ЧАСТЬ I

Внутреннее устройство архитектуры Arm

Если вы только что взяли эту книгу с полки, то, скорее всего, вам интересно узнать, как проводить реверс-инжиниринг скомпилированных двоичных файлов Arm, поскольку основные производители техники сейчас используют архитектуру Arm. Возможно, вы уже опытный ветеран реверс-инжиниринга x86-64, но хотите не отставать и узнать больше об архитектуре, начавшей захват рынка процессоров. Возможно, вы хотите приступить к анализу безопасности, чтобы найти уязвимости в программном обеспечении на базе Arm или проанализировать вредоносное ПО на базе Arm. Или, может, вы только начинаете заниматься реверс-инжинирингом и дошли до момента, когда для достижения цели требуется более глубокий уровень детализации.

Где бы вы ни находились на пути во вселенную реверс-инжиниринга на базе Arm, эта книга подготовит вас, читателя, к пониманию языка двоичных файлов Arm, покажет, как их анализировать, и, что более важно, подготовит вас к будущему устройств Arm.

Изучение языка ассемблера и умение анализировать скомпилированные программы пригодятся в самых разных областях. Как и любой другой навык, изучение синтаксиса поначалу может показаться трудным и сложным, но с практикой все становится проще.

В части I мы рассмотрим основы архитектуры Cortex-A компании Arm, в частности Armv8-A, и основные инструкции, с которыми вы столкнетесь при выполнении реверс-инжиниринга программного обеспечения, скомпилированного для этой платформы. В части II рассмотрим некоторые распространенные инструменты и методы реверс-инжиниринга. Чтобы вдохновить вас на различные применения реверс-инжиниринга на базе Arm, мы разберем практические примеры, в том числе анализ вредоносных программ, скомпилированных для чипа M1 от Apple.

ГЛАВА 1
Введение в реверс-инжиниринг

ВВЕДЕНИЕ В АССЕМБЛЕР

Если вы читаете эту книгу, то, скорее всего, уже слышали о такой штуке, как *язык ассемблера Arm*, и знаете, что его понимание — ключ к анализу двоичных файлов, работающих на архитектуре Arm[1]. Но что это за язык и почему он существует? Программисты обычно пишут код на языках высокого уровня, таких как C/C++, и почти никто не программирует на ассемблере напрямую. Языки высокого уровня, в конце концов, гораздо удобнее для них.

К сожалению, эти языки высокого уровня слишком сложны, чтобы процессоры могли интерпретировать их напрямую. Вместо этого программисты *компилируют* программы высокого уровня в двоичный машинный код, который может выполнять процессор.

Этот машинный код не совсем то же самое, что язык ассемблера. Если бы вы посмотрели на него прямо в текстовом редакторе, он выглядел бы непонятно. Процессоры также не используют язык ассемблера — они воспринимают только машинный код. Почему же он так важен для реверс-инжиниринга?

Чтобы понять назначение ассемблера, совершим краткий экскурс в историю вычислительной техники, чтобы понять, как мы оказались там, где находимся, и как все между собой связано.

Биты и байты

В начале времен люди решили создать компьютеры и заставить их выполнять простые задачи. Компьютеры не говорят на человеческих языках — в конце концов, это всего лишь электронные устройства, поэтому людям нужен был способ общаться с ними в электронном виде. На самом низком уровне компьютеры работают с электрическими сигналами, а они формируются путем переключения электрического напряжения между одним из двух уровней: включено и выключено.

Первая проблема заключается в том, что нам нужен способ описать эти включения и выключения для связи, хранения и просто описания состояния системы. Поскольку существуют два состояния, было вполне естественно использовать

---

[1] В книге мы используем название архитектуры, как у автора. — *Примеч. ред.*

двоичную систему для кодирования этих значений. Каждый двоичный разряд, или *бит*, может быть 0 или 1. Хотя каждый бит может хранить лишь минимально возможное количество информации, объединение нескольких битов позволяет представлять гораздо большие числа. Например, число 30 284 334 537 может быть представлено в 35 битах следующим образом:

```
11100001101000101100100010111001001
```

Такая система уже позволяет кодировать большие числа, но теперь у нас появилась новая проблема: где заканчивается одно число в памяти (или на магнитной ленте) и начинается следующее? Возможно, это странный вопрос для современного читателя, но в те времена, когда компьютеры только создавались, это было серьезной проблемой. Самым простым решением здесь будет создание групп битов фиксированного размера. Ученые-компьютерщики назвали эту группу двоичных цифр или битов *байтом*.

Итак, сколько битов должно быть в байте? Для современного уха это может показаться очевидным вопросом, ведь мы все знаем, что современный байт состоит из 8 битов. Но так было не всегда.

Первоначально различные системы по-разному определяли количество битов в байтах. Предшественником восьмибитного байта, известного нам сегодня, был шестибитный формат двоично-десятичного кода обмена (Binary Coded Decimal Interchange Code, BCDIC) для представления алфавитно-цифровой информации. Он использовался в ранних компьютерах IBM, таких как IBM 1620, разработанный в 1959 году. До этого длина байтов часто была 4 бита, а еще раньше байт обозначал произвольное количество битов, большее 1. Лишь позднее, в 1960-х годах, появился восьмибитный расширенный двоично-десятичный код обмена (Extended Binary Coded Decimal Interchange Code, EBCDIC) компании IBM. Он применялся в линейке компьютеров-мейнфреймов System/360 и имел восьмибитную память с байтовой адресацией. Байт начал стандартизироваться и содержать 8 битов. Это привело к тому, что восьмибитный размер памяти был принят и в других широко распространенных компьютерных системах, включая Intel 8080 и Motorola 6800.

Следующий отрывок взят из книги *Planning a Computer System*, опубликованной в 1962 году, в нем перечислены три основные причины перехода на восьмибитный байт[1].

1. *«Его полная емкость в 256 символов считалась достаточной для подавляющего большинства приложений.*
2. *В пределах этой емкости один символ представлен одним байтом, так что длина любой конкретной записи не зависит от совпадения символов в этой записи.*
3. *Восьмибитные байты достаточно экономно расходуют пространство для хранения данных».*

---

[1] *Planning a Computer System, Project Stretch.* — McGraw — компания Hill Book Company, Inc., 1962 (http://archive.computerhistory.org/resources/text/IBM/Stretch/pdfs/Buchholz_102636426.pdf).

Восьмибитный байт может содержать одно из 256 уникально различных значений от 00000000 до 11111111. Их интерпретация, конечно, зависит от использующего их программного обеспечения. Например, можно хранить в этих байтах положительные числа, чтобы представить положительное число от 0 до 255 включительно. Можно также применить схему двойного дополнения для представления *знаковых* чисел от –128 до 127 включительно.

Кодировка символов

Конечно, компьютеры задействовали байты не только для кодирования и обработки целых чисел. В них также часто хранятся и обрабатываются человекочитаемые буквы и цифры, называемые *символами*.

Ранние кодировки символов, такие как ASCII, остановились на использовании 7 битов на байт, но это давало лишь ограниченный набор из 128 возможных символов. Такой подход позволял кодировать англоязычные буквы и цифры, а также несколько символов и управляющих знаков, но отображать множество букв, применяемых в других языках, возможности уже не было. Стандарт EBCDIC, использующий восьмибитовые байты, выбрал совершенно другой набор символов, с кодовыми страницами для переключения на разные языки. Но в конечном счете он оказался слишком громоздким и негибким.

Со временем стало ясно, что нужен действительно универсальный набор символов, поддерживающий все существующие в мире языки и специальные символы. Кульминацией разработки в этом направлении стало создание проекта Unicode в 1987 году. Существует несколько различных кодировок Юникода, но доминирующей, используемой в Интернете, является UTF-8. Символы, входящие в набор символов ASCII, включены в UTF-8 дословно, а расширенные символы могут распространяться на несколько последовательных байтов.

Поскольку символы теперь кодируются как байты, можно представлять символы с помощью двух шестнадцатеричных цифр. Например, символы *A*, *R* и *M* обычно кодируются октетами, показанными на рис. 1.1.

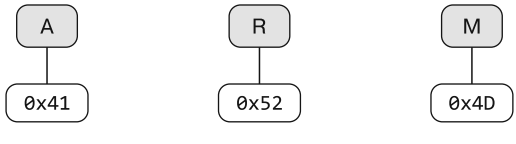

Рис. 1.1. Буквы A, R и M и их шестнадцатеричные значения

Каждая шестнадцатеричная цифра может быть закодирована четырехбитным шаблоном в диапазоне от 0000 до 1111 (рис. 1.2).

Поскольку для кодирования символа ASCII требуются два шестнадцатеричных значения, идеальным вариантом для хранения текста на большинстве письменных языков мира были 8 битов или значение, кратное 8 битам, для символов, которые не могут быть представлены только 8 битами.

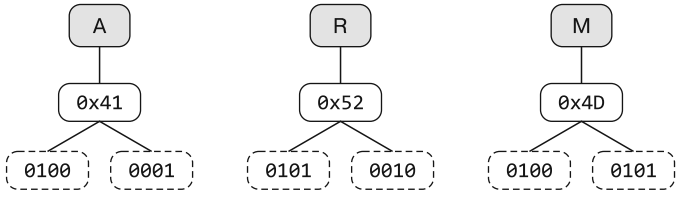

Рис. 1.2. Шестнадцатеричные значения ASCII и их восьмибитные двоичные эквиваленты

Используя эту схему, легче интерпретировать значение длинной строки битов. Следующий битовый шаблон кодирует слово *Arm*:

```
0100 0001 0101 0010 0100 1101
```

Машинный код и ассемблер

Уникальность компьютеров, в отличие от предшествовавших им механических калькуляторов, заключается в том, что они могут кодировать свою *логику* в виде данных. Этот *код* может храниться в памяти или на диске, его можно обрабатывать или изменять по требованию. Например, обновление программного обеспечения способно полностью изменить операционную систему компьютера и не потребуется покупать новую машину.

Мы уже видели, как кодируются числа и символы, но как кодируется логика? Именно здесь в игру вступают архитектура процессора и его набор инструкций.

Если бы мы создали собственный компьютерный процессор с нуля, можно было бы разработать собственную *кодировку инструкций*. Это позволило бы отобразить двоичные шаблоны в машинные коды, которые наш процессор сможет интерпретировать и на которые будет отвечать, фактически создав собственный машинный язык. Поскольку машинные коды предназначены для инструктирования схемы на выполнение операций, они называются также *кодами инструкций* или чаще *кодами операций* (*операционными кодами*, или *опкодами*).

На практике большинство людей используют существующие компьютерные процессоры и, соответственно, кодировки инструкций, определенные производителем процессора. В архитектуре Arm у кодировок инструкций фиксированный размер, они могут быть либо 32-, либо 16-битными в зависимости от набора инструкций, применяемого программой. Процессор получает и интерпретирует каждую инструкцию и выполняет их по очереди, чтобы выполнить логику программы. Каждая инструкция представляет собой двоичный шаблон, или *кодировку инструкции*, следующую конкретным заданным архитектурой Arm правилам.

В качестве примера предположим, что мы создаем крошечный 16-битный набор инструкций и определяем, как будет выглядеть каждая инструкция. Первая задача — обозначить часть кодировки, называемую опкодом и указывающую, какой именно тип инструкции должен быть выполнен. Например, можно установить первые 7 битов инструкции в качестве опкода и указать опкоды для сложения и вычитания (табл. 1.1).

Таблица 1.1. Операционные коды сложения и вычитания

Операция	Опкод
Сложение	0001110
Вычитание	0001111

Написание машинного кода вручную возможно, но излишне громоздко. На практике лучше писать на каком-нибудь человекочитаемом языке ассемблера, преобразуемом в эквивалент машинного кода. Для этого необходимо также определить сокращенное обозначение инструкции, называемое *мнемоникой* инструкции (табл. 1.2).

Таблица 1.2. Мнемоники

Операция	Опкод	Мнемоника
Сложение (addition)	0001110	ADD
Вычитание (subtraction)	0001111	SUB

Конечно, недостаточно сказать процессору, чтобы он просто выполнил сложение. Необходимо также указать ему, какие два элемента нужно сложить и что делать с результатом. Например, если мы пишем программу, выполняющую команду `a = b + c`, то значения `b` и `c` должны быть где-то сохранены до начала выполнения инструкции, а последняя должна знать, куда записать результат `a`.

В большинстве процессоров, в частности в процессорах Arm, эти временные значения обычно хранятся в *регистрах*, предназначенных для хранения небольшого количества рабочих значений. Программы могут забирать данные из памяти (или с диска) в регистры, готовые к обработке, и после обработки возвращать результаты в долговременное хранилище.

Количество и названия регистров зависят от архитектуры. Поскольку программное обеспечение становится все более и более сложным, программы часто должны жонглировать большим количеством значений одновременно. Хранение этих значений и работа с ними в регистрах происходят быстрее, чем непосредственно в памяти, а это значит, что регистры сокращают количество обращений программы к памяти и ускоряют ее выполнение.

Вернемся к нашему примеру — мы разрабатывали 16-битную инструкцию для выполнения операции, складывающей значение со значением в регистре и записывающей результат в другой регистр. Поскольку 7 битов используются для самой операции (`ADD`/`SUB`), оставшиеся 9 битов можно применять для кодирования регистров источника и назначения, а также постоянного значения, которое мы хотим сложить или вычесть. В этом примере разделим оставшиеся биты поровну и назначим им обозначение и соответствующие машинные коды (табл. 1.3).

Таблица 1.3. Назначение машинных кодов вручную

Операция	Мнемоника	Машинный код
Сложение	ADD	0001110
Вычитание	SUB	0001111
Целочисленное значение 2	2	010
Регистр операнда	R0	000
Регистр назначения	R1	001

Вместо того чтобы генерировать машинные коды вручную, можно написать небольшую программу, преобразующую синтаксис `ADD R1, R0, #2` (R1 = R0 + 2) в соответствующий шаблон машинного кода и передающую его процессору из примера (табл. 1.4).

Таблица 1.4. Программирование машинных кодов

Инструкция	Двоичный машинный код	Шестнадцатеричное кодирование
ADD R1, R0, #2	0001110 010 000 001	0x1C81
SUB R1, R0, #2	0001111 010 000 001	0x1E81

Созданный нами битовый шаблон представляет собой одну из кодировок 16-битных инструкций ADD и SUB, входящих в набор инструкций T32. На рис. 1.3 показаны его компоненты и то, как они упорядочены в кодировке инструкции.

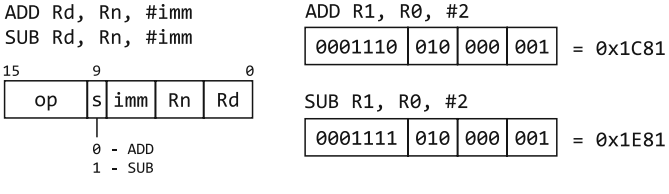

Рис. 1.3. 16-битное Thumb-кодирование мгновенных инструкций ADD и SUB

Конечно, это упрощенный пример. Современные процессоры предоставляют сотни возможных инструкций, часто с более сложными субкодировками. Например, Arm определяет инструкцию загрузки регистра (с мнемоникой LDR), загружающую 32-битное значение из памяти в регистр (рис. 1.4). В этой инструкции адрес для загрузки указывается в регистре 2 (называется R2), а считанное значение записывается в регистр 3 (называется R3).

Синтаксис написания скобок вокруг R2 указывает на то, что значение в R2 должно интерпретироваться как адрес в памяти, а не как обычное значение. Другими

словами, мы не хотим копировать значение, находящееся в R2, в регистр R3, а хотим получить содержимое памяти по *адресу*, заданному R2, и загрузить это значение в регистр R3. Существует множество причин, по которым программа может ссылаться на ячейку памяти, включая вызов функции или загрузку значения из памяти в регистр.

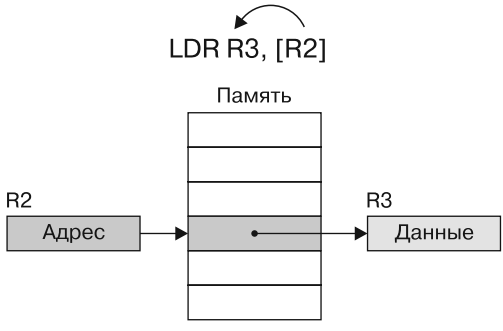

Рис. 1.4. Инструкция LDR загружает значение из адреса, взятого в R2, в регистр R3

В этом, по сути, и заключается разница между машинным и ассемблерным кодом. Язык ассемблера — это понятный человеку синтаксис, показывающий, как следует интерпретировать каждую закодированную инструкцию. Машинный код, напротив, это фактические двоичные данные, принимаемые и обрабатываемые реальным процессором, кодировка которых точно определена его разработчиком.

Написание кода ассемблера

Поскольку процессоры понимают только машинный код, а не язык ассемблера, как преобразовать одно в другое? Нам нужна программа для преобразования рукописных ассемблерных инструкций в их машинные эквиваленты. Программы, выполняющие эту задачу, называются *ассемблерами*.

На практике ассемблеры способны не только понимать и преобразовывать отдельные инструкции в машинный код, но и интерпретировать *директивы ассемблера*[1], которые предписывают ему выполнять другие действия, например переключаться между данными и кодом или собирать различные наборы инструкций. Синтаксис и значение отдельных директив и выражений ассемблера зависят от конкретного ассемблера.

Эти директивы и выражения являются полезными сокращениями. Их можно использовать в программе на ассемблере, однако они не являются частью самого языка ассемблера, скорее представляют собой указания, как он должен работать.

Для разных платформ доступны разные ассемблеры, например ассемблер GNU `as`, применяемый для сборки ядра Linux, ассемблер ARM Toolchain `armasm` или одноименный ассемблер Microsoft (`armasm`), входящий в состав Visual Studio.

---

[1] https://ftp.gnu.org/old-gnu/Manuals/gas-2.9.1/html_chapter/as_7.html.

Предположим, что требуется собрать следующие две 16-битные инструкции, записанные в файле `myasm.s`:

```
.section .text
.global _start
_start:
.thumb
    movs r1, #5
    ldr  r3, [r2]
```

В этой программе первые три строки — это директивы ассемблера. Они сообщают ему, куда должны быть собраны данные (в данном случае в секцию `.text`), определяют метку точки входа нашего кода (в данном случае `_start`) как глобальный символ и, наконец, указывают, что должна использоваться кодировка инструкций под названием Thumb. Набор инструкций Thumb (T32) является частью архитектуры Arm и позволяет использовать инструкции размером 16 битов.

Можно использовать ассемблер GNU, `as`, чтобы скомпилировать эту программу на машине с операционной системой Linux, работающей на процессоре Arm:

```
$ as myasm.s -o myasm.o
```

Ассемблер считывает программу на языке ассемблера `myasm.s` и создает объектный файл `myasm.o`. Он содержит 4 байта машинного кода, соответствующие нашим двум двухбайтовым инструкциям в шестнадцатеричном формате:

```
05 10 a0 e3 00 30 92 e5
```

Еще одной полезной особенностью ассемблеров является *метка* — она ссылается на определенный адрес в памяти, например на адрес цели перехода, функции или глобальной переменной.

В качестве примера рассмотрим программу на ассемблере:

```
.section .text
.global _start

_start:
        mov r1, #5
        mov r2, #6
        b mylabel
result:
        mov r0, r4
        b _exit
mylabel:
        add r4, r1, r2
        b result

_exit:
        mov r7, #0
        svc #0
```

Эта программа начинается с заполнения двух регистров значениями и перехода, или перескакивания, на метку `mylabel` для выполнения инструкции `ADD`. После

выполнения инструкции ADD программа переходит на метку `result`, выполняет инструкцию перемещения (`mov`) и завершается переходом на метку `_exit`. Ассемблер будет использовать эти метки для подсказок компоновщику, назначающему им относительные места в памяти. На рис. 1.5 показан поток выполнения программы.

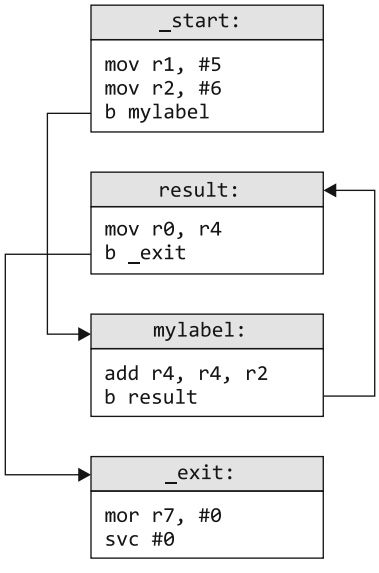

Рис. 1.5. Программный поток примера программы на ассемблере

Метки полезны не только для ссылки на инструкции для перехода, но также могут использоваться для извлечения содержимого ячейки памяти. Например, в следующем фрагменте ассемблерного кода метки применяются для извлечения содержимого из ячейки памяти или перехода к различным инструкциям в коде:

```
.section .text
.global _start

_start:
    mov r1, #5          // 1. Заполнение r1 значением 5
    adr r2, myvalue     // 2. Заполнение r2 адресом myvalue
    ldr  r3, [r2]       // 3. Заполнение r3 значением по адресу, указанному в r2
    b mylabel           // 4. Переход к адресу mylabel
result:
    mov r0, r4          // 7. Заполнение r0 значением  из регистра r4
    b _exit             // 8. Переход к адресу метки _exit
mylabel:
    add r4, r1, r3      // 5. Заполнение r4 результатом r1 + r3
    b result            // 6. Переход к метке результата – result

myvalue:
.word 2                 // Значение размером со слово, содержащее значение 2
```

Инструкция ADR загружает адрес переменной *myvalue* в регистр R2 и использует инструкцию LDR для загрузки содержимого этого адреса в регистр R3. Затем программа переходит к инструкции, на которую ссылается метка *mylabel*, выполняет инструкцию ADD и переходит к инструкции, на которую ссылается метка *result* (рис. 1.6).

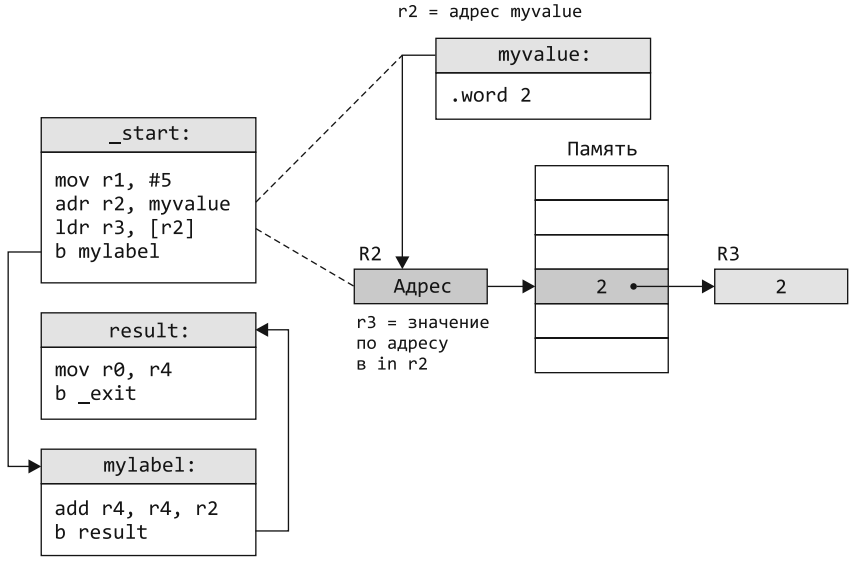

Рис. 1.6. Иллюстрация логики команд ADR и LDR

В качестве чуть более интересного примера можно привести следующий ассемблерный код, выводящий надпись Hello World! в консоли, а после завершающий работу. Он использует метку для ссылки на строку hello, поместив относительный адрес своей метки mystring в регистр R1 с помощью инструкции ADR:

```
.section .text
.global _start

_start:
    mov r0, #1              // STDOUT
    adr r1, mystring        // R1 = адрес строки
    mov r2, #6              // R2 = размер строки
    mov r7, #4              // R7 = номер системного вызова функции 'write()'
    svc #0                  // Запуск системного вызова

_exit:
    mov r7, #0
    svc #0

mystring:
.string "Hello\n"
```