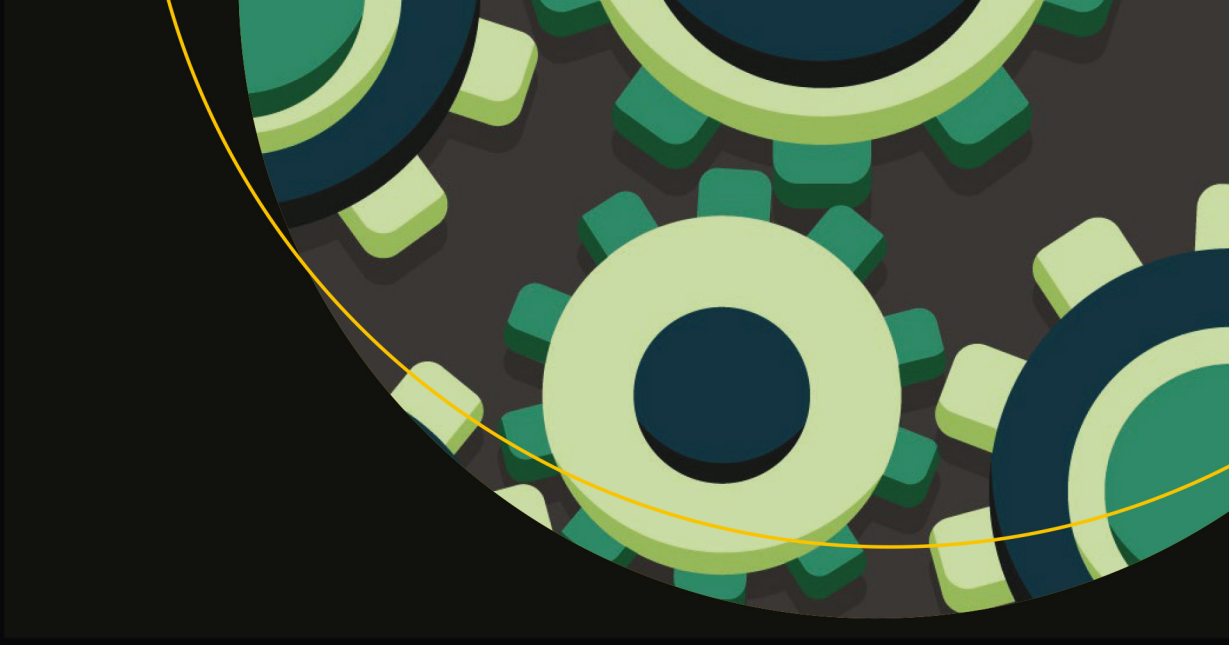


Программирование на ассемблере x64 для начинающих

Йо Ван Гуй

ДМК Пресс
издательство

УДК 004.4
ББК 32.97
Г93

Йо Ван Гуй
Г93 Программирование на ассемблере x64 для начинающих / пер. с анг. 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 потребуется знание основы программирования на каком-либо языке высокого уровня, например C.

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 (анг.) 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 `cpuid` ... 179
Использование инструкции `test` ... 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 mxcsr ... 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 `imm8` 208
Использование управляющего байта `imm8` 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: `printm4x4` 264
Умножение матриц: `multi4x4` 265
Обращение матрицы: `inverse4x4` 268
Теорема Гамильтона–Кэли 268
Алгоритм Фаддеева–Леверье 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

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 C»). Керниган принимал участие в разработке языка программирования C в компании Bell Labs. С тех пор язык C 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 C. Язык C иногда называют переносимым языком ассемблера, и как программист, стремящийся к овладению ассемблером, вы должны знать C. 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 Linux[1], – 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.



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 Format[2] (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 **Tab**le **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.*

сматривает 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`, `B` 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
`resb`	8 бит	Байт
`resw`	16 бит	Слово
`resd`	32 бита	Двойное слово
`resq`	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 B801000000                  mov     rax, 1              ; 1 = write
 9     9 00000005 BF01000000                  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 0000001B B83C000000                  mov     rax, 60             ; 60 = exit
15    15 00000020 BF00000000                  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 rax` преобразована в `B8`, а `mov rdi` 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.