираются из операндов-источников. Единица (1) в маске означает, что в этой локации выбирается значение из второго операнда-источника. Ноль (0) в маске означает, что в этой локации выбирается значение из первого операнда-источника. Общая схема этого процесса показана на рис. 36.1, но следует отметить, что на этой схеме значения в регистрах показаны в соответствии с номерами битов в маске. В отладчике, возможно, позиции в регистре `ymm0` будут отображаться как `a1`, `a0`, `a3`, `a2`.

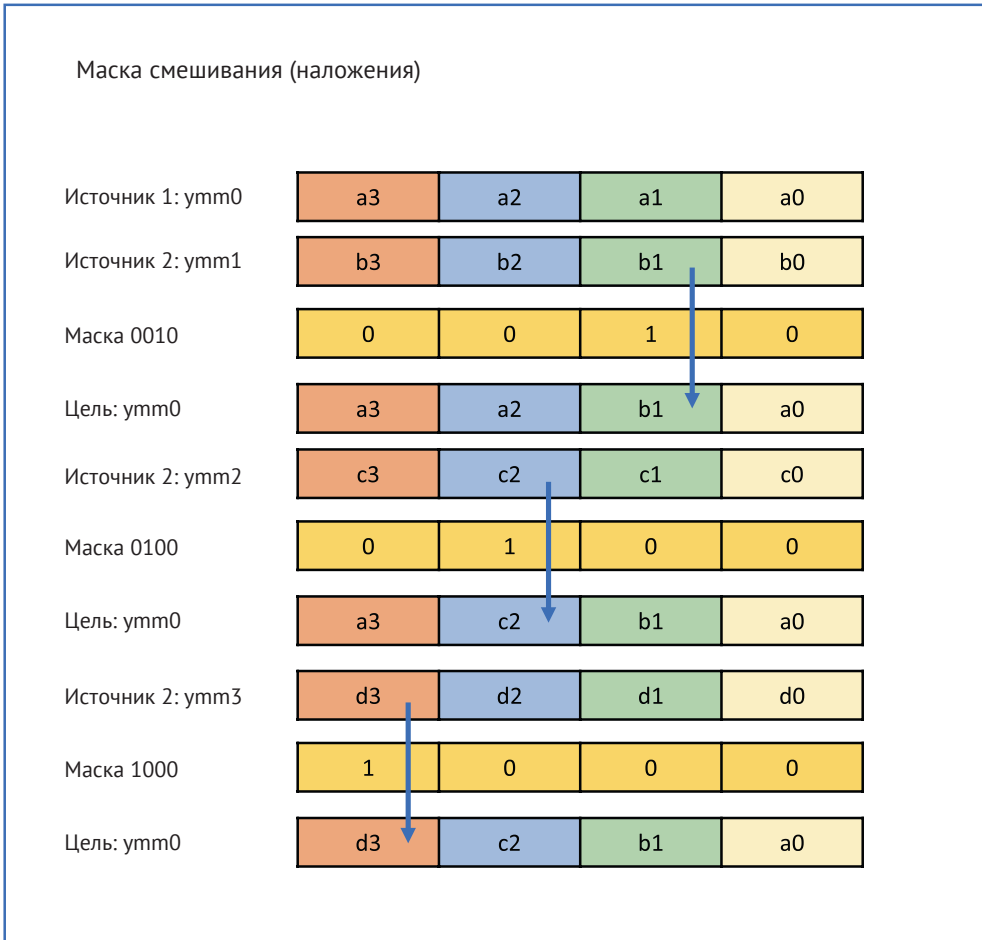


Рис. 36.1. Маска смешивания (наложения)

При вычислении первого следа после операции смешивания (наложения) (blend) регистр `ymm0` содержит элементы следа 2, 13, 29, 47. Это можно проверить с помощью отладчика SASM. Пусть вас не смущает порядок представления значений `ymm0` в отладчике: 13, 2, 47, 29. Теперь нужно вычислить сумму этих значений. Проще всего это можно сделать, извлекая и суммируя требуемые значения, но для наглядной демонстрации здесь будут использоваться инструкции AVX. Применяется инструкция горизонтального сложения `vhaddpd`. После этого регистр `ymm0` содержит значения 15, 15, 76, 76, т. е. сумму двух младших значений и сумму двух старших значений. Затем выполняется инструкция перестановки `vpermq` с маской 00100111. Каждое двухбитовое значение в этой маске выбирает значение в операнде-источнике, схема данной операции показана на рис. 36.2. Теперь младшая часть регистра `ymm0`, т. е. регистр `xmm0` содержит два значения, которые нужно сложить, чтобы получить след матрицы. Выполняется горизонтальное сложение для регистра `xmm0`

с помощью инструкции `haddpd`. Вычисляемые следы сохраняются в регистрах `xmm8`, `xmm9`, `xmm10`, `xmm11` для дальнейшего использования.

Вам не кажется слишком сложным и запутанным такое вычисление следа матрицы? Этот способ был применен только для демонстрации использования нескольких инструкций AVX и соответствующих масок.

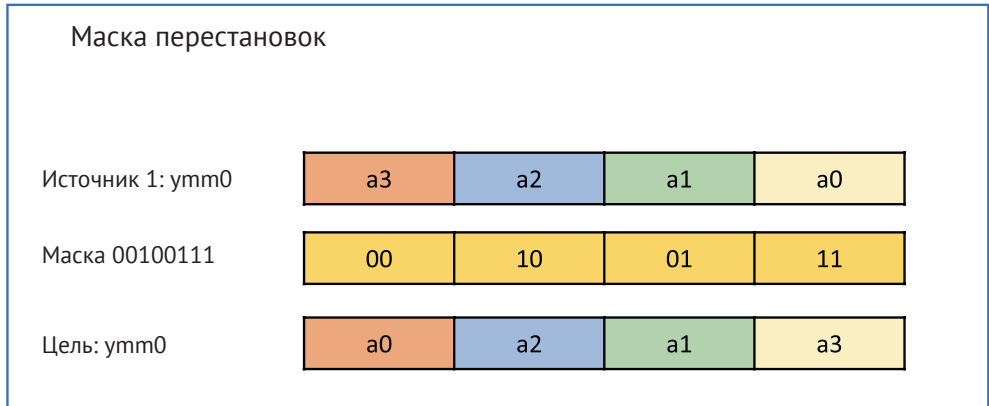


Рис. 36.2. Маска перестановок

После получения всех требуемых следов матриц можно начать вычисление коэффициентов p . Обратите внимание на изменение знака следа с помощью применения отрицательной маски и инструкции `vxorpd`. Инструкции `vmadd213sd` и `vmadd231sd` используются для объединения операций умножения и сложения в одной инструкции. Инструкция `vmadd213sd` означает умножение первого и второго операндов, прибавление третьего операнда и запись результата в первый операнд. Инструкция `vmadd231sd` означает умножение второго и третьего операндов, прибавление первого операнда и запись результата в первый операнд. Полный список аналогичных инструкций см. в руководствах Intel. Внимательно изучите их описание.

После вычисления всех требуемых коэффициентов выполняется скалярное умножение матриц `matrix`, `matrix2`, `matrix3` и `matrixI` с коэффициентами в соответствии с приведенной выше формулой. Результат умножения с использованием `matrix` сохраняется в `matrix1`. Матрица `matrix4` больше не нужна, поэтому из соображений экономии памяти можно использовать пространство для `inverse` как временную память вместо `matrix4`.

Требуется деление на коэффициент p_4 , поэтому необходима проверка p_4 на ненулевое значение. Можно было бы выполнить эту простую операцию раньше, сразу после вычисления p_4 , но в нашем случае нужно показать, как используется регистр `mxcsr`. В регистре `mxcsr` устанавливается в 1 бит деления на ноль и выполняется операция деления с помощью инструкции `vddivsd`. Если после деления третий бит (индекс 2) в регистре `mxcsr` установлен в 1, то произошло деление на ноль, следовательно, матрица является сингулярной (вырожденной) и не может быть обращена. В инструкции `and` применяется десятичное значение 4, соответствующее двоичному значению `0000 0100`, так что очевидно проверяется третий

бит. При обнаружении деления на ноль происходит перенаправление к выходу с 1 в регистре `rax`, т. е. сообщение об ошибке для вызывающей функции.

Если матрица вырожденная (сингулярная), то программа не «падает», потому что по умолчанию деление на ноль маскировано в регистре `mxcsr`. После завершения подробного анализа этого кода удалите комментарии в той части, которая проверяет деление на ноль, и понаблюдайте, что происходит.

Если коэффициент p_4 не равен нулю, то выполняется сложение четырех полученных матриц, потом скалярное умножение результата на $-1/p_4$. Сложение и умножение выполняются в одной итерации цикла. Если все идет, как предполагалось, то мы получаем обращенную матрицу, а вызывающей функции возвращается 0 в регистре `rax`.

На рис. 36.3 показан вывод этой программы.

```
jo@UbuntuDesktop:~/Desktop/linux64/gcc/44 avx_matrix$ make
nasm -f elf64 -g -F dwarf matrix4x4.asm -l matrix4x4.lst
gcc -o matrix4x4 matrix4x4.o -no-pie
jo@UbuntuDesktop:~/Desktop/linux64/gcc/44 avx_matrix$ ./matrix4x4
```

4x4 DOUBLE PRECISION FLOATING POINT MATRICES

This is matrixA:

1.000000	3.000000	5.000000	7.000000
9.000000	11.000000	13.000000	15.000000
17.000000	19.000000	21.000000	23.000000
25.000000	27.000000	29.000000	31.000000

This is matrixB:

2.000000	4.000000	6.000000	8.000000
10.000000	12.000000	14.000000	16.000000
18.000000	20.000000	22.000000	24.000000
26.000000	28.000000	30.000000	32.000000

This is matrixA x matrixB:

304.000000	336.000000	368.000000	400.000000
752.000000	848.000000	944.000000	1040.000000
1200.000000	1360.000000	1520.000000	1680.000000
1648.000000	1872.000000	2096.000000	2320.000000

This is matrixC:

2.000000	11.000000	21.000000	37.000000
3.000000	13.000000	23.000000	41.000000
5.000000	17.000000	29.000000	43.000000
7.000000	19.000000	31.000000	47.000000

This is the inverse of matrixC:

1.000000	-1.000000	-1.000000	1.000000
-2.000000	1.833333	0.944444	-0.888889
1.000000	-1.100000	-0.066667	0.233333
0.000000	0.133333	-0.188889	0.077778

Proof: matrixC x inverse =

1.000000	0.000000	0.000000	0.000000
0.000000	1.000000	0.000000	0.000000
-0.000000	-0.000000	1.000000	-0.000000
0.000000	0.000000	-0.000000	1.000000

This is matrixS:

1.000000	2.000000	3.000000	4.000000
5.000000	6.000000	7.000000	8.000000
9.000000	10.000000	11.000000	12.000000
13.000000	14.000000	15.000000	16.000000

This matrix is singular!

Рис. 36.3. Вывод программы *matrix4x4.asm*

РЕЗЮМЕ

В этой главе вы узнали об:

- операциях с матрицами с использованием инструкций AVX;
- инструкциях AVX с тремя операндами;
- объединенных операциях AVX;
- использовании масок для смешивания (наложения) и перестановок.

Глава 37

Транспонирование матриц

Рассмотрим последнюю полезную операцию с матрицами: транспонирование. В этой главе представлен исходный код двух версий: в первой используются неупакованные данные, во второй применяется метод перемешивания (*shuffling*).

ПРИМЕР ИСХОДНОГО КОДА ДЛЯ ТРАНСПОНИРОВАНИЯ МАТРИЦ

В листинге 37.1 показан исходный код программы.

Листинг 37.1. Программа *transpose4x4.asm*

```
; transpose4x4.asm
extern printf

section .data
    fmt0 db "4x4 DOUBLE PRECISION FLOATING POINT MATRIX TRANSPOSE",10,0
    fmt1 db 10,"This is the matrix:",10,0
    fmt2 db 10,"This is the transpose (unpack):",10,0
    fmt3 db 10,"This is the transpose (shuffle):",10,0
    align 32
    matrix dq 1., 2., 3., 4.
             dq 5., 6., 7., 8.
             dq 9., 10., 11., 12.
             dq 13., 14., 15., 16.

section .bss
    alignb 32
    transpose resd 16

section .text
    global main
main:
    push rbp
    mov rbp, rsp

; Вывод заголовка.
    mov rdi, fmt1
    call printf
```

```

; Вывод матрицы.
    mov     rdi,fmt1
    call    printf
    mov     rsi,matrix
    call    printm4x4

; Вычисление транспонированной матрицы с неупакованными данными.
    mov     rdi, matrix
    mov     rsi, transpose
    call    transpose_unpack_4x4

; Вывод результата.
    mov     rdi, fmt2
    xor     rax,rax
    call    printf
    mov     rsi, transpose
    call    printm4x4

; Вычисление транспонированной матрицы с применением перемешивания.
    mov     rdi, matrix
    mov     rsi, transpose
    call    transpose_shuffle_4x4

; Вывод результата.
    mov     rdi, fmt3
    xor     rax,rax
    call    printf
    mov     rsi, transpose
    call    printm4x4
leave
ret

;-----
transpose_unpack_4x4:
push    rbp
mov     rbp,rsp
; Загрузка матрицы в регистры.
    vmovapd    ymm0,[rdi]      ; 1  2  3  4
    vmovapd    ymm1,[rdi+32]   ; 5  6  7  8
    vmovapd    ymm2,[rdi+64]   ; 9 10 11 12
    vmovapd    ymm3,[rdi+96]   ; 13 14 15 16
; Распаковка.
    vunpcklpd  ymm12,ymm0,ymm1      ; 1  5  3  7
    vunpckhpd  ymm13,ymm0,ymm1      ; 2  6  4  8
    vunpcklpd  ymm14,ymm2,ymm3      ; 9 13 11 15
    vunpckhpd  ymm15,ymm2,ymm3      ; 10 14 12 16
; Перестановка.
    vperm2f128 ymm0,ymm12,ymm14, 00100000b ; 1  5  9 13
    vperm2f128 ymm1,ymm13,ymm15, 00100000b ; 2  6 10 14
    vperm2f128 ymm2,ymm12,ymm14, 00110001b ; 3  7 11 15
    vperm2f128 ymm3,ymm13,ymm15, 00110001b ; 4  8 12 16
; Запись в память.
    vmovapd    [rsi],    ymm0
    vmovapd    [rsi+32], ymm1
    vmovapd    [rsi+64], ymm2
    vmovapd    [rsi+96], ymm3
leave
ret

```

```

;-----
transpose_shuffle_4x4:
push rbp
mov rbp, rsp
; Загрузка матрицы в регистры.
vmovapd ymm0, [rdi] ; 1 2 3 4
vmovapd ymm1, [rdi+32] ; 5 6 7 8
vmovapd ymm2, [rdi+64] ; 9 10 11 12
vmovapd ymm3, [rdi+96] ; 13 14 15 16
; Перемешивание.
vshufpd ymm12, ymm0, ymm1, 0000b ; 1 5 3 7
vshufpd ymm13, ymm0, ymm1, 1111b ; 2 6 4 8
vshufpd ymm14, ymm2, ymm3, 0000b ; 9 13 11 15
vshufpd ymm15, ymm2, ymm3, 1111b ; 10 14 12 16
; Перестановка.
vperm2f128 ymm0, ymm12, ymm14, 00100000b ; 1 5 9 13
vperm2f128 ymm1, ymm13, ymm15, 00100000b ; 2 6 10 14
vperm2f128 ymm2, ymm12, ymm14, 00110001b ; 3 7 11 15
vperm2f128 ymm3, ymm13, ymm15, 00110001b ; 4 8 12 16
; Запись в память.
vmovapd [rsi], ymm0
vmovapd [rsi+32], ymm1
vmovapd [rsi+64], ymm2
vmovapd [rsi+96], ymm3
leave
ret

;-----
printm4x4:
section .data
.fmt db "%.f", 9, "%.f", 9, "%.f", 9, "%.f", 10, 0
section .text
push rbp
mov rbp, rsp
push rbx ; Регистр, сохраняемый вызываемой функцией.
push r15 ; Регистр, сохраняемый вызываемой функцией.
mov rdi, .fmt
mov rcx, 4
xor rbx, rbx ; Счетчик строк.
.loop:
movsd xmm0, [rsi+rbx]
movsd xmm1, [rsi+rbx+8]
movsd xmm2, [rsi+rbx+16]
movsd xmm3, [rsi+rbx+24]
mov rcx, 4 ; Четыре числа с плавающей точкой.
push rcx ; Регистр, сохраняемый вызывающей функцией.
push rsi ; Регистр, сохраняемый вызывающей функцией.
push rdi ; Регистр, сохраняемый вызывающей функцией.
; Требуется выравнивание стека.
xor r15, r15
test rsp, 0fh ; Последний байт равен 8 (стек не выровнен)?
setnz r15b ; Установить флаг, если стек не выровнен.
shl r15, 3 ; Умножение на 8.
sub rsp, r15 ; Вычитание 0 или 8.
call printf
add rsp, r15 ; Прибавление 0 или 8.

```

```

        pop    rdi
        pop    rsi
        pop    rcx
        add    rbx,32      ; Следующая строка.
        loop   .loop
pop r15
pop rbx
leave
ret

```

На рис. 37.1 показан вывод этой программы.

```

jo@ubuntu18:~/Desktop/Book/45 avx_transpose$ ./transpose4x4

This is the matrix:

This is the matrix:
1      2      3      4
5      6      7      8
9      10     11     12
13     14     15     16

This is the transpose (unpack):
1      5      9      13
2      6      10     14
3      7      11     15
4      8      12     16

This is the transpose (shuffle):
1      5      9      13
2      6      10     14
3      7      11     15
4      8      12     16
jo@ubuntu18:~/Desktop/Book/45 avx_transpose$

```

Рис. 37.1. Вывод программы *transpose.asm*

ВЕРСИЯ С ИСПОЛЬЗОВАНИЕМ НЕУПАКОВАННЫХ ДАННЫХ

Сначала необходимо сделать замечание относительно прямого порядка байтов и упакованных значений в *ymm*-регистрах. Если в примере имеются строки 1, 2, 3, 4, то в формате с прямым порядком байтов (от младшего к старшему) они должны выглядеть так: 4, 3, 2, 1. Но поскольку в рассматриваемом здесь примере в *ymm*-регистре хранятся упакованные значения, содержимое этого *ymm*-регистра в *SASM* будет представлено следующим образом: 2, 1, 4, 3. Это можно проверить в любом отладчике. Такое представление данных может сбивать с толку при отладке программы. В дальнейшем всюду будет использоваться формат с прямым порядком байтов 4, 3, 2, 1, а не «формат отладчика» 2, 1, 4, 3.

С учетом приведенного выше замечания после загрузки матрицы в *ymm*-регистры расположение данных в этих регистрах будет таким, как показано в табл. 37.1 (в круглых скобках приведены значения примера).

Таблица 37.1. Содержимое уmm-регистров после загрузки матрицы

ymm0	Старшее слово qword2 (4)	Младшее слово qword2 (3)	Старшее слово qword1 (2)	Младшее слово qword1 (1)
ymm1	Старшее слово qword4 (8)	Младшее слово qword4 (7)	Старшее слово qword3 (6)	Младшее слово qword3 (5)
...				

Инструкция `vunpsklpd ymm12,ymm0,ymm1` принимает первые младшие учетверенные слова из операндов 2 и 3 и сохраняет их в операнде 1, затем таким же способом принимает вторые (самые) младшие учетверенные слова для получения результата, показанного в табл. 37.2.

Таблица 37.2. Распаковка данных из младших учетверенных слов

ymm12	Младшее слово qword4 (7)	Младшее слово qword2 (3)	Младшее слово qword3 (5)	Младшее слово qword1 (1)
-------	-----------------------------	-----------------------------	-----------------------------	-----------------------------

Аналогичным образом инструкция `vunpskhrpd ymm13,ymm0,ymm1` принимает старшие учетверенные слова из операндов 2 и 3, затем сохраняет их в операнде 1 таким же способом (см. табл. 37.3).

Таблица 37.3. Распаковка данных из старших учетверенных слов

ymm13	Старшее слово qword4 (8)	Старшее слово qword2 (4)	Старшее слово qword3 (6)	Старшее слово qword1 (2)
-------	-----------------------------	-----------------------------	-----------------------------	-----------------------------

Цель этого метода распаковки данных – замена пар в столбцах на пары в строках. Например, $\begin{bmatrix} 1 \\ 5 \end{bmatrix}$ превращается в [1, 5].

После распаковки уmm-регистры выглядят так, как показано в табл. 37.4 (формат с прямым порядком байтов).

Таблица 37.4. Содержимое уmm-регистров после распаковки

ymm12	7	3	5	1
ymm13	8	4	6	2
ymm14	15	11	13	9
ymm15	16	12	14	10

Вместо формата с прямым порядком байтов (от младшего к старшему) в табл. 37.5 показан удобный для чтения человеком формат.

Таблица 37.5. Матрица в формате, удобном для чтения человеком

1	5	3	7
2	6	4	8
9	13	11	15
10	14	12	16

Теперь необходимо переставить значения в строках, чтобы получить их правильный порядок. В формате с прямым порядком байтов требуется получить результат, показанный в табл. 37.6.

Таблица 37.6. Матрица после перестановки в правильном порядке

13	9	5	1
14	10	6	2
15	11	7	3
16	12	8	4

Возможно, вы заметили, что два младших значения в регистрах `ymm12` и `ymm13` находятся на правильных позициях. Точно так же два старших значения в регистрах `ymm14` и `ymm15` расположены правильно.

Необходимо переместить два младших значения регистра `ymm14` в позиции старших значений регистра `ymm12`, а два младших значения регистра `ymm15` в позиции старших значений регистра `ymm13`.

Два старших значения из `ymm12` должны переместиться в позиции младших значений `ymm14`, а два старших значения из `ymm13` – в позиции младших значений `ymm15`.

Требуется выполнение операции, называемой перестановкой (*permutation*). С помощью инструкции `vperm2f128` можно выполнять перестановку двух значений (128 бит). Для управления перестановкой применяется маска: например, маска `00110001` читается, начиная с младших битов. Ниже приведено описание смысла маски с учетом того, что индексация начинается с 0:

- 01 – взять 128-битовое старшее поле из источника 1 и поместить его в позицию 0 цели;
- 00 – эти биты имеют особый смысл; см. описание ниже;
- 11 – взять 128-битовое старшее поле из источника 2 и поместить его в позицию 128 цели;
- 00 – эти биты имеют особый смысл; см. описание ниже.

Здесь также используется формат с прямым порядком байтов (4, 3, 2, 1) и не учитывается порядок, в котором эти значения хранятся в `ymm`-регистрах.

Таким образом, в действительности два 128-битовых поля из двух источников нумеруются последовательно:

- младшее поле источника 1 = 00;
- старшее поле источника 1 = 01;
- младшее поле источника 2 = 10;
- старшее поле источника 2 = 11.

Упомянутый выше «особый смысл» заключается в следующем: если устанавливается в 1 третий бит (индекс 3) в маске, то младшее поле цели будет обнулено, а если устанавливается в 1 седьмой бит (индекс 7) в маске, то старшее поле цели будет обнулено.

Второй, третий, шестой и седьмой биты здесь не используются. В большинстве случаев можно читать маску 00110001 как 00110001.

Именно это и происходит в программе:

vrerm2f128 умм0, умм12, умм14, 00100000b

Здесь используется маска 00100000.

- Младшие биты 00 означают: взять младшее поле умм12 (5, 1) и поместить его в младшее поле умм0.
- Старшие биты 10 означают: взять младшее поле умм14 (13, 9) и поместить его в старшее поле умм0.

Таблица 37.7. умм-регистры после первой перестановки

Регистр		Значения		
умм12	7	3	5	1
умм14	15	11	13	9
умм0	13	9	5	1

Теперь регистр умм0 содержит завершенную строку. Далее формируется следующая строка.

vrerm2f128 умм1, умм13, умм15, 00100000b

Здесь используется маска 00100000.

- Младшие биты 00 означают: взять младшее поле умм13 (6, 2) и поместить его в младшее поле умм1.
- Старшие биты 10 означают: взять младшее поле умм15 (14, 10) и поместить его в старшее поле умм1.

Таблица 37.8. умм-регистры после второй перестановки

Регистр		Значения		
умм13	8	4	6	2
умм15	16	12	14	10
умм1	14	10	6	2

Теперь регистр `ymm1` содержит завершенную строку. Формируется следующая строка:

`vperm2f128 ymm2, ymm12, ymm14, 00110001b`

Здесь используется маска **00110001**.

- Младшие биты 01 означают: взять старшее поле `ymm12` (7, 3) и поместить его в младшее поле `ymm2`.
- Старшие биты 11 означают: взять старшее поле `ymm14` (15, 11) и поместить его в старшее поле `ymm2`.

Таблица 37.9. `ymm`-регистры после третьей перестановки

Регистр		Значения		
<code>ymm12</code>	7	3	5	1
<code>ymm14</code>	15	11	13	9
<code>ymm2</code>	15	11	7	3

Теперь регистр `ymm2` содержит завершенную строку. Формируется последняя строка:

`vperm2f128 ymm3, ymm13, ymm15, 00110001b`

Здесь используется маска **00110001**.

- Младшие биты 01 означают: взять старшее поле `ymm13` (8, 4) и поместить его в младшее поле `ymm3`.
- Старшие биты 11 означают: взять старшее поле `ymm15` (16, 12) и поместить его в старшее поле `ymm3`.

Таблица 37.10. `ymm`-регистры после четвертой перестановки

Регистр		Значения		
<code>ymm13</code>	8	4	6	2
<code>ymm15</code>	16	12	14	10
<code>ymm3</code>	16	12	8	4

Перестановка выполнена. Осталось лишь скопировать строки из `ymm`-регистров в память в правильном порядке.

ВЕРСИЯ С ПРИМЕНЕНИЕМ ПЕРЕМЕШИВАНИЯ

Мы уже использовали инструкцию перемешивания `rshufd` в главе 33. Здесь будет применяться инструкция `vshufpd`, для которой также требуется маска для управления операцией перемешивания. Не перепутайте – для инструкции `rshufd` использовалась 8-битовая маска. В масках, которые применяются здесь, значимыми являются только 4 бита.

В этой версии примера также используется формат с прямым порядком байтов (4, 3, 2, 1) и не учитывается порядок хранения упакованных значений в умм-регистрах. Это забота процессора.

В табл. 37.11 приведены примеры, соответствующие следующему описанию. Два младших бита в управляющей маске определяют, какие упакованные значения записываются в две младшие позиции цели. Два старших бита в управляющей маске определяют, какие упакованные значения записываются в две старшие позиции цели. Биты 0 и 2 определяют, какое значение берется из источника 1, биты 1 и 3 определяют, какое значение берется из источника 2.

Таблица 37.11. Примеры значений битов управляющей маски

Выбор из двух старших значений в источнике 2	Выбор из двух старших значений в источнике 1	Выбор из двух младших значений в источнике 2	Выбор из двух младших значений в источнике 1
0 = младшее значение источника 2	0 = младшее значение источника 1	0 = младшее значение источника 2	0 = младшее значение источника 1
1 = старшее значение источника 2	1 = старшее значение источника 1	1 = старшее значение источника 2	1 = старшее значение источника 1

Два младших значения в каждом источнике никогда не могут оказаться в старших позициях цели, а два старших значения в каждом источнике никогда не могут оказаться в младших позициях цели. На рис. 37.2 показаны схемы для нескольких примеров управляющих масок.

Маска, управляющая перемешиванием

Источник 1: умм0	a3	a2	a1	a0
Источник 2: умм1	b3	b2	b1	b0
Маска 0000	0	0	0	0
Цель: умм3	b2	a2	b0	a0
Источник 1: умм0	a3	a2	a1	a0
Источник 2: умм1	b3	b2	b1	b0
Маска 1111	1	1	1	1
Цель: умм3	b3	a3	b1	a1
Источник 1: умм0	a3	a2	a1	a0
Источник 2: умм1	b3	b2	b1	b0
Маска 0110	0	1	1	0
	b2	a3	b1	a0
Источник 1: умм0	a3	a2	a1	a0
Источник 2: умм1	b3	b2	b1	b0
Маска 0011	0	0	1	1
	b2	a2	b1	a1

Рис. 37.2. Примеры масок, управляющих перемешиванием

В табл. 37.12–37.15 показано, как это работает в рассматриваемом здесь примере программы.

vshufpd умм12,умм0,умм1, 0000b

Таблица 37.12

Регистр		Значения		
умм0	4	3	2	1
умм1	8	7	6	5
умм12	Младший старший умм1	Младший старший умм0	Младший старший умм1	Младший старший умм0
	7	3	5	1

vshufpd умм13,умм0,умм1, 1111b

Таблица 37.13

Регистр		Значения		
умм0	4	3	2	1
умм1	8	7	6	5
умм13	Старший старший умм1	Старший старший умм0	Старший младший умм1	Старший младший умм0
	8	4	6	2

vshufpd умм14,умм2,умм3, 0000b

Таблица 37.14

Регистр	Значения			
умм2	12	11	10	9
умм3	16	15	14	13
умм14	Младший старший умм3	Младший старший умм2	Младший младший умм3	Младший младший умм2
	15	11	14	9

Последний пример:

vshufpd умм15,умм2,умм3, 1111b

Таблица 37.15

Регистр		Значения		
умм2	12	11	10	9
умм3	16	15	14	13
умм15	Старший старший умм3	Старший старший умм2	Старший младший умм3	Старший младший умм2
	16	12	14	10

После применения маски управления смешиванием мы получаем восемь пар значений в `umt`-регистрах. Регистры выбраны так, чтобы получать те же промежуточные результаты, что и в версии с использованием неупакованных данных. После этого пары значений должны быть перемещены в правильные позиции, чтобы сформировать транспонированную матрицу. Это делается точно таким же способом, что и в версии с использованием неупакованных данных: перестановкой полей (блоков) размером 128 бит с помощью инструкции `vperm2f128`.

РЕЗЮМЕ

В этой главе вы узнали:

- о двух способах транспонирования матриц;
- об использовании инструкций перемешивания, распаковки и перестановки;
- о существовании различных масок для перемешивания, распаковки и перестановки.

Глава 38

Оптимизация производительности

Вы, вероятно, согласитесь, что подавляющее большинство инструкций AVX нельзя назвать интуитивно понятными, особенно разнообразные схемы масок, которые делают исходный код весьма трудным для чтения и понимания. Более того, битовые маски иногда записываются в шестнадцатеричном формате, поэтому приходится сначала преобразовывать их в двоичный формат, чтобы понять, что они означают.

В этой главе будет продемонстрировано, как применение инструкций AVX может существенно улучшить производительность, так что трудозатраты на освоение и практическое использование AVX окупаются во многих случаях. Интересный документ по тестированию производительности кода можно найти здесь: <https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-32-ia-64-benchmark-code-execution-paper.pdf>.

В примерах этой главы будет использоваться методика измерения производительности, представленная в вышеуказанном документе.

Производительность вычисления транспонированной матрицы

В примере исходного кода, показанного в листинге 38.1, применяются два метода вычисления транспонированной матрицы: один с использованием «классических» инструкций ассемблера, во втором применяются инструкции AVX. Также добавлен код для измерения времени выполнения обоих алгоритмов.

Листинг 38.1. Программа *transpose.asm*

```
; transpose.asm
extern printf

section .data
    fmt0 db "4x4 DOUBLE PRECISION FLOATING POINT MATRIX TRANSPOSE",10,0
    fmt1 db 10,"This is the matrix:",10,0
    fmt2 db 10,"This is the transpose (sequential version): ",10,0
    fmt3 db 10,"This is the transpose (AVX version): ",10,0
    fmt4 db 10,"Number of loops: %d",10,0
```

```

fmt5    db    "Sequential version elapsed cycles: %d",10,0
fmt6    db    "AVX Shuffle version elapsed cycles: %d",10,0
align   32
matrix  dq     1.,    2.,    3.,    4.
         dq     5.,    6.,    7.,    8.
         dq     9.,   10.,   11.,   12.
         dq    13.,   14.,   15.,   16.

loops   dq    10000

section .bss
alignb   32
transpose resq    16
bahi_cy  resq    1 ; Таймеры для версии avx.
balo_cy  resq    1
eahi_cy  resq    1
ealo_cy  resq    1
bshi_cy  resq    1 ; Таймеры для версии с последоват. вычислениями.
bslo_cy  resq    1
eshi_cy  resq    1
eslo_cy  resq    1

section .text
global main
main:
push    rbp
mov     rbp,rbp
; Вывод заголовка.
mov     rdi,fmt0
call    printf
; Вывод матрицы.
mov     rdi,fmt1
call    printf
mov     rsi,matrix
call    printm4x4

; ВЕРСИЯ С ПОСЛЕДОВАТЕЛЬНЫМИ ВЫЧИСЛЕНИЯМИ.
; Вычисление транспонированной матрицы.
mov     rdi, matrix
mov     rsi, transpose
mov     rdx, [loops]
; Начало отсчета циклов процессора.
cpuid
rdtsc
mov     [bshi_cy],edx
mov     [bslo_cy],eax
call    seq_transpose
; Окончание отсчета циклов процессора.
rdtscp
mov     [eshi_cy],edx
mov     [eslo_cy],eax
cpuid
; Вывод результата.
mov     rdi,fmt2
call    printf
mov     rsi,transpose
call    printm4x4

```

```
; ВЕРСИЯ AVX.
; Вычисление транспонированной матрицы.
    mov rdi, matrix
    mov rsi, transpose
    mov rdx, [loops]
; Начало отсчета циклов процессора.
    cpuid
    rdtsc
    mov [bahi_cy],edx
    mov [balo_cy],eax
    call AVX_transpose
; Окончание отсчета циклов процессора.
    rdtscp
    mov [eahi_cy],edx
    mov [ealo_cy],eax
    cpuid

; Вывод результата.
    mov rdi,fmt3
    call printf
    mov rsi,transpose
    call printm4x4
; Вывод программных циклов.
    mov rdi,fmt4
    mov rsi,[loops]
    call printf
; Вывод счетчика циклов процессора.
; Циклы версии с последовательными вычислениями.
    mov rdx,[eslo_cy]
    mov rsi,[eshi_cy]
    shl rsi,32
    or rsi,rdx ; rsi содержит время окончания.
    mov r8,[bslo_cy]
    mov r9,[bshi_cy]
    shl r9,32
    or r9,r8 ; r9 содержит время начала.
    sub rsi,r9 ; rsi содержит прошедшее время (интервал).
; Вывод результата измерения времени.
    mov rdi,fmt5
    call printf
; Счетчик циклов процессора для версии AVX blend.
    mov rdx,[ealo_cy]
    mov rsi,[eahi_cy]
    shl rsi,32
    or rsi,rdx ; rsi содержит время окончания.
    mov r8,[balo_cy]
    mov r9,[bahi_cy]
    shl r9,32
    or r9,r8 ; r9 содержит время начала.
    sub rsi,r9 ; rsi содержит прошедшее время (интервал).
; Вывод результата измерения времени.
    mov rdi,fmt6
    call printf
leave
ret
```

```

;-----
seq_transpose:
push    rbp
mov     rbp, rsp
.loopx:                                ; Количество циклов.
    pxor    xmm0, xmm0
    xor     r10, r10
    xor     rax, rax
    mov     r12, 4
    .loopo:
        push rcx
        mov  r13, 4
        .loopi:
            movsd xmm0, [rdi+r10]
            movsd [rsi+rax], xmm0
            add     r10, 8
            add     rax, 32
            dec     r13
            jnz     .loopi
            add     rax, 8
            xor     rax, 100000000b    ; rax - 128
            inc     rbx
            dec     r12
            jnz     .loopo
        dec rdx
    jnz .loopx
leave
ret

;-----
AVX_transpose:
push    rbp
mov     rbp, rsp
.loopx:                                ; Количество циклов.
; Загрузка матрицы в регистры.
    vmovapd    ymm0, [rdi]    ; 1  2  3  4
    vmovapd    ymm1, [rdi+32] ; 5  6  7  8
    vmovapd    ymm2, [rdi+64] ; 9 10 11 12
    vmovapd    ymm3, [rdi+96] ; 13 14 15 16
; Перемешивание.
    vshufpd    ymm12, ymm0, ymm1, 0000b    ; 1  5  3  7
    vshufpd    ymm13, ymm0, ymm1, 1111b    ; 2  6  4  8
    vshufpd    ymm14, ymm2, ymm3, 0000b    ; 9 13 11 15
    vshufpd    ymm15, ymm2, ymm3, 1111b    ; 10 14 12 16
; Перестановка.
    vperm2f128 ymm0, ymm12, ymm14, 00100000b ; 1  5  9 13
    vperm2f128 ymm1, ymm13, ymm15, 00100000b ; 2  6 10 14
    vperm2f128 ymm2, ymm12, ymm14, 00110001b ; 3  7 11 15
    vperm2f128 ymm3, ymm13, ymm15, 00110001b ; 4  8 12 16
; Запись в память.
    vmovapd    [rsi], ymm0
    vmovapd    [rsi+32], ymm1
    vmovapd    [rsi+64], ymm2
    vmovapd    [rsi+96], ymm3
    dec rdx
    jnz .loopx

```

```

leave
ret

;-----
printm4x4:
section .data
    .fmt db    "%f",9,"%f",9, "%f",9,"%f",10,0
section .text
push rbp
mov rbp,rsp
push rbx                ; Регистр, сохраняемый вызываемой функцией.
push r15                ; Регистр, сохраняемый вызываемой функцией.
mov     rdi,.fmt
mov     rcx,4
xor     rbx,rbx          ; Счетчик строк.
.loop:
movsd xmm0, [rsi+rbx]
movsd xmm1, [rsi+rbx+8]
movsd xmm2, [rsi+rbx+16]
movsd xmm3, [rsi+rbx+24]
mov     rax,4            ; Четыре числа с плавающей точкой.
push    rcx              ; Регистр, сохраняемый вызываемой функцией.
push    rsi              ; Регистр, сохраняемый вызываемой функцией.
push    rdi              ; Регистр, сохраняемый вызываемой функцией.
; Выравнивание стека, если необходимо.
xor     r15,r15
test    rsp,0fh          ; Последний байт равен 8 (стек не выровнен)?
setnz   r15b             ; Установить флаг, если стек не выровнен.
shl     r15,3            ; Умножение на 8.
sub     rsp,r15          ; Вычитание 0 или 8.
call    printf
add     rsp,r15          ; Прибавление 0 или 8.
pop     rdi
pop     rsi
pop     rcx
add     rbx,32           ; Следующая строка.
loop    .loop
pop r15
pop rbx
leave
ret

```

Перед вызовом функции транспонирования матрицы начинается процесс измерения времени. Современные процессоры поддерживают внеочередное исполнение кода, результатом которого может стать выполнение «рабочих» инструкций в неподходящий момент: перед началом процесса измерения времени или уже после завершения этого процесса. Чтобы избежать этого, необходимо использовать «сериализацию» инструкций, которая гарантирует, что инструкции контроля времени измеряют тот интервал выполнения, который нам нужен. Более подробное описание этой методики можно найти в документе, ссылка на который приведена выше. Одной из инструкций, которую можно применить для сериализации, является `cruid`. Перед пуском таймера инструкцией `rdtsc` выполняется инструкция `cruid`. Инструкция `rdtsc` используется для записи метки времени как счетчик «младших циклов» в регистр `eax`.

и «старших циклов» в регистр `edx`, потом эти значения сохраняются в памяти. Инструкция `gdtsc` использует эти два регистра по историческим причинам: в 32-битовых процессорах одного регистра не хватало для хранения показаний таймера. Один 32-битовый регистр содержал младшую часть значения счетчика времени, в другом регистре сохранялась старшая часть этого значения. После записи начальных показаний таймера выполняется «измеряемый» код, а инструкция `gdtscr` используется для останова процесса измерений. Итоговые значения счетчиков «старших циклов» и «младших циклов» еще раз сохраняются в памяти, и снова выполняется инструкция `cruid` для полной уверенности в том, что процессор не будет выполнять какие-либо отложенные инструкции.

В рассматриваемом здесь примере используется рабочая среда 64-битового процессора, поэтому выполняется сдвиг влево на 32 разряда старших значений метки времени, затем выполняется инструкция `horg` для старшего и младшего значений метки времени, чтобы получить полные метки времени в 64-битовом регистре. Разность между начальными и конечными значениями счетчика равна количеству использованных циклов процессора.

Функция `seq_transpose` использует «классические» инструкции, а функция `AVX_transpose` – это функция `transpose_shuffle4x4` из предыдущей главы. Эти функции выполняются многократно – столько раз, сколько задано в переменной `loops`.

На рис. 38.1 показан вывод данной программы.

```
jo@UbuntuDesktop:~/Desktop/linux64/gcc/46 performance1$ make
nasm -f elf64 -g -F dwarf transpose.asm -l transpose.lst
gcc -o transpose transpose.o -no-pie
jo@UbuntuDesktop:~/Desktop/linux64/gcc/46 performance1$ ./transpose
4x4 DOUBLE PRECISION FLOATING POINT MATRIX TRANSPOSE

This is the matrix:
1.000000      2.000000      3.000000      4.000000
5.000000      6.000000      7.000000      8.000000
9.000000     10.000000     11.000000     12.000000
13.000000     14.000000     15.000000     16.000000

This is the transpose (sequential version):
1.000000      5.000000      9.000000     13.000000
2.000000      6.000000     10.000000     14.000000
3.000000      7.000000     11.000000     15.000000
4.000000      8.000000     12.000000     16.000000

This is the transpose (AVX version):
1.000000      5.000000      9.000000     13.000000
2.000000      6.000000     10.000000     14.000000
3.000000      7.000000     11.000000     15.000000
4.000000      8.000000     12.000000     16.000000

Number of loops: 10000
Sequential version elapsed cycles: 132687
AVX Shuffle version elapsed cycles: 12466
jo@UbuntuDesktop:~/Desktop/linux64/gcc/46 performance1$
```

Рис. 38.1. Вывод программы *transpose.asm*

Здесь можно видеть, что обработка с использованием инструкций AVX выполняется заметно быстрее.

В комплекте руководств Intel есть отдельный том, посвященный оптимизации кода: <https://software.intel.com/sites/default/files/managed/9e/bc/64-ia-32-architectures-optimization-manual.pdf>.

Этот том руководства содержит большой объем полезной информации о методах улучшения производительности ассемблерного кода. Ищите фразу «*handling port 5 pressure*» (упомянутую в главе 14). В этом разделе вы найдете несколько версий алгоритма транспонирования матриц размером 8×8 , а также оценку воздействия различных инструкций на производительность. В предыдущей главе демонстрировались два метода транспонирования матриц: с использованием операций распаковки и перемешивания данных. В руководствах Intel эта тема рассматривается более глубоко и подробно, и если производительность важна для вас, то этот том станет для вас «кладезем бесценной информации».

Производительность вычисления следа матрицы

Пример в листинге 38.2 показывает, что инструкции AVX не всегда обеспечивают более высокую скорость, чем «классические» ассемблерные инструкции. В этом примере вычисляется след матрицы размером 8×8 .

Листинг 38.2. Программа *trace.asm*

```
; trace.asm
extern printf
section .data
    fmt0 db "8x8 SINGLE PRECISION FLOATING POINT MATRIX TRACE",10,0
    fmt1 db 10,"This is the matrix:",10,0
    fmt2 db 10,"This is the trace (sequential version): %f",10,0
    fmt5 db "This is the trace (AVX blend version): %f",10,0
    fmt6 db 10,"This is the tranpose: ",10,0
    fmt30 db "Sequential version elapsed cycles: %u",10,0
    fmt31 db "AVX blend version elapsed cycles: %d",10,10,0
    fmt4 db 10,"Number of loops: %d",10,0

    align 32
    matrix dd 1., 2., 3., 4., 5., 6., 7., 8.
             dd 9., 10., 11., 12., 13., 14., 15., 16.
             dd 17., 18., 19., 20., 21., 22., 23., 24.
             dd 25., 26., 27., 28., 29., 30., 31., 32.
             dd 33., 34., 35., 36., 37., 38., 39., 40.
             dd 41., 42., 43., 44., 45., 46., 47., 48.
             dd 49., 50., 51., 52., 53., 54., 55., 56.
             dd 57., 58., 59., 60., 61., 62., 63., 64.

    loops dq 1000
    permpps dd 0,1,4,5,2,3,6,7 ; Маска для перестановки значений sp в умм.

section .bss
    alignb 32
    transpose resq 16
    trace resq 1
    bbhi_cy resq 1
    bblo_cy resq 1
```

```

        ebhi_cy      resq  1
        eblo_cy      resq  1
        bshi_cy      resq  1
        bslo_cy      resq  1
        eshi_cy      resq  1
        eslo_cy      resq  1

section .text
        global main
main:
push    rbp
mov     rbp, rsp
; Вывод заголовка.
        mov     rdi, fmt0
        call    printf
; Вывод матрицы.
        mov     rdi, fmt1
        call    printf
        mov     rsi, matrix
        call    printm8x8

; ВЕРСИЯ С ПОСЛЕДОВАТЕЛЬНЫМИ ВЫЧИСЛЕНИЯМИ.
; Вычисление следа матрицы.
        mov     rdi, matrix
        mov     rsi, [loops]
; Начало отсчета циклов процессора.
        cpuid
        rdtsc
        mov     [bshi_cy], edx
        mov     [bslo_cy], eax
        call    seq_trace
; Останов отсчета циклов процессора.
        rdtscp
        mov     [eshi_cy], edx
        mov     [eslo_cy], eax
        cpuid
; Вывод результата.
        mov     rdi, fmt2
        mov     rax, 1
        call    printf

; ВЕРСИЯ СО СМЕШИВАНИЕМ.
; Вычисление следа матрицы.
        mov     rdi, matrix
        mov     rsi, [loops]
; Начало отсчета циклов процессора.
        cpuid
        rdtsc
        mov     [bbhi_cy], edx
        mov     [bblo_cy], eax
        call    blend_trace
; Останов отсчета циклов процессора.
        rdtscp
        mov     [ebhi_cy], edx
        mov     [eblo_cy], eax
        cpuid

```



```
; Вывод результата.
mov     rdi, fmt5
mov     rax, 1
call    printf
; Вывод количества программных циклов.
mov     rdi, fmt4
mov     rsi, [loops]
call    printf
; Вывод количества циклов процессора.
; Количество циклов в версии с последовательными вычислениями.
mov     rdx, [eslo_cy]
mov     rsi, [eshi_cy]
shl     rsi, 32
or      rsi, rdx
mov     r8, [bslo_cy]
mov     r9, [bshi_cy]
shl     r9, 32
or      r9, r8
sub     rsi, r9          ; rsi содержит кол-во выполненных циклов.
; Вывод.
mov     rdi, fmt30
call    printf
; Количество циклов в версии AVX со смешиванием.
mov     rdx, [eblo_cy]
mov     rsi, [ebhi_cy]
shl     rsi, 32
or      rsi, rdx
mov     r8, [bblo_cy]
mov     r9, [bbhi_cy]
shl     r9, 32
or      r9, r8
sub     rsi, r9
; Вывод.
mov     rdi, fmt31
call    printf
leave
ret

;-----
seq_trace:
push    rbp
mov     rbp, rsp
.loop0:
xor     xmm0, xmm0
mov     rcx, 8
xor     rax, rax
xor     rbx, rbx
.loop:
addss   xmm0, [rdi+rax]
add     rax, 36          ; В каждой строке 32 байта.
loop    .loop
cvtss2sd    xmm0, xmm0
dec         rsi
jnz         .loop0
leave
ret
```

```

;-----
blend_trace:
push    rbp
mov     rbp, rsp
.loop:
    ; Формирование матрицы в памяти.
    vmovaps    ymm0, [rdi]
    vmovaps    ymm1, [rdi+32]
    vmovaps    ymm2, [rdi+64]
    vmovaps    ymm3, [rdi+96]
    vmovaps    ymm4, [rdi+128]
    vmovaps    ymm5, [rdi+160]
    vmovaps    ymm6, [rdi+192]
    vmovaps    ymm7, [rdi+224]

    vblendps    ymm0, ymm0, ymm1, 00000010b
    vblendps    ymm0, ymm0, ymm2, 00000100b
    vblendps    ymm0, ymm0, ymm3, 00001000b
    vblendps    ymm0, ymm0, ymm4, 00010000b
    vblendps    ymm0, ymm0, ymm5, 00100000b
    vblendps    ymm0, ymm0, ymm6, 01000000b
    vblendps    ymm0, ymm0, ymm7, 10000000b

    vhaddps     ymm0, ymm0, ymm0
    vmovdqu     ymm1, [perm]
    vpermps     ymm0, ymm1, ymm0
    haddps      xmm0, xmm0
    vextractps   r8d, xmm0, 0
    vextractps   r9d, xmm0, 1
    vmovd       xmm0, r8d
    vmovd       xmm1, r9d
    vaddss       xmm0, xmm0, xmm1
    dec         rsi
    jnz         .loop
cvtss2sd xmm0, xmm0
leave
ret

printm8x8:
section .data
.fmt db  "%.f", 9, "%.f", 9, "%.f", 9, "%.f", 9, "%.f", 9, "%.f", 9, "%.f", 9, "%.f", 10, 0
section .text
push    rbp
mov     rbp, rsp
push    rbx                ; Регистр, сохраняемый вызываемой функцией.
mov     rdi, .fmt
mov     rcx, 8
xor     rbx, rbx            ; Счетчик строк.
vzeroall
.loop:
    movss      xmm0, dword[rsi+rbx]
    cvtss2sd   xmm0, xmm0
    movss      xmm1, [rsi+rbx+4]
    cvtss2sd   xmm1, xmm1
    movss      xmm2, [rsi+rbx+8]
    cvtss2sd   xmm2, xmm2

```

```

movss    xmm3, [rsi+rbx+12]
cvtss2sd xmm3,xmm3
movss    xmm4, [rsi+rbx+16]
cvtss2sd xmm4,xmm4
movss    xmm5, [rsi+rbx+20]
cvtss2sd xmm5,xmm5
movss    xmm6, [rsi+rbx+24]
cvtss2sd xmm6,xmm6
movss    xmm7, [rsi+rbx+28]
cvtss2sd xmm7,xmm7
mov     rcx,8      ; 8 чисел с плавающей точкой.
push    rcx        ; Регистр, сохраняемый вызывающей функцией.
push    rsi        ; Регистр, сохраняемый вызывающей функцией.
push    rdi        ; Регистр, сохраняемый вызывающей функцией.
; Выравнивание стека, если необходимо.
xor     r15,r15
test    rsp,0fh    ; Последний байт равен 8 (стек не выровнен)?
setnz   r15b       ; Установка флага, если стек не выровнен.
shl     r15,3      ; Умножение на 8.
sub     rsp,r15    ; Вычитание 0 или 8.
call    printf
add     rsp,r15    ; Прибавление 0 или 8.
pop     rdi
pop     rsi
pop     rcx
add     rbx,32     ; Следующая строка.
loop    .loop
pop     rbx        ; Регистр, сохраняемый вызываемой функцией.
leave
ret

```

Функция `blend_trace` представляет собой расширение функции вычисления следа матрицы с размера 4×4 до размера 8×8 . Исходная версия функции использовалась в главе 36 в коде обращения матрицы с применением инструкций AVX. Функция `seq_trace` реализует последовательный проход по матрице, определяет элементы следа и выполняет их сложение. При запуске этой программы вы увидите, что функция `seq_trace` работает намного быстрее, чем функция `blend_trace`.

На рис. 38.2 показан вывод данной программы.

```

jo@ubuntu18:~/Desktop/Book/47 performance2$ ./trace
8x8 SINGLE PRECISION FLOATING POINT MATRIX TRACE

This is the matrix:
1,      2,      3,      4,      5,      6,      7,      8
9,      10,     11,     12,     13,     14,     15,     16
17,     18,     19,     20,     21,     22,     23,     24
25,     26,     27,     28,     29,     30,     31,     32
33,     34,     35,     36,     37,     38,     39,     40
41,     42,     43,     44,     45,     46,     47,     48
49,     50,     51,     52,     53,     54,     55,     56
57,     58,     59,     60,     61,     62,     63,     64

This is the trace (sequential version): 260.000000
This is the trace (AVX blend version): 260.000000

Number of loops: 1000
Sequential version elapsed cycles: 48668
AVX blend version elapsed cycles: 175509

jo@ubuntu18:~/Desktop/Book/47 performance2$ █

```

Рис. 38.2. Вывод программы *trace.asm*

Если вы хотите более подробно узнать об оптимизации, обратитесь к упомянутому выше тому руководства Intel. Еще один превосходный источник информации находится здесь: <https://www.agner.org/>.

РЕЗЮМЕ

В этой главе вы узнали о:

- методах измерения и вычисления выполненных циклов процессора;
- том, что инструкции AVX могут существенно ускорить обработку данных;
- том, что инструкции AVX не всегда эффективны (их эффективность зависит от конкретной задачи).

Глава 39

Приветствуем мир Windows

В этой главе и в нескольких следующих главах мы начинаем изучать программирование на ассемблере в ОС Windows. Если вашей основной системой является Linux, то наилучшим решением будет установка виртуальной машины Windows. Пробную версию Windows 10 с лицензией на 90 дней можно загрузить здесь: <https://www.microsoft.com/en-us/evalcenter/evaluate-windows-10-enterprise>. После установки пробной версии Windows 10 загрузите и установите все существующие на текущий момент обновления этой системы.

НАЧИНАЕМ ИЗУЧЕНИЕ

Компания Microsoft разработала собственный ассемблер под названием MASM, который включен в программную среду Visual Studio. Возможность использования Visual Studio, несомненно, является преимуществом, так как это полнофункциональное инструментальное средство разработки. Инструкции ассемблера, используемые в MASM, те же самые, что и в NASM, но директивы ассемблера совершенно другие. Конфигурирование и обучение практической работе с Visual Studio характеризуются кривой обучения, зависящей от вашего опыта как разработчика в среде Windows.

Для смягчения воздействия «культурного шока» при встрече с другой средой в этой книге используется версия ассемблера NASM для Windows с интерфейсом командной строки (CLI). В предыдущих главах вы вполне освоили NASM в среде Linux, и это является неплохой форой для начала изучения. Кроме того, переход на MASM не должен вызывать каких-либо серьезных затруднений.

Если вы планируете вести разработки для Windows, то освоение практического использования Visual Studio стоит затраченных усилий. В интернете можно даже найти информацию о том, как использовать NASM в Visual Studio.

Найдите в интернете NASM для Windows (в настоящее время версия NASM для Windows находится здесь: <https://www.nasm.us/pub/nasm/releasebuilds/2.14.03rc2/win64/>) и установите эту версию. Проверьте и убедитесь в том, что переменная среды PATH содержит запись, указывающую на каталог (папку), в которую установлен NASM (см. рис. 39.1). Проверить правильность установки NASM можно в командной строке, выполнив команду `nasm -v`.

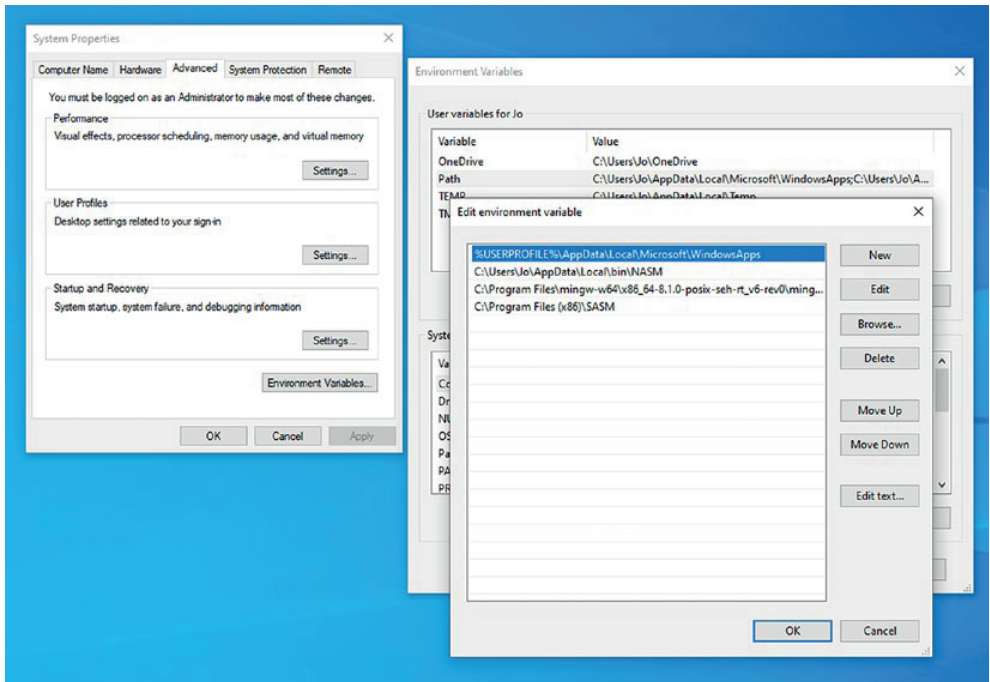


Рис. 39.1. Проверка переменной среды PATH в Windows 10

Мы также будем использовать версию MinGW (Minimalist GNU for Windows), представляющую собой комплект инструментальных средств разработки Linux, адаптированный для среды Windows. MinGW позволяет использовать такие инструменты разработки, как `make` и `gcc`, которыми мы часто пользовались в предыдущих главах. Необходимо установить версию MinGW-w64. Перед началом загрузки и установки, если вы планируете использовать SASM в Windows, следует знать, что SASM устанавливает NASM и некоторые инструменты MinGW-w64 в собственные подкаталоги (за исключением `make`). Если вы вручную устанавливаете SASM и MinGW-w64, то в итоге получите дублированные установленные компоненты. С помощью параметров настройки можно сконфигурировать SASM так, чтобы он использовал уже установленные версии NASM и GCC вместо более старых версий, включенных в дистрибутивный комплект SASM.

В настоящее время загружаемые файлы для MinGW-w64 расположены здесь: <http://mingw-w64.org/doku.php/download>. Выберите версию MinGW-w64-builds, загрузите и установите ее, в окне установки выберите вариант `x86_64`.

Далее необходимо открыть окно переменных среды Windows и добавить путь к каталогу `bin` в иерархии MinGW-w64 в переменную среды PATH, как показано на рис. 39.1. Каталог `bin` содержит компилятор `gcc`. После обновления переменной среды PATH откройте окно командной строки PowerShell и проверьте правильность установки командой `gcc -v`.

Загрузите версию SASM для win64 (<https://dman95.github.io/SASM/english.html>), и если необходимо, чтобы SASM использовал новые версии NASM и GCC, то измените параметры сборки (*build settings*) так, чтобы они указывали на только что установленные NASM и GCC. Не забудьте добавить в переменную среды PATH запись для SASM.

Если вы пока еще не выбрали текстовый редактор для работы в Windows, рекомендуется установить Notepad++. Это простой текстовый редактор, поддерживающий подсветку синтаксиса многих языков программирования, включая ассемблер. В нем можно без затруднений установить любую кодировку: UTF-8, UTF-16 и т. д. Настройку подсветки синтаксиса ассемблера можно найти в пункте основного меню **Language** (Язык).

Некоторое неудобство состоит в том, что в MinGW-w64 нет команды `make`, а существует только выполняемый файл *mingw32-make.exe*, имя которого слишком длинное для ввода в командной строке. Для решения этой проблемы в редакторе Notepad++ (требуется запуск от имени администратора) создайте файл *make.bat*, содержащий следующую строку:

```
mingw32-make.exe
```

Сохраните этот файл в кодировке UTF-8 в каталоге *bin* в иерархии MinGW-w64.

Ниже приведены некоторые советы и рекомендации, полезные при работе в Windows:

- чтобы открыть приложение от имени администратора, щелкните правой кнопкой по значку приложения и в контекстном меню выберите пункт **Run as administrator** (Запуск от имени администратора);
- всегда полезно иметь быстрый доступ к PowerShell, интерфейсу командной строки в Windows. На панели задач в поле поиска введите PowerShell, затем щелкните по кнопке **Open** (Открыть). На панели задач появится значок PowerShell, щелкните по нему правой кнопкой мыши и выберите пункт **Pin to taskbar** (Закрепить на панели задач);
- в окне, где отображаются значки файлов и каталогов, нажмите и удерживайте клавишу **Shift** и одновременно щелкните правой кнопкой мыши, далее в появившемся контекстном меню можно выбрать пункт **Open PowerShell window here** (Открыть окно PowerShell здесь);
- для отображения скрытых файлов и каталогов щелкните по значку **Проводник** (File Explorer) на панели задач. Откройте пункт меню **Вид** (View) и поставьте метку (галочку) для параметра **Скрытые элементы** (Hidden items);
- чтобы найти переменные среды, в поле поиска на панели задач введите строку «переменные среды» (environment variables).

ПИШЕМ КОД В WINDOWS

Теперь вы готовы к началу кодирования на ассемблере. В листингах 39.1 и 39.2 показана первая программа.

Листинг 39.1. Программа *hello.asm*

```

; hello.asm
extern printf
section .data
    msg db 'Hello, Windows World!',0
    fmt db "Windows 10 says: %s",10,0
section .text
    global main
main:
push rbp
mov rbp, rsp
    mov rcx, fmt
    mov rdx, msg
    sub rsp, 32
    call printf
    add rsp, 32
leave
ret

```

Листинг 39.2. *makefile*

```

hello.exe: hello.obj
    gcc -o hello.exe hello.obj
hello.obj: hello.asm
    nasm -f win64 -g -F cv8 hello.asm -l hello.lst

```

В этом ассемблерном коде нет ничего примечательного. Или все-таки есть?

Во-первых, здесь используется инструкция `sub rsp, 32`, которая в Linux применялась для создания стековых переменных. С помощью этой инструкции создается скрытое пространство (*shadow space*) в стеке перед вызовом функции. Более подробно мы обсудим это немного позже. После выполнения функции `printf` стек восстанавливается инструкцией `add rsp, 32`, которая в рассматриваемом здесь примере не является обязательной, так как стек будет восстановлен инструкцией `leave`. Регистры, используемые для передачи аргументов в функцию `printf`, отличаются от регистров, используемых в Linux. Причина в том, что соглашения о вызовах функций для Windows отличаются от соглашений о вызовах функций для Linux. В Windows требуется соблюдение соглашений Microsoft x64, тогда как Linux требует использования соглашений System V Application Binary Interface, также называемой System V ABI.

Общий обзор соглашений о вызовах функций Microsoft x64 можно найти здесь: <https://docs.microsoft.com/en-us/cpp/build/x64-calling-convention?view=vs-2019>. Место расположения этой страницы иногда изменяется, так что если вы не можете ее найти, то поищите на сайте Microsoft «x64 calling convention». Ниже приводится краткое содержание данного обзора:

- целочисленные аргументы передаются в регистрах `rcx`, `rdx`, `r8`, `r9` в указанном здесь порядке;
- если необходимо передать большее количество аргументов, то они помещаются в стек;
- аргументы с плавающей точкой передаются в регистрах `xmm0`–`xmm3`. Большее количество аргументов передается через стек;

- регистры `rcx`, `rdx`, `r8`, `r9` и дополнительно `rax`, `r10`, `r11`, `xmm4` и `xmm5` являются изменяемыми, т. е. вызывающая сторона должна сохранять их, если это необходимо. Прочие регистры должны сохраняться вызываемой стороной (функцией);
- вызывающая сторона должна обеспечить 32-байтовое пространство в стеке (скрытое пространство – *shadow space*) для четырех аргументов функции, передаваемых вызываемой стороне, даже если вызываемая сторона не принимает так много аргументов;
- как и в Linux, стек обязательно должен быть выровнен по 16-байтовой границе.

На рис. 39.2 показан вывод первой программы в Windows.



```

Windows PowerShell
PS C:\Users\Jo\asm64win\01 hello> make
C:\Users\Jo\asm64win\01 hello> mingw32-make.exe
nasm -f win64 -g -F cv8 hello.asm -l hello.lst
gcc -o hello.exe hello.obj
PS C:\Users\Jo\asm64win\01 hello> .\hello.exe
Windows 10 says: Hello, Windows World!
PS C:\Users\Jo\asm64win\01 hello>

```

Рис. 39.2. Вывод программы *hello.asm*

Отладка

После запуска GDB для отладки первой Windows-программы вы будете удивлены. Вы сможете выполнить некоторые команды, но пошаговый проход по коду не работает. Выводится следующее сообщение:

```

Single stepping until exit from function main,
which has no line number information.
0x0000000000402a60 in printf ()

```

Это означает, что в Windows использование GDB ограничено. Но на помощь приходит SASM. В SASM такой проблемы не существует. Тем не менее в *makefile* сохранены флаги отладки: возможно, в следующей версии GDB проблема пошагового прохода будет решена. В *makefile* для Windows задан флаг `cv8` (Microsoft CodeView 8), определяющий формат отладки.

СИСТЕМНЫЕ ВЫЗОВЫ

В рассматриваемом здесь примере исходного кода использовалась функция `printf` вместо инструкции системного вызова `syscall`, примененной в первой ассемблерной программе в Linux. Причина в том, что в Windows не используются системные вызовы. Системные вызовы существуют, но предназначены они «только для внутреннего пользования» Windows. Если необходим доступ к системным ресурсам, то необходимо воспользоваться прикладным программным интерфейсом Windows API. Разумеется, можно покопаться в коде Windows или поискать в интернете имена системных вызовов, но при этом

следует помнить о том, что в новых версиях Windows использование системных вызовов может измениться, и если вы решились применить их, то ваш код, возможно, перестанет работать.

РЕЗЮМЕ

В этой главе вы узнали:

- как установить и использовать NASM, SASM и комплект инструментальных средств разработки Linux в операционной системе Windows;
- о том, что соглашения о вызовах функций Windows отличаются от соглашений о вызовах функций Linux;
- о том, что в Windows лучше не использовать системные вызовы.

Глава 40

Использование Windows API

Прикладной программный интерфейс Windows – Windows API – это набор инструкций, которые может использовать разработчик для взаимодействия с операционной системой. Как было отмечено в предыдущей главе, системные вызовы не являются надежным способом обмена информацией с операционной системой, но компания Microsoft предоставляет обширный набор прикладных программных интерфейсов API для выполнения практически любых операций, которые могут потребоваться разработчику. Windows API написан с учетом свойств языка программирования C, но если соблюдать соглашения о вызовах функций, то можно без затруднений использовать Windows API в ассемблерных программах. В настоящее время (на момент написания книги) описание Windows API можно найти здесь: <https://docs.microsoft.com/en-us/windows/win32/api/>.

Вывод в консоли

В листинге 40.1 показана версия программы Hello World, в которой используется Windows API для вывода сообщения на экран.

Листинг 40.1. Программа *helloworld.asm*

```
; helloworld.asm
%include "win32n.inc"
    extern WriteFile
    extern WriteConsoleA
    extern GetStdHandle

section .data
    msg          db      'Hello, World!!',10,0
    msglen       EQU     $-msg-1          ; Отбросить NULL.

section .bss
    hFile        resq     1              ; Идентификатор файла.
    lpNumberOfBytesWritten resq 1

section .text
    global main
main:
    push    rbp
    mov     rbp,rsp
```

```

; Связать идентификатор файла со стандартным выводом stdout.
; HANDLE WINAPI GetStdHandle(
;   _In_ DWORD nStdHandle
; );
    mov     rcx, STD_OUTPUT_HANDLE
    sub     rsp, 32           ; Обеспечение скрытого пространства.
    call    GetStdHandle     ; Возврат INVALID_HANDLE_VALUE, если попытка неудачна.
    add     rsp, 32
    mov     qword[hFile], rcx ; Сохранить полученный идентификатор в памяти.

; BOOL WINAPI WriteConsole(
;   _In_      HANDLE      hConsoleOutput,
;   _In_      const VOID  *lpBuffer,
;   _In_      DWORD       nNumberOfCharsToWrite,
;   _Out_     LPDWORD      lpNumberOfCharsWritten,
;   _Reserved_ LPVOID      lpReserved
; );
    sub     rsp, 8           ; Выравнивание стека.
    mov     rcx, qword[hFile]
    lea     rdx, [msg]       ; lpBuffer
    mov     r8, msglen       ; nNumberOfBytesToWrite
    lea     r9, [lpNumberOfBytesWritten]
    push    NULL             ; lpReserved
    sub     rsp, 32
    call    WriteConsoleA    ; Возврат ненулевого значения, если успешное выполнение.
    add     rsp, 32+8

; BOOL WriteFile(
;   HANDLE      hFile,
;   LPCVOID     lpBuffer,
;   DWORD       nNumberOfBytesToWrite,
;   LPDWORD     lpNumberOfBytesWritten,
;   LPOVERLAPPED lpOverlapped
; );
    mov     rcx, qword[hFile] ; Идентификатор файла.
    lea     rdx, [msg]       ; lpBuffer
    mov     r8, msglen       ; nNumberOfBytesToWrite
    lea     r9, [lpNumberOfBytesWritten]
    push    NULL             ; lpOverlapped
    sub     rsp, 32
    call    WriteFile        ; Возврат ненулевого значения, если успешное выполнение.
leave
ret

```

В документации Windows API описано использование нескольких тысяч символьных констант. Это делает исходный код более удобным для чтения и упрощает применение Windows API, поэтому в исходный код включен файл *win32n.inc* в начале программы. Это список всех символьных констант и их значений. Файл *win32n.inc* можно скачать здесь: <http://rs1.szif.hu/~tomcat/win32/>³. Но следует помнить о том, что включение данного файла в исходный код значительно увеличивает размер выполняемого файла. Если размер важен, то следует

³ Внимание: при попытке перехода по этой ссылке выводится сообщение, предупреждающее о том, что этот сайт, возможно, содержит вредоносные программы и атакует сетевые компьютеры. – Прим. перев.

выборочно включить в исходный код только те константы, которые действительно необходимы в программе. Если используется SASM, найдите каталог (папку), в котором установлен SASM, и вручную скопируйте файл *win32n.inc* в подкаталог *include* в иерархии SASM в вашей системе.

В исходный код примера включены копии структур вызовов функций Windows в виде комментариев, поэтому можно с легкостью наблюдать за тем, что происходит. Аргументы записываются в регистры в соответствии с соглашением о вызовах функций, обеспечивается наличие скрытого пространства (*shadow space*) в стеке, вызывается функция, затем восстанавливается указатель стека.

Функция *GetStdHandle* возвращает идентификатор (*handle*), если все идет правильно, иначе возвращается специальное значение *INVALID_HANDLE_VALUE*. Чтобы не усложнять программу, мы не обеспечиваем контроль ошибок, но в реальных программных продуктах рекомендуется реализовать полноценную систему контроля ошибок. Критический сбой может не только привести к краху программы, но, что значительно хуже, может стать причиной нарушения системы безопасности.

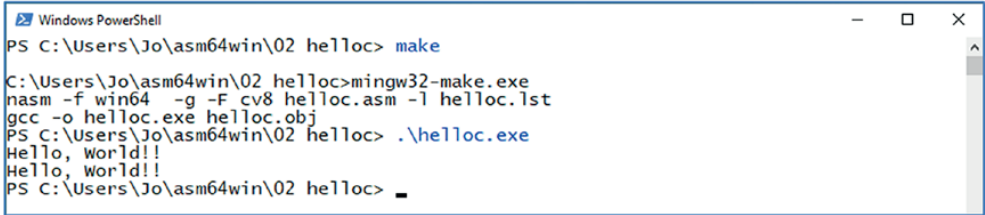
После получения идентификатора вызывается функция *WriteConsoleA*, в нее передается идентификатор, выводимая строка, длина этой строки, шаблон для резервирования заданного количества записываемых байтов и *NULL* как зарезервированный аргумент. Первые четыре аргумента передаются в регистрах, пятый аргумент помещается в стек. Эта операция записи в стек приводит к тому, что стек становится невыровненным, – об этом нужно помнить перед записью аргумента в стек. Если выравнивание выполнено после записи в стек (после инструкции *push*), то вызываемая функция не найдет аргумент в стеке. Поэтому непосредственно перед вызовом функции в стеке создается скрытое пространство.

Рассматриваемая здесь программа использует два метода записи в консоль: первый метод применяет функцию *WriteConsoleA*, второй – функцию *WriteFile*. Функция *WriteFile* использует тот же идентификатор и рассматривает консоль как обычный файл для записи в него. После выполнения *WriteConsoleA* стек восстанавливается с учетом предварительного создания скрытого пространства выравнивания стека. После выполнения *WriteFile* стек не восстанавливается явно, потому что это будет сделано инструкцией *leave*.

Если вы не нашли описания функции *WriteConsoleA* в документации Windows API, то попробуйте поискать *WriteConsole*. В документации объясняется, что существуют две версии: *WriteConsoleA* для вывода символов в кодировке ANSI и *WriteConsoleW* для вывода символов в кодировке Unicode.

Если начать выполнение этого кода в SASM, то вы увидите, что первый метод с использованием функции *WriteConsoleA* не работает. Функция возвращает 0 в регистре *eax*, сообщая о каких-то неполадках. Причина – в возникновении конфликта между системной консолью и собственной консолью SASM. Метод с использованием функции *WriteFile* работает нормально.

На рис. 40.1 показан вывод этой программы.



```

Windows PowerShell
PS C:\Users\Jo\asm64win\02 hellow> make
C:\Users\Jo\asm64win\02 hellow> mingw32-make.exe
nasm -f win64 -g -F cv8 hellow.asm -l hellow.lst
gcc -o hellow.exe hellow.obj
PS C:\Users\Jo\asm64win\02 hellow> ./hellow.exe
Hello, World!!
Hello, World!!
PS C:\Users\Jo\asm64win\02 hellow> _

```

Рис. 40.1. Вывод программы *hellow.asm*

СОЗДАНИЕ ОКОН WINDOWS

Теперь вместо вывода в консоль воспользуемся графическим пользовательским интерфейсом (GUI) Windows. Здесь не будет рассматриваться полнофункциональная программа Windows, необходимо всего лишь продемонстрировать, как выводится окно. Если вы хотите большего, то придется более подробно изучать документацию Windows API. Когда вы поймете, как работать с окном, останется только найти требуемую функцию в документации Windows API и передать в нее необходимые аргументы в регистрах и в стеке.

В листинге 40.2 показан пример исходного кода.

Листинг 40.2. Программа *hellow.asm*

```

; hellow.asm
%include "win32n.inc"
extern ExitProcess
extern MessageBoxA

section .data
    msg     db 'Welcome to Windows World!',0
    cap     db "Windows 10 says:",0

section .text
    global main
main:
    push    rbp
    mov     rbp,rsp

; int MessageBox(
;     HWND hWnd,           Владелец окна.
;     LPCSTR lpText,       Выводимый текст.
;     LPCSTR lpCaption,    Заголовок окна.
;     UINT  uType           Поведение окна.
; )
    mov     rcx,0           ; У окна нет владельца.
    lea     rdx,[msg]       ; lpText
    lea     r8,[cap]        ; lpCaption
    mov     r9d,MB_OK       ; Окно с кнопкой ОК.
    sub     rsp,32          ; Скрытое пространство в стеке.
    call    MessageBoxA     ; Возврат IDOK=1, если выбрана кнопка ОК.
    add     rsp,32

leave
ret

```

На рис. 40.2 показан вывод этой программы.

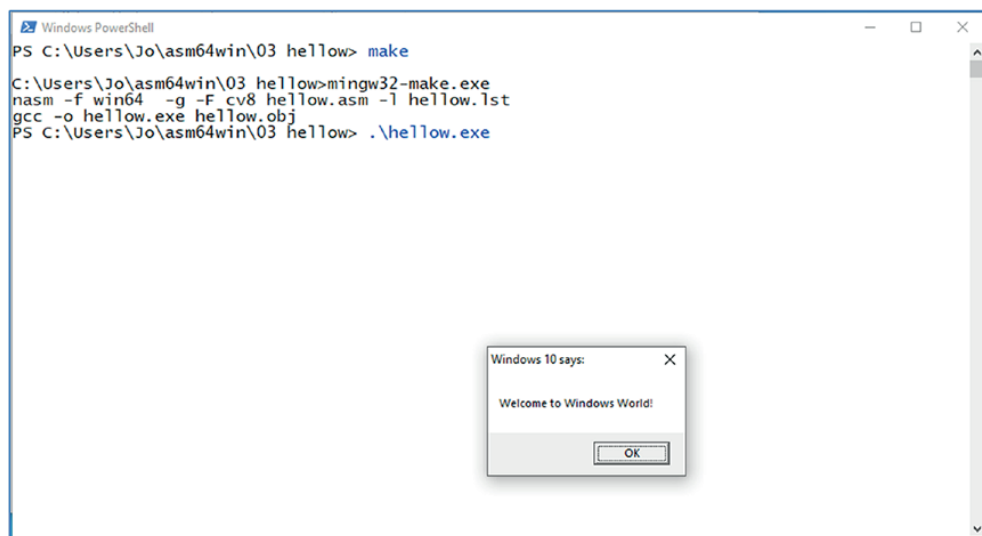


Рис. 40.2. Вывод программы *hellow.asm*

Разумеется, у вас может возникнуть вопрос: является ли ассемблер подходящим языком программирования для создания GUI для программы Windows? Для этой цели гораздо проще использовать С или С++ и вызывать ассемблерные функции (или применять ассемблерные вставки непосредственно в код) в тех частях программы, в которых требуется особенно высокая производительность.

В любом случае вы можете взять качественную книгу по программированию в Windows на языке С или С++, где подробно описывается Windows API, и переписать все вызовы функций на ассемблере с использованием правильных регистров, а затем вызывать эти функции, как показано в данном разделе. Разумеется, в программе с GUI необходима более сложная функциональность, например подсистема контроля ошибок, но подобные подсистемы гораздо проще разрабатывать на языке высокого уровня.

РЕЗЮМЕ

В этой главе вы узнали, как:

- использовать Windows API;
- вывести сообщение в окне командной строки (PowerShell) Windows;
- использовать инструкции `GetStdHandle`, `WriteConsole` и `WriteFile`;
- создать окно с кнопкой.

Глава 41

Функции в Windows

Передача аргументов в функции выполняется просто, если необходимо передать в них четыре или меньше аргументов, не являющихся значениями с плавающей точкой. Для этого используются регистры rcx, rdx, r8, r9 и обеспечивается наличие скрытого пространства (*shadow space*) в стеке перед вызовом функции. После вызова необходимо скорректировать стек с учетом скрытого пространства, и все будет в порядке. Вызов функции становится более сложным, если необходимо передать более четырех аргументов.

ИСПОЛЬЗОВАНИЕ БОЛЕЕ ЧЕТЫРЕХ АРГУМЕНТОВ ФУНКЦИИ

Сначала рассмотрим, почему вызов функции становится более сложным при необходимости передать в нее более четырех аргументов, как показано в листинге 41.1.

Листинг 41.1. Программа *arguments1.asm*

```
; arguments1.asm
extern printf
section .data
    first    db    "A",0
    second   db    "B",0
    third     db    "C",0
    fourth    db    "D",0
    fifth     db    "E",0
    sixth     db    "F",0
    seventh   db    "G",0
    eighth    db    "H",0
    ninth     db    "I",0
    tenth     db    "J",0
    fmt       db    "The string is: %s%s%s%s%s%s%s%s",10,0
section .bss
section .text
    global main
main:
    push     rbp
    mov      rbp, rsp
    sub      rsp, 8
    mov      rcx, fmt
    mov      rdx, first
    mov      r8, second
```



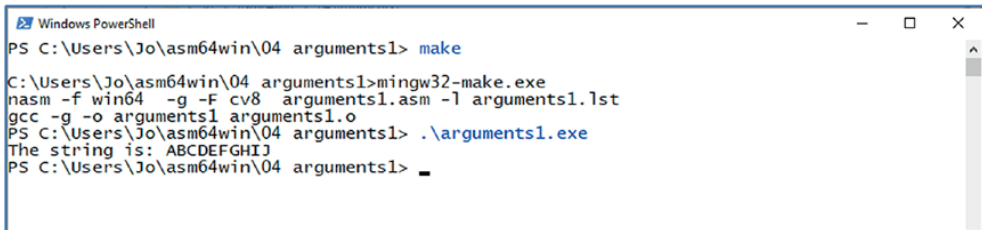
```

mov    r9, third
push   tenth           ; Здесь начинается запись аргументов в стек
push   ninth           ; в обратном порядке.
push   eighth
push   seventh
push   sixth
push   fifth
push   fourth
sub     rsp,32          ; Скрытое пространство в стеке.
call   printf
add     rsp,32+8        ; Восстановление стека.
leave
ret

```

Обратите внимание на инструкцию `sub rsp,8` – она нужна потому, что вызывается функция `printf`, поэтому стек должен быть выровнен по 16-байтовой границе. Почему бы не использовать одну инструкцию, например, `sub rsp,40` непосредственно перед вызовом функции `printf`? В этом случае стек был бы выровнен по 16-байтовой границе, но функция `printf`, вероятнее всего, будет выполнена некорректно. Если уменьшить указатель стека на 40 вместо 32 непосредственно перед вызовом функции, то аргументы будут находиться в стеке не в тех позициях, где их предполагает найти функция `printf`, т. е. прямо над скрытым пространством. Поэтому стек необходимо выравнивать перед началом записи аргументов. Следует отметить, что аргументы должны записываться в стек в обратном порядке. После вызова функции стек восстанавливается с учетом его предварительного выравнивания и создания скрытого пространства.

На рис. 41.1 показан вывод этой программы.



```

Windows PowerShell
PS C:\Users\Jo\asm64win\04 arguments1> make
C:\Users\Jo\asm64win\04 arguments1> mingw32-make.exe
nasm -f win64 -g -F cv8 arguments1.asm -l arguments1.lst
gcc -g -o arguments1 arguments1.o
PS C:\Users\Jo\asm64win\04 arguments1> .\arguments1.exe
The string is: ABCDEFGHIJ
PS C:\Users\Jo\asm64win\04 arguments1>

```

Рис. 41.1. Вывод программы *arguments1.asm*

Правильно сформировать стек можно другим способом. В листинге 41.2 показано, как это делается.

Листинг 41.2. Программа *arguments2.asm*

```

;arguments2.asm
extern printf
section .data
    first      db    "A",0
    second     db    "B",0
    third      db    "C",0
    fourth     db    "D",0
    fifth      db    "E",0
    sixth      db    "F",0

```

```

    seventh      db  "G",0
    eighth       db  "H",0
    ninth        db  "I",0
    tenth        db  "J",0
    fmt          db  "The string is: %s%s%s%s%s%s%s%s",10,0
section .bss
section .text
    global main
main:
push  rbp
mov   rbp, rsp
sub   rsp, 32+56+8 ; Скрытое пространство + 7 аргументов в стеке + выравнивание.
mov   rcx, fmt
mov   rdx, first
mov   r8, second
mov   r9, third
mov   qword[rsp+32], fourth
mov   qword[rsp+40], fifth
mov   qword[rsp+48], sixth
mov   qword[rsp+56], seventh
mov   qword[rsp+64], eighth
mov   qword[rsp+72], ninth
mov   qword[rsp+80], tenth
call  printf
add   rsp, 32+56+8 ; Это не обязательно перед инструкцией leave.
leave
ret

```

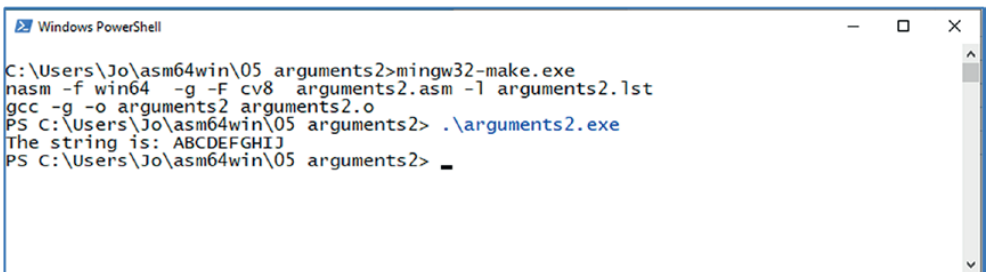
Сначала используется инструкция `sub rsp, 32+56+8` для выравнивания стека:

- 32 байта для скрытого пространства;
- в стек записывается 7 аргументов по 8 байт, всего 56 байт.

Затем начинается формирование стека, и когда вы видите, что необходимо выровнять стек, еще 8 байт требуется вычесть из указателя стека.

Теперь в нижней части стека имеется 32 байта для скрытого пространства, а непосредственно над ним расположен четвертый аргумент, выше него находится пятый аргумент и т. д. Стек, сформированный здесь, выглядит так же, как стек в предыдущей программе (листинг 41.1). Решайте сами, какую версию предпочесть.

На рис. 41.2 показан вывод данной программы.



```

C:\Users\Jo\asm64win\05 arguments2>mingw32-make.exe
nasm -f win64 -g -F cv8 arguments2.asm -l arguments2.lst
gcc -g -o arguments2 arguments2.o
PS C:\Users\Jo\asm64win\05 arguments2> .\arguments2.exe
The string is: ABCDEFGHIJ
PS C:\Users\Jo\asm64win\05 arguments2>

```

Рис. 41.2. Вывод программы *arguments2.asm*

Как это работает в вызываемой функции? В листинге 41.3 показан пример исходного кода, в котором используется функция `lfunc` для создания буфера строки, выводимой функцией `printf`.

Листинг 41.3. Программа *stack.asm*

```
; stack.asm
extern printf
section .data
    first      db    "A"
    second     db    "B"
    third      db    "C"
    fourth     db    "D"
    fifth      db    "E"
    sixth      db    "F"
    seventh    db    "G"
    eighth     db    "H"
    ninth      db    "I"
    tenth      db    "J"
    fmt        db    "The string is: %s",10,0
section .bss
    flist resb 14      ; Длина строки плюс завершающий 0.
section .text
    global main
main:
    push rbp
    mov rbp, rsp
    sub rsp, 8
    mov rcx, flist
    mov rdx, first
    mov r8, second
    mov r9, third
    push tenth      ; Здесь начинается запись аргументов в стек
    push ninth      ; в обратном порядке.
    push eighth
    push seventh
    push sixth
    push fifth
    push fourth
    sub rsp,32      ; Скрытое пространство.
    call lfunc
    add rsp,32+8
; Вывод результата.
    mov rcx, fmt
    mov rdx, flist
    sub rsp,32+8
    call printf
    add rsp,32+8
leave
ret

;-----
lfunc:
    push rbp
    mov rbp, rsp
    xor rax, rax      ; Очистка rax (особенно старших битов).
```

```

; Обработка аргументов в регистрах.
mov al,byte[rdx]          ; Запись содержимого аргумента в al.
mov [rcx], al             ; Запись al в память.
mov al, byte[r8]
mov [rcx+1], al
mov al, byte[r9]
mov [rcx+2], al
; Обработка аргументов в стеке.
xor rbx,rbx
mov rax, qword [rbp+8+8+32] ; rsp + rbp + адрес возврата + скрытое пространство.
mov bl,[rax]
mov [rcx+3], bl
mov rax, qword [rbp+48+8]
mov bl,[rax]
mov [rcx+4], bl
mov rax, qword [rbp+48+16]
mov bl,[rax]
mov [rcx+5], bl
mov rax, qword [rbp+48+24]
mov bl,[rax]
mov [rcx+6], bl
mov rax, qword [rbp+48+32]
mov bl,[rax]
mov [rcx+7], bl
mov rax, qword [rbp+48+40]
mov bl,[rax]
mov [rcx+8], bl
mov rax, qword [rbp+48+48]
mov bl,[rax]
mov [rcx+9], bl
mov bl,0                  ; Завершающий ноль.
mov [rcx+10], bl
leave
ret

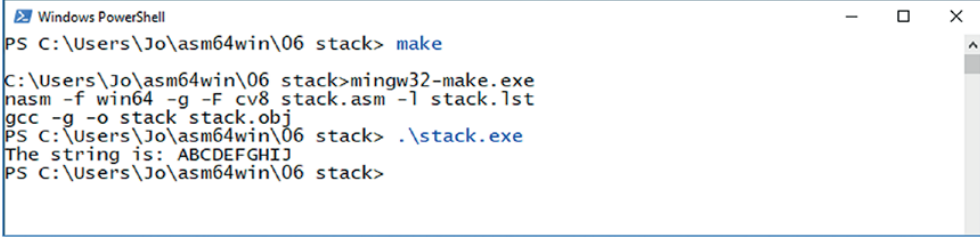
```

Основная функция `main` та же, что в программе *arguments1.asm*, но вызываемой функцией здесь становится `lfunc`, а не `printf`, которая вызывается позже.

В функции `lfunc` обратите внимание на инструкцию `mov rax, qword [rbp+8+8+32]`, которая загружает четвертый аргумент из стека в регистр `rax`. Регистр `rbp` содержит копию указателя стека. Первое 8-байтовое значение в стеке – это содержимое `rbp`, сохраненное в прологе функции `lfunc`. Выше находится 8-байтовое значение адреса возврата в функцию `main`, которое автоматически сохраняется в стеке при вызове функции `lfunc`. Затем следует скрытое пространство размером 32 байта. Наконец, мы добрались до аргументов, передаваемых в стеке. Таким образом, четвертый и все последующие аргументы можно найти по адресу `rbp+48` и выше по стеку.

После возврата в основную функцию `main` стек снова выравнивается, и вызывается функция `printf`.

На рис. 41.3 показан вывод этой программы, который, разумеется, полностью совпадает с выводом предыдущей программы.



```

Windows PowerShell
PS C:\Users\Jo\asm64win\06 stack> make

C:\Users\Jo\asm64win\06 stack> mingw32-make.exe
nasm -f win64 -g -F cv8 stack.asm -l stack.lst
gcc -g -o stack stack.obj
PS C:\Users\Jo\asm64win\06 stack> ./stack.exe
The string is: ABCDEFGHIJ
PS C:\Users\Jo\asm64win\06 stack>

```

Рис. 41.3. Вывод программы *stack.asm*

ОБРАБОТКА ЗНАЧЕНИЙ С ПЛАВАЮЩЕЙ ТОЧКОЙ

Для работы со значениями с плавающей точкой применяется другая методика. В листинге 41.4 показан пример исходного кода.

Листинг 41.3. Программа *stack_float.asm*

```

; stack_float.asm
extern printf
section .data
    zero    dq    0.0          ; 0x0000000000000000
    one     dq    1.0          ; 0x3FF0000000000000
    two     dq    2.0          ; 0x4000000000000000
    three   dq    3.0          ; 0x4008000000000000
    four    dq    4.0          ; 0x4010000000000000
    five    dq    5.0          ; 0x4014000000000000
    six     dq    6.0          ; 0x4018000000000000
    seven   dq    7.0          ; 0x401C000000000000
    eight   dq    8.0          ; 0x4020000000000000
    nine    dq    9.0          ; 0x4022000000000000

section .bss
section .text
    global main
main:
    push rbp
    mov rbp, rsp
    movq xmm0, [zero]
    movq xmm1, [one]
    movq xmm2, [two]
    movq xmm3, [three]

    movq xmm4, [nine]
    sub  rsp, 8
    movq [rsp], xmm4

    movq xmm4, [eight]
    sub  rsp, 8
    movq [rsp], xmm4

    movq xmm4, [seven]
    sub  rsp, 8
    movq [rsp], xmm4

    movq xmm4, [six]

```

```

    sub    rsp, 8
    movq   [rsp], xmm4

    movq   xmm4, [five]
    sub    rsp, 8
    movq   [rsp], xmm4

    movq   xmm4, [four]
    sub    rsp, 8
    movq   [rsp], xmm4

    sub    rsp, 32      ; Скрытое пространство.
    call   lfunc
    add    rsp, 32

leave
ret

;-----
lfunc:
push    rbp
mov     rbp, rsp
movsd   xmm4, [rbp+8+8+32]
movsd   xmm5, [rbp+8+8+32+8]
movsd   xmm6, [rbp+8+8+32+16]
movsd   xmm7, [rbp+8+8+32+24]
movsd   xmm8, [rbp+8+8+32+32]
movsd   xmm9, [rbp+8+8+32+40]

leave
ret

```

В этой небольшой программе вывод не выполняется из-за некоторого курьеза, который будет описан в следующей главе. Здесь потребуется отладчик для наблюдения за `xmm`-регистрами. Для удобства значения с плавающей точкой в комментариях представлены в шестнадцатеричном формате. Первые четыре значения передаются в функцию в регистрах `xmm0`–`xmm3`. Остальные аргументы записываются в стек. Следует помнить, что `xmm`-регистры могут содержать одно скалярное значение двойной точности, два упакованных значения двойной точности или четыре упакованных значения обычной (одиночной) точности. В рассматриваемом здесь примере используется одно скалярное значение двойной точности, а для наглядности значения сохраняются в стеке без применения инструкции `push`. Это один из возможных способов сохранения упакованных значений в стеке с выравниванием указателя стека `rsp` соответствующим образом после каждой операции записи. Более эффективным способом является запись скалярного значения непосредственно из памяти в стек, как показано ниже:

```
push qword[nine]
```

В вызываемой функции необходимо копировать значения из стека в `xmm`-регистры, в которых можно в дальнейшем выполнять обработку этих значений.

РЕЗЮМЕ

В этой главе вы узнали, как:

- передавать в функции аргументы в регистрах и в стеке;
- использовать скрытое пространство в стеке;
- получить доступ к аргументам, переданным в стеке;
- сохранять значения с плавающей точкой в стеке.

Глава 42

Функции с переменным числом аргументов

Функция с переменным числом аргументов (*variadic function*) – это функция, которая может принимать различное (меняющееся) количество аргументов. Хорошим примером такой функции является `printf`. Напомню, что в ассемблерных программах для Linux при работе функции `printf` с `xmm`-регистрами требовалось соблюдение соглашения о том, что регистр `%x` содержит количество `xmm`-регистров, которые должна использовать функция `printf`. Число регистров можно также извлечь из инструкции формата `printf`, так что зачастую можно обойтись без использования `%x`. Например, следующая строка формата определяет, что требуется вывод четырех значений с плавающей точкой, каждое из которых содержит девять десятичных цифр:

```
fmt    db    "%.f",9, "%.f",9, "%.f",9, "%.f",10,0
```

Даже если соглашение об определении числа обрабатываемых значений с плавающей точкой в регистре `%x` не соблюдается, функция `printf` все равно должна вывести эти четыре значения.

ФУНКЦИИ С ПЕРЕМЕННЫМ ЧИСЛОМ АРГУМЕНТОВ В WINDOWS

В Windows применяется другой процесс обработки переменного числа аргументов. Если в `xmm`-регистрах содержатся первые четыре аргумента, то необходимо скопировать их в соответствующие регистры для аргументов. В листинге 42.1 показан пример исходного кода.

Листинг 42.1. Программа *variadic1.asm*

```
; variadic1.asm
extern printf
section .data
    one    dq    1.0
    two    dq    2.0
    three  dq    3.0

    fmt    dq    "The values are: %.1f %.1f %.1f",10,0
```



```

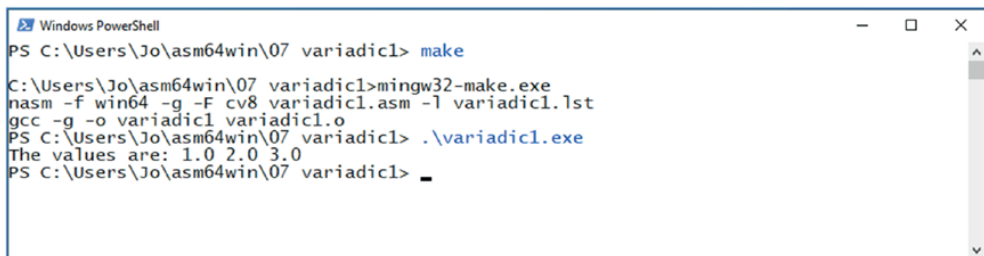
section .bss
section .text
    global main
main:
push    rbp
mov     rbp,rsp
    sub    rsp,32          ; Скрытое пространство в стеке.
    mov    rcx,fmt
    movq   xmm0,[one]
    movq   rdx,xmm0
    movq   xmm1,[two]
    movq   r8,xmm1
    movq   xmm2,[three]
    movq   r9,xmm2
    call   printf
    add    rsp, 32          ; Это не обязательно перед инструкцией leave.
leave
ret

```

При создании скрытого пространства перед вызовом функции рекомендуется установить для себя полезное правило: всегда удалять это скрытое пространство после завершения выполнения функции. В рассматриваемом здесь примере в инструкции `add rsp,32` нет необходимости, потому что прямо за ней следует инструкция `leave`, которая в любом случае восстанавливает указатель стека. Здесь вызывается только одна функция (`printf`), но если в программе вызывается несколько функций, то никогда не следует забывать об обязательном создании скрытого пространства в стеке перед вызовом и об обязательном удалении скрытого пространства каждый раз, когда продолжается выполнение после вызова функции.

Здесь можно видеть, что значения с плавающей точкой копируются в `xmm`-регистры и в регистры общего назначения, предназначенные для передачи аргументов. Это требование Windows. Подробное описание данного процесса выходит за рамки этой книги, но соблюдение такого требования обязательно при использовании функций языка C без прототипа или с переменным числом аргументов. Если закомментировать инструкции копирования значений в регистры общего назначения, то `printf` не сможет вывести правильные результаты.

На рис. 42.1 показан вывод этой программы.



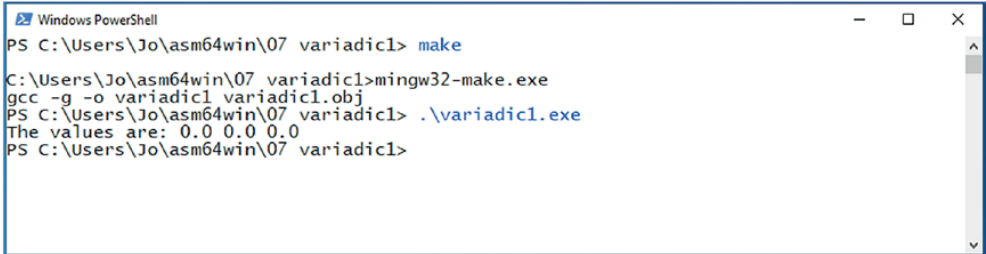
```

Windows PowerShell
PS C:\Users\Jo\asm64win\07 variadic1> make
C:\Users\Jo\asm64win\07 variadic1> mingw32-make.exe
nasm -f win64 -g -F cv8 variadic1.asm -l variadic1.lst
gcc -g -o variadic1 variadic1.o
PS C:\Users\Jo\asm64win\07 variadic1> .\variadic1.exe
The values are: 1.0 2.0 3.0
PS C:\Users\Jo\asm64win\07 variadic1>

```

Рис. 42.1. Вывод программы *variadic1.asm*

На рис. 42.2 представлен вывод той же программы без использования регистров общего назначения.



```

Windows PowerShell
PS C:\Users\Jo\asm64win\07 variadic1> make
C:\Users\Jo\asm64win\07 variadic1> mingw32-make.exe
gcc -g -o variadic1 variadic1.obj
PS C:\Users\Jo\asm64win\07 variadic1> .\variadic1.exe
The values are: 0.0 0.0 0.0
PS C:\Users\Jo\asm64win\07 variadic1>

```

Рис. 42.2. Некорректный вывод программы *variadic1.asm*

ОБРАБОТКА СМЕШАННЫХ ЗНАЧЕНИЙ

В листинге 42.2 показан пример обработки значений с плавающей точкой, объединенных со значениями других типов.

Листинг 42.2. Программа *variadic2.asm*

```

; variadic2.asm
extern printf

section .data
    fmt db    "%1f %s %1f %s %1f %s %1f %s %1f %s",10,0
    one dq    1.0
    two dq    2.0
    three dq   3.0
    four dq    4.0
    five dq    5.0
    A db    "A",0
    B db    "B",0
    C db    "C",0
    D db    "D",0
    E db    "E",0

section .bss
section .text
    global main
main:
    push rbp
    mov rbp, rsp
    sub    rsp,8                ; Сначала выравнивается стек.
    mov    rcx,fmt              ; Первый аргумент.
    movq   xmm0,[one]           ; Второй аргумент.
    movq   rdx,xmm0
    mov    r8,A                 ; Третий аргумент.
    movq   xmm1,[two]           ; Четвертый аргумент.
    movq   r9,xmm1
    ; Теперь запись аргументов в стек в обратном порядке.
    push   E                    ; 11-й аргумент.

    push   qword[five]          ; 10-й аргумент.

```

```

    push    D                ; 9-й аргумент.

    push    qword[four]      ; 8-й аргумент.

    push    C                ; 7-й аргумент.

    push    qword[three]     ; 6-й аргумент.

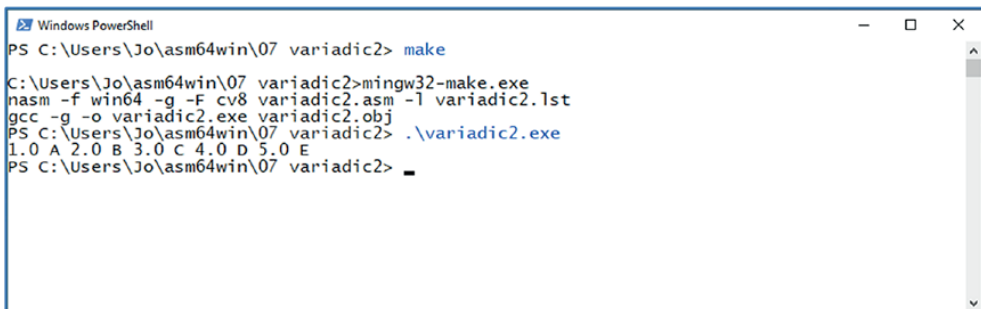
    push    B                ; 5-й аргумент.

; Вывод результата.
    sub     rsp,32
    call    printf
    add     rsp,32
leave
ret

```

Здесь можно видеть, что главное внимание необходимо сосредоточить на соблюдении правильного порядка передачи аргументов, а также на копировании `xmm`-регистров в регистры общего назначения и на записи в стек остальных аргументов в обратном порядке.

На рис. 42.3 показан вывод этой программы.



```

Windows PowerShell
PS C:\Users\Jo\asm64win\07 variadic2> make
C:\Users\Jo\asm64win\07 variadic2> mingw32-make.exe
nasm -f win64 -g -F cv8 variadic2.asm -l variadic2.lst
gcc -g -o variadic2.exe variadic2.obj
PS C:\Users\Jo\asm64win\07 variadic2> .\variadic2.exe
1.0 A 2.0 B 3.0 C 4.0 D 5.0 E
PS C:\Users\Jo\asm64win\07 variadic2>

```

Рис. 42.3. Вывод программы *variadic2.asm*

РЕЗЮМЕ

В этой главе вы узнали о том, что:

- значения с плавающей точкой в `xmm`-регистрах, соответствующие первым четырем аргументам, необходимо обязательно скопировать в соответствующие регистры общего назначения;
- если требуется передать более четырех аргументов с плавающей точкой или других типов, то они должны быть записаны в стек в обратном порядке.

Глава 43

Работа с файлами в Windows

В Linux для работы с файлами мы использовали системные вызовы (syscalls). В Windows необходимо соблюдать другие правила и требования. Как уже было отмечено в нескольких предыдущих главах, вместо системных вызовов используются функции Windows API.

В листинге 43.1 показан пример исходного кода.

Листинг 43.1. Программа *files.asm*

```
%include "win32n.inc"
extern printf
extern CreateFileA
extern WriteFile
extern SetFilePointer
extern ReadFile
extern CloseHandle

section .data
    msg      db 'Hello, Windows World!',0
    nNumberOfBytesToWrite equ $-msg
    filename db 'mytext.txt',0
    nNumberOfBytesToRead  equ 30
    fmt      db "The result of reading the file: %s",10,0

section .bss
    fHandle          resq 1
    lpNumberOfBytesWritten resq 1
    lpNumberOfBytesRead  resq 1
    readbuffer        resb 64

section .text
    global main
main:
    push    rbp
    mov     rbp,rsp

; HANDLE CreateFileA(
;   LPCSTR          lpFileName,
;   DWORD           dwDesiredAccess,
;   DWORD           dwShareMode,
;   LPSECURITY_ATTRIBUTES lpSecurityAttributes,
;   DWORD           dwCreationDisposition,
```

```
; DWORD          dwFlagsAndAttributes,
; HANDLE          hTemplateFile
; );
    sub     rsp,8
    lea     rcx,[filename]           ; Имя файла.
    mov     rdx, GENERIC_READ|GENERIC_WRITE ; Требуемые права доступа.
    mov     r8,0                     ; Совместное использование запрещено.
    mov     r9,0                     ; Защита по умолчанию.
; Запись в стек в обратном порядке.
    push    NULL                     ; Шаблон нет.
    push    FILE_ATTRIBUTE_NORMAL    ; Флаги и атрибуты.
    push    CREATE_ALWAYS             ; Диспозиция файла.
    sub     rsp,32                    ; Скрытое пространство в стеке.
    call    CreateFileA
    add     rsp,32+8
    mov     [fHandle],rax

; BOOL WriteFile(
; HANDLE hFile,
; LPCVOID lpBuffer,
; DWORD nNumberOfBytesToWrite,
; LPDWORD lpNumberOfBytesWritten,
; LPOVERLAPPED lpOverlapped
; );
    mov     rcx,[fHandle]             ; Идентификатор (дескриптор) файла.
    lea     rdx,[msg]                 ; Выводимое сообщение.
    mov     r8,nNumberOfBytesToWrite ; Число выводимых байтов.
    mov     r9,[lpNumberOfBytesWritten] ; Возврат числа записанных байтов.
    push    NULL
    sub     rsp,32                    ; Скрытое пространство в стеке.
    call    WriteFile
    add     rsp,32

; DWORD SetFilePointer(
; HANDLE hFile,
; LONG lDistanceToMove,
; PLONG lpDistanceToMoveHigh,
; DWORD dwMoveMethod
; );
    mov     rcx,[fHandle]             ; Идентификатор (дескриптор) файла.
    mov     rdx, 7                    ; Младшие биты позиции.
    mov     r8,0                      ; В позиции не используются старшие биты.
    mov     r9,FILE_BEGIN             ; Начать запись с начала файла.
    call    SetFilePointer

; BOOL ReadFile(
; HANDLE hFile,
; LPCVOID lpBuffer,
; DWORD nNumberOfBytesToRead,
; LPDWORD lpNumberOfBytesRead,
; LPOVERLAPPED
; );
    sub     rsp,8                     ; Выравнивание стека.
    mov     rcx,[fHandle]             ; Идентификатор (дескриптор) файла.
    lea     rdx,[readbuffer]          ; Буфер считывания.
    mov     r8,nNumberOfBytesToRead    ; Число считываемых байтов.
```

```

    mov     r9,[lpNumberOfBytesRead]    ; Число реально прочитанных байтов.
    push    NULL
    sub     rsp,32                      ; Скрытое пространство в стеке.
    call    ReadFile
    add     rsp,32+8

; Вывод результата выполнения функции ReadFile.
    mov     rcx, fmt
    mov     rdx, readbuffer
    sub     rsp,32+8
    call    printf
    add     rsp,32+8

; BOOL WINAPI CloseHandle(
;   _In_ HANDLE hObject
; );
    mov     rcx,[fHandle]
    sub     rsp,32+8
    call    CloseHandle
    add     rsp,32+8
leave
ret

```

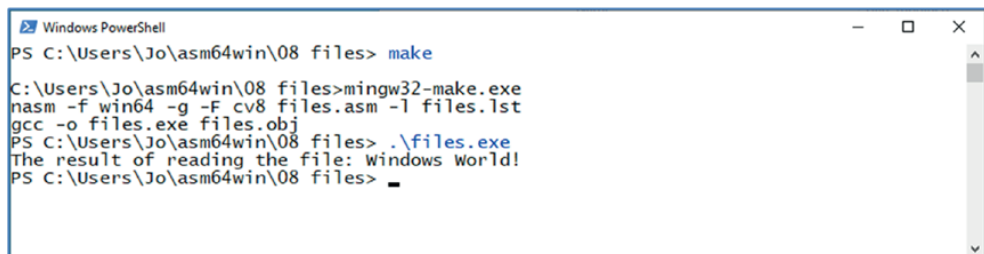
Как и ранее, мы просто используем шаблон на языке C для функции Windows API, чтобы сформировать собственные вызовы на ассемблере. Для создания файла используются простейшие параметры настройки (принятые по умолчанию) для доступа к файлу и его защиты. При успешном создании файла функция `CreateFileA` возвращает идентификатор (дескриптор) созданного файла. Обратите внимание на параметры. Подробное описание всех параметров можно найти в официальной документации Microsoft. Существуют некоторые возможности, которые могут помочь более точно управлять свойствами файла.

Идентификатор (дескриптор) файла используется в функции `WriteFile` для записи некоторого текста в файл. В главе 40 функция `WriteFile` уже использовалась для вывода сообщения в консоли.

После записи текста в файл нужно прочитать этот текст обратно в память, начиная с позиции 7, при этом первый байт имеет индекс 0. С помощью функции `SetFilePointer` указатель перемещается в позицию, с которой необходимо начать чтение. Если параметр `lpDistanceToMoveHigh` равен `NULL`, то параметр `lDistanceToMove` является 32-битовым значением, определяющим количество байтов смещения от начала файла. Иначе `lpDistanceToMove` и `lDistanceToMove` в совокупности формируют 64-битовое значение количества байтов смещения от начала файла. В регистре `r9` определяется позиция, с которой должен начаться отсчет перемещения, возможными вариантами являются `FILE_BEGIN`, `FILE_CURRENT` и `FILE_END`.

После установки указателя в требуемую позицию используется функция `ReadFile` для начала чтения с этой позиции. Считываемые байты сохраняются в буфере памяти, затем выводятся на экран. В завершающей части программы файл закрывается. Проверьте свой рабочий каталог, и вы увидите созданный текстовый файл *mytext.txt*.

На рис. 43.1 показан вывод этой программы.



```
Windows PowerShell
PS C:\Users\Jo\asm64win\08 files> make

C:\Users\Jo\asm64win\08 files> mingw32-make.exe
nasm -f win64 -g -F cv8 files.asm -l files.lst
gcc -o files.exe files.obj
PS C:\Users\Jo\asm64win\08 files> .\files.exe
The result of reading the file: windows world!
PS C:\Users\Jo\asm64win\08 files> _
```

Рис. 43.1. Вывод программы *files.asm*

РЕЗЮМЕ

В этой главе вы узнали о:

- средствах работы с файлами в Windows;
- существовании огромного количества параметров для точной настройки при обработке файлов.

Послесловие. Что дальше?

После изучения этой книги вы овладели основами программирования на современном языке ассемблера. Следующий шаг зависит от того, что вам нужно. В этом послесловии содержатся некоторые рекомендации.

Инженеры-аналитики, работающие в области обеспечения безопасности, могут использовать приобретенные знания для исследования вредоносного программного обеспечения, вирусов и прочих средств взлома компьютеров и сетей. Вредоносные программы в двоичном формате пытаются проникнуть в компьютеры и сети. Можно взять их двоичный код, выполнить реверс-инжиниринг и попробовать определить, что именно делает этот код. Разумеется, это следует делать в изолированной лабораторной системе. Изучайте средства реверс-инжиниринга и приобретайте необходимые инструментальные средства. Также следует отметить необходимость изучения ассемблера ARM для анализа кода для смартфонов.

Программист на одном из языков высокого уровня может рассмотреть возможность создания собственной библиотеки высокопроизводительных функций для связывания с кодом основной программы. Изучайте возможности оптимизации кода, ведь исходный код в этой книге написан не для достижения высокой производительности, а для демонстрации возможностей современного ассемблера. В этой книге приведено несколько ссылок на документы, которые помогут вам писать оптимизированный код.

Если вам необходимо более глубокое понимание работы процессоров Intel, то загрузите руководства Intel и внимательно изучайте их. В этих руководствах вы найдете огромный объем интересной информации, а знания о совместной работе аппаратного и программного обеспечения позволят выйти на передний край разработки системного программного обеспечения или инструментальных средств диагностики критических отказов систем.

Как программист на языке высокого уровня с хорошим знанием ассемблера вы гораздо лучше подготовлены к отладке собственного кода. Анализируйте объектные файлы (.obj) и файлы листинга (.lst) и применяйте реверс-инжиниринг, чтобы узнать, что происходит в коде программы. Наблюдайте, как используемый вами компилятор преобразовывает исходный код в машинный язык. Возможно, использование других инструкций ассемблера окажется более эффективным.

Предметный указатель

.gdbinit, файл конфигурации
отладчика GDB 39

А

Аргумент

- дополнительный 113
- извлечение из регистра 113
- извлечение из стека 113
- не являющийся значением
с плавающей точкой 113
- с плавающей точкой 114

Арифметическая инструкция 84

Ассемблер

- встроенный код
- простой 164
- расширенный 166
- синтаксис инструкций
встроенного кода 166

Ассемблерная функция 160

Б

Бит 30

В

Ввод/вывод в консоли

- пример использования 138

Внеочередное исполнение кода 290

Внешняя функция 99, 160

Внешняя функция из языка C 107

Выравнивание данных в памяти 186

Выравнивание по адресам памяти 29

Г

Гамильтона–Кэли теорема 268

Д

Данные

- выровненные 186
- невыровненные 186

Двоичное число 30

Десятичное представление чисел 30

Директива препроцессора ассемблера
135, 152

Дополнительный двоичный код
(twos complement) 31

Доступ к аргументам командной
строки 154

И

Инструкция перехода и значения
флагов 70

Инструкция проверки условия 69

Интегрированная среда разработки
21, 64

К

Калькулятор с функциями
преобразования форматов чисел 31

Л

Линкер 23

М

Макрокоманда 134

Макрос 134, 172

многострочный 135

однострочный 135

Маска, управляющая

перемешиванием 283

Матрица вырожденная

(сингулярная) 272

Машинный язык 28

Метка времени 290

О

Обратный порядок байтов

(от старшего к младшему big
endian) 52

Объектный файл 110

Операция с битами 124, 130

арифметическая 126

Оптимизация кода программы 292

Отладка программы 36

Отладка программы с аргументами
командной строки 155

Отладчик 36

с графическим пользовательским
интерфейсом 37

с интерфейсом командной строки 36

П

Память 73

структура в программе 79

Переменная

локальная 102

Переменная среды Windows 299

Перемешивание 227

байтов в слове 234

в обратном порядке 233

вращением 234

в случайном порядке 231

маска 231, 235

Переносимый язык ассемблера 19

Переполнение

обработка 140

Порядок следования байтов

(endianness) 52

Пролог функции 49, 54, 106

Простая функция, пример 99

Процессор 32

поддержка AVX 250

Прямой порядок байтов

(от младшего к старшему) 277

Прямой порядок байтов (от младшего к старшему little endian) 52

Р

Распространение знакового разряда 85

Распространение,

или расширение знака 125

Регистр 33

затираемый (clobbered) 165, 168

общего назначения 33

сохраняемый вызываемой функцией 119, 120

Регистровое ограничение 168

Редактор связей 23

С

Сдвиг

арифметический 84

Сериализация инструкций 290

Синтаксис ассемблера версии Intel 37

Системный вызов 144

Скалярные данные 185

Соглашение о вызовах функций 112

Microsoft x64 301

System V AMD64 ABI 112, 155

выравнивание стека по 16-байтовой границе 115

использование регистров 118

регистр

неразрушаемый, или
долговременный
(nonvolatile) 120

разрушаемый, или
кратковременный
(volatile) 120

Сравнение использования инструкции
цикла и инструкции условного
перехода 72

Стек 74

выравнивание по 16-байтовой

границе 103, 115, 310

скрытое пространство 301, 309

схема

пример использования 116

схема «последним пришел, первым
вышел» (last in first out LIFO) 90

фрейм 105

Страница памяти 215

Строка

анализ 174

обработка 170

определение длины 213

поиск 174, 216

реверсирование (изменение порядка
букв на обратный) 88

с длиной, не определенной явно 207

с неявно заданной длиной

сравнение 220

сравнение 174

с явно заданной длиной 207

сравнение 222

Счетчик

младших циклов процессора 290

старших циклов процессора 291

Т

Тип данных 25

строка 26

У

Указатель базы (стека) 104

Указатель стека 74

Упакованные данные 185

Условная инструкция 69

Условное ассемблирование 145, 152

Ф

Фаддеева–Леверье алгоритм 268

Файловый ввод/вывод 144

инструкция 152

Фрейм стека 105

Функция

ветвь 105

внешняя 99, 107

лист 105

с переменным числом
аргументов 317

Функция языка C 158

Ц

Центральное процессорное
устройство (ЦПУ) 32

Цикл 70

счетчик 71

Ч

Число

восьмеричное 145

с двойной точностью 94

с обычной (одинойной)
точностью 94

с плавающей точкой 32

знаковый бит 94

мантисса 95

показатель степени 94

пример программы 96

целое 31

беззнаковое 31

восьмеричное 32

знаковое 31

отрицательное 31

шестнадцатеричное 32

Э

Эпилог функции 49, 54, 106

А

addpd, инструкция 199, 202

addps, инструкция 199

addss, инструкция 97

add, инструкция 84

adouble.asm, файл функции 162

Advanced Vector Extension (AVX) 184

Advanced Vector Extensions (AVX) 250

alive.asm, программа 48

анализ 49

вывод 53

AND, логический оператор 58

ASCII 26, 40, 172

Assembler preprocessor directive 135, 152

asum.asm, файл функции 161

AVX 257

инструкция 292

производительность 286

маска, управляющая
перемешиванием 282

матрица

вывод 264

маска перестановок 271

маска смешивания (наложения) 269

обращение 268

операция 257

операция, пример исходного
кода 257

оценка производительности
вычисления следа 292

оценка производительности
транспонирования 286

транспонирование 274

транспонирование методом
перемешивания 282

транспонирование с
использованием неупакованных
данных 277

умножение 265

поддержка процессором 250

пример программы 252

В

Base pointer 104

betterloop.asm, программа 71

bitflags, переменная 132

blend_trace, функция 296

bss, Block Started by Symbol 26

btr, инструкция 130

bts, инструкция 130

bt, инструкция 130

С

Callee-saved register 119

Calling convention 112

circle.asm, файл, содержащий только
внешние функции 108

circle.asm, файл функции 160

cld, инструкция 177

cmpps, инструкция 174, 177, 178

cmp, инструкция 69, 182

Conditional assembly 145, 152

CountReg 174

cpuid, инструкция 179, 290

CreateFileA, функция Windows API 323

createFile, функция 152

cvtss2sd, инструкция 199, 202

Стек 88

D

Data Display Debugger (DDD),
отладчик 60
DDD, отладчик 91
 использование команд
 отладчика GDB 62
 исследование памяти 62
DDD, отладчик
 наблюдение за изменением
 регистров 62
dec, инструкция 84
DF, флаг направления 173, 177
divsd, инструкция 195
divss, инструкция 97
Double precision 94
dwarf, формат отладки 23

E

eflags, регистр флагов в SASM 69
elf64, Executable and Linkable Format
for 64-bit 23
ELF, формат выполняемого файла
поддержка в NASM 76
extern, ключевое слово, объявление
внешней функции 108
extractps, инструкция 255

F

FPU-инструкция 95
function2.asm,
 программа 100
function4.asm, программа 107
function.asm, программа 99

G

GCC, GNU Compiler Collection 21
gdb memory, команда 76
GDB, отладчик 36, 155, 302
 информация о регистрах 42
 команда
 tui enable 45
 прерывание работы программы 42
 процесс отладки программы 37
gedit, текстовый редактор 20
GetStdHandle, функция Windows
 API 306
global, ключевое слово,
 объявление глобальной
 переменной 108
grep, утилита 77

H

haddpd, инструкция 270
handling port 5 pressure, в руководстве
 Intel 292
hello world, программа 19
 улучшенная версия 45

I

icalc.asm, программа 81
IDE, integrated development
 environment 21
idiv, инструкция 87
IEEE-754, международный стандарт 32
imm8, управляющий байт для строк
 SSE 208
 IntRes1 209
 IntRes2 210
 использование битов 209
imul, инструкция 85
inc, инструкция 84

J

jge, инструкция 69
jmp, инструкция 70
jnz, инструкция 177
jump.asm, программа 66
jumploop.asm, программа 70
jz, инструкция 177, 218

L

ldmxcsr, инструкция 189, 195
leave, инструкция 100
lea, инструкция 75, 173
lodsrb, инструкция 178
loop, инструкция 71, 90

M

makefile 22
 расширенная версия 110
MASM 298
memory.asm, программа 73
MinGW (Minimalist GNU for
 Windows) 299
MinGW-w64 299
MMX 184
movapd, инструкция 202
movaps, инструкция 202
movdqa, инструкция 206
movdqu, инструкция 231
move.asm, программа 60
movq, инструкция 55

movsb, инструкция 173, 174
 movsd, инструкция 97, 174
 movss, инструкция 97, 199
 movsw, инструкция 174
 movupd, инструкция 199
 movups, инструкция 199
 mov, инструкция 27, 75
 mulss, инструкция 97
 mul, инструкция 85
 mxcsr, регистр 271
 mxcsr, регистр управления и состояния
 SSE 188, 195
 значения битов 188
 MXCSR, регистр флагов для SIMD 35

N

NASM 21, 134, 152, 202, 298
 nasm -v, команда 298
 NASM, версия ассемблера 20
 neg, инструкция 58
 нор, инструкция 40
 NOT, логический оператор 57

O

objdump, утилита командной строки
 136
 OR, логический оператор 57

P

paddd, инструкция 206
 PATH, переменная среды Windows 298
 pcharsrch, функция 241
 pcmpestri, инструкция 207, 225
 pcmpestrm, инструкция 207, 237
 pcmpistri, инструкция 207, 214, 218,
 221, 241
 pcmpistrm, инструкция 207, 237, 241
 pextrd, инструкция 206
 PIE, position independent executable,
 перемещаемый код 24
 pinsrb, инструкция 218, 241
 pinsrd, инструкция 206, 231
 pmovmskb, инструкция 242
 popcnt, инструкция 242
 popf, инструкция 91
 pop, инструкция 88
 PowerShell 299
 printb, функция 121
 printdppf, функция 199
 printf, функция 53, 86, 99, 108, 111,
 112, 115

printf, функция 199, 202
 printString, функция 152
 prints, функция 142
 print_xmm, функция 195
 pshufb, инструкция 235
 pshufd, инструкция 234, 282
 pshufhw, инструкция 234
 pshufw, инструкция 234
 pstrcmp, функция 221, 225
 strlen, функция 214
 ptrln, функция 241
 ptrscan_f, функция 218
 ptrscan_l, функция 218
 pushf, инструкция 91
 push, инструкция 88
 xor, инструкция 214, 218

R

rax, регистр 27, 63, 75, 85, 100, 125
 rbp, регистр указателя базы стека 104
 rbx, регистр 71
 rdi, регистр 28, 152
 rdtsc, инструкция 291
 rdtsc, инструкция 290
 rdx, регистр 28
 readelf, утилита 76
 ReadFile, функция Windows API 323
 reads, функция 142
 rect.asm, файл функции 160
 Register constraint 168
 repe, инструкция 178
 repne, инструкция 178
 reverse_xmm0, функция 241
 rflags, регистр флагов процессора 34, 69
 rip, регистр 43
 rip, регистр счетчика команд 34
 rol, инструкция 126
 ror, инструкция 126
 rsp, регистр указателя стека 74, 88, 93

S

sal, инструкция 84, 125
 sar, инструкция 85, 125
 SASM 36, 64, 143, 299, 302
 диалоговая панель Settings
 (Параметры настройки) 65
 строка Linking Options (Параметры
 связывания) 111
 scasb, инструкция 174, 178
 section .bss, раздел программы 27

section .data, раздел программы 25
 section .txt, раздел программы 27
 seq_trace, функция 296
 seq_transpose, функция 291
 setCC, группа инструкций 132
 setc, инструкция 132
 SetFilePointer, функция Windows API 323
 setnz, инструкция 264
 Shadow space 301, 309
 shl, инструкция 125
 shr, инструкция 125
 Shuffling 227
 Sign extension 85, 125
 SIMD 184
 Simple ASM. См. SASM; См. SASM
 Single precision 94
 sqrtss, инструкция 97
 sreverse.asm, файл функции 161
 SSE 197
 выровненные данные 200
 инструкция
 для работы с целыми числами 204
 маска строки 237
 поиск подстроки 246
 поиск символа 237
 поиск символов из заданного
 диапазона 243
 невывровненные данные 197
 обработка строк 207
 агрегация 209
 использование флагов 211
 сравнение 209
 SSE-инструкция 179
 STABS, формат отладки 23
 Stack 88
 stack.asm, программа 88
 Stack frame 105
 stmxcsr, инструкция 189, 195
 stosb, инструкция 172, 174
 stosd, инструкция 174
 stosw, инструкция 174
 Streaming SIMD Extension (SSE) 184
 subss, инструкция 97
 sub, инструкция 84

T
 tail, утилита 78
 testfile.txt, тестовый файл 152
 test, инструкция 181, 264
 time, команда Linux 72

V

Variadic function 317
 vblendpd, инструкция 269
 vdivsd, инструкция 271
 vextractf128, инструкция 255
 vfmadd213sd, инструкция 271
 vfmadd231pd, инструкция 265
 vfmadd231sd, инструкция 271
 vhaddpd, инструкция 270
 Visual Studio 169, 298
 vmovups, инструкция 255
 vperm2f128, инструкция 279, 285
 vpermpd, инструкция 270
 vshufpd, инструкция 282
 vtrace, функция 269
 vunpckhpd, инструкция 278
 vunpcklpd, инструкция 278
 vxorpd, инструкция 271
 vzeroall, инструкция 265

W

Windows 298
 системный вызов 302
 функция
 аргумент с плавающей точкой 314
 аргументы, не являющиеся
 значениями с плавающей
 точкой 309
 передача аргументов различных
 типов 319
 с переменным числом
 аргументов 317
 Windows API 304, 321
 вывод в консоли 304
 создание окна 307
 WriteConsoleA, функция Windows
 API 306
 WriteConsole, функция Windows API 306
 WriteFile, функция Windows API 306, 323

X

xmm-регистр 227, 315, 320
 xmm, регистр 35
 XOR, логический оператор 58

Y

ymm-регистр 250, 255, 278, 284
 ymm, регистр 35, 266

Книги издательства «ДМК Пресс» можно заказать
в торгово-издательском холдинге «Планета Альянс» наложенным платежом,
выслав открытку или письмо по почтовому адресу:

115487, г. Москва, 2-й Нагатинский пр-д, д. 6А.

При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги;
фамилию, имя и отчество получателя.

Желательно также указать свой телефон и электронный адрес.

Эти книги вы можете заказать и в интернет-магазине: www.a-planet.ru.

Оптовые закупки: тел. (499) 782-38-89.

Электронный адрес: books@aliants-kniga.ru.

Йо Ван Гуй

Программирование на ассемблере x64: от начального уровня до профессионального использования AVX

Главный редактор *Мовчан Д. А.*

dmkpress@gmail.com

Зам. главного редактора *Сенченкова Е. А.*

Перевод *Снастин А. В.*

Корректор *Синяева Г. И.*

Верстка *Луценко С. В.*

Дизайн обложки *Мовчан А. Г.*

Формат 70×100 1/16.

Гарнитура «PT Serif». Печать цифровая.

Усл. печ. л. 26,98. Тираж 200 экз.

Веб-сайт издательства: www.dmkpress.com