# 第6章 ARM汇编相关的iOS逆向理论基础

# Chapter 6 ARM related iOS reverse engineering

前面的章节中介绍了iOS逆向工程的基础知识，包括一些常见工具的组合使用，在掌握了这些知识之后，简单地把玩一下Objective-C私有函数，满足一下自己的好奇心已经没问题了，可以针对App开发tweak了。但是，既然看到了这里，相信大家都具有比较强的钻研精神，如果想要真正提高自己的能力，就要尝试一些更有挑战性的内容。那么，从这一章开始，iOS逆向工程就将进入“极昼”，我们将零距离接触编程世界里最让人头大的知识。请先深呼吸一分钟，然后问问自己：“我是否真的适合深入学习iOS逆向工程？”在完成本章之后，相信你会得到答案。

In previous chapters we have already introduced the fundamental knowledge and tool usage in iOS reverse engineering. Now, you should be able to satisfy your curiosity by playing with private methods and develop some mini tweaks. However, since you’ve come so far, I believe you have a strong delving spirit and truly want to improve your development ability. If so, it’d be better for you to try something more challenging. Well, starting from this chapter, iOS reverse engineering will enter polar night, and you’ll have to face the most arcane yet magical hieroglyphics in the programming world. Take a deep breath first, and then ask yourself, “Is iOS reverse engineering a right choice for me?” After finishing this chapter, hopefully you will get the answer.

下面即将面对iOS逆向工程中的第一个进阶难点：阅读ARM汇编语言。经过前几章的学习，相信大家已经知道，Objective-C代码在经过编译后形成机器码，它们由设备的CPU直接执行。别说编写，阅读机器码都已经是一个非常恼人的工作；好在Objective-C和机器码之间有汇编语言这座桥，它的可读性虽然远不如Objective-C，但比机器码要强多了——如果你能够啃下这块硬骨头，那么恭喜你，你有着成为逆向工程师的天份；如果你在啃骨头的时候牙被崩掉了，或许AppStore开发才是你更好的归宿……

Next, we’ll meet the first advanced challenge in iOS reverse engineering: reading ARM assembly. According to the previous chapters, you have already got the idea that Objective-C code would become machine code after compiling, and then will be executed directly by CPU. It is overwhelming work to read machine code let alone write them. However, it’s lucky that there is assembly, which bridges Objective-C code with machine code. Even though the readability of assembly is not as good as Objective-C, it’s much better than machine code. If you can crash this hard nut, congratulations, you have the talents to be a reverse engineer. Conversely, if you cannot, AppStore may suit you better.

## 6.1 ARM汇编基础

## 6.1 Introduction to ARM assembly

对于很多iOS开发者来说，ARM汇编是一门全新的语言；如果你是计算机专业科班出身，应该已经对汇编语言有了初步的印象，只是对于很多人来说，大学期间的汇编语言课简直跟天书一样深奥，它在我们心里埋下了恐惧的种子，仿佛一提到汇编语言，它就会像紧箍咒一样勒紧我们的头，让我们疼痛不已。汇编语言真的有这么难？是，因为汇编的语法晦涩难懂；但另一方面，毕竟它只是一门语言，跟英语一样，熟能生巧。

ARM assembly is a brand new language to most iOS developers. If your major in college is Computer related, you may already have some impression about assembly. Actually, assembly is too esoteric for most college students; we’re nervous and uncomfortable dealing with it. Is assembly really too hard to learn? Yes, it’s obscure and difficult to understand. On the other hand, however, as a human readable language, it is no much difference with other human languages, namely, if you use it more often, you will get familiar with it quicker.

我们一般的工作中与汇编打交道的机会并不多，如果不刻意练习，陡然面对时必然掌握不了，所以会觉得它很难。不过归根到底还是投入的时间和精力是否足够的问题——好了，iOS逆向工程给你学习ARM汇编提供了一个绝佳的条件——当我们在逆向一个功能时，往往需要分析大量ARM汇编代码，并把它们翻译成高级语言，试图重新实现这个功能；虽然暂时还不需要写汇编代码，但大量的阅读必然能加深我们对这门语言的理解。如果想在iOS逆向工程这条路上走下去，ARM汇编是必须掌握的语言，也是一定能够掌握的语言；跟英语类似，ARM汇编的基本概念相当于26个字母和音标；指令相当于单词，它们的变种相当于单词的各种形态；调用规则相当于语法，定义句子之间的联系。接下来，让我们一步步地深入。

As App developers, chances are rare for us to deal with assembly in our daily work. In this situation, if you don’t practice deliberately, you cannot handle it for sure. In a nutshell, it’s all about whether your time and energy is poured into learning it. But now, iOS reverse engineering offers you a great chance to learn ARM assembly. When we’re reversing a function, we need to analyze massive lines of ARM assembly, and translate them to high-level language manually, to reconstruct the function. Even though there is no need to write assembly yet, a vast reading will definitely improve our understanding about it. ARM assembly is a necessity in iOS reverse engineering; you have to master it if you really want to be a figure in this field. Like English, basic ARM assembly concepts correspond to 26 letters and phonetic symbols in English; its instructions correspond to words, and instructions’ variants correspond to different word tenses; its calling conventions correspond to grammars, which define the connection between words. Sounds not that bad, right? Let’s delve into it step by step.

### 6.1.1 基本概念

### 6.1.1 Basic concepts

如果要完整地介绍ARM汇编，ARM公司的用户手册已经做得足够好了。笔者对ARM汇编也只是略知一二，肯定没有用户手册那么全面，但对于iOS逆向工程初学者来说，这些知识足以应对，适度就好。随着iPhone 5s的推出，苹果引入了性能强大的64位处理器，但本书前半部分介绍的大多数工具对64位处理器的支持都不太好，因此后半部分的内容仍以32位处理器为准，但思路是通用的。

For a thorough introduction to ARM assembly, the ARM® Architecture Reference Manual does a great job. However, as rookies, most of us don’t need a thorough introduction at all, the thousands pages ARM® Architecture Reference Manual is no better than my limited knowledge about ARM assembly, which is enough and fits junior iOS reverse engineers better. With the release of iPhone 5s, Apple brings in the more powerful 64bit processor, arm64. However, the tools introduced in the previous chapters do not fully support arm64. Therefore, the following chapters will still focus on 32bit processors, i.e. armv7 and armv7s. Nonetheless, the general methods and thoughts work on both 32bit and 64bit processors.

### 寄存器、内存和栈

### Register, memory, and stack

在高级语言，如Objective-C、C和C++里，操作对象是变量；在ARM汇编里，操作对象是寄存器（register）、内存和栈（stack）。其中，寄存器可以看成是CPU自带的变量，它们的数量一般是很有限的；当我们需要更多变量时，就可以把它们存放在内存中；不过，数量上去了，质量也下来了，对内存的操作比对寄存器的操作要慢得多。

In high-level languages like Objective-C, C, and C++, our operands are variables; whereas in ARM assembly, the operands are registers, memory, and stack. Registers can be regarded as CPU built-in variables; their amounts are often very limited. If we need more variables, we can put them in memory. However, this is a trade off between performance and amounts; memory operation is slower than register operation.

栈其实也是一片内存区域，但它具有栈的特点：先进后出。ARM的栈是满递减（Full Descending）的，向下增长，也就是开口朝下，新的变量被存放到栈底的位置；越靠近栈底，内存地址越小，如图6-1所示。

In fact, stack is in memory as well. But it works like a stack, i.e. follows the “first in last out” rule. The stack of ARM is full descending, meaning that the stack grows towards lower address, the latest object is placed at the bottom, which is at the lowest address, as shown in the figure 6-1.

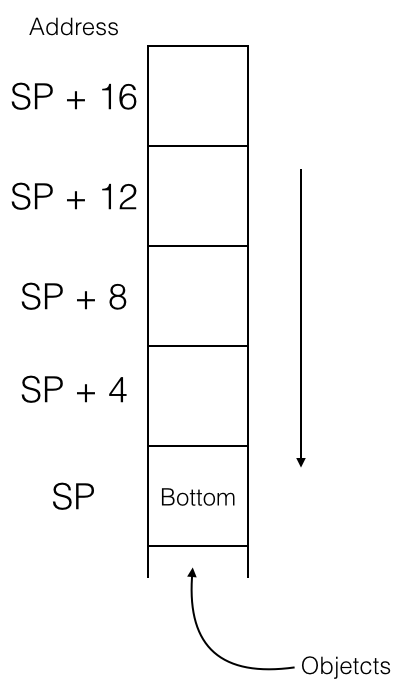


图6- 1 ARM的栈

Figure 6-1 The stack of ARM

一个名为“stack pointer”（简称SP）的寄存器保存栈的栈底地址，称为栈地址；我们可以把一个变量给入（push）栈以保存它的值，也可以让它出（pop）栈，恢复变量的原始值。在实际操作中，栈地址会不断变化；但是在执行一块代码的前后，栈地址应该是不变的，不然程序就要出问题了。为什么？举例说明：

A register, named “stack pointer” (hereafter referred to as SP), holds the bottom address of stack, i.e. the stack address. We can push a register into stack to save its value, or pop a register out of stack to load its value. During process running, SP changes a lot, but before and after a block of code is executed, SP should stay the same, otherwise there will be a fatal problem. Why? Let’s take an example:

static int global\_var0;

static int global\_var1;

…

void foo(void)

{

bar();

// other operations;

}

在上面4行代码中，假设函数foo()用到了A、B、C、D四个寄存器；foo()内部调用了bar()，假设bar()用到了A、B、C三个寄存器。因为2个不同的函数用到了3个相同的寄存器，所以bar()在开始执行前需要将3个寄存器中原来的值入栈以保存其原始值，在结束执行前将它们出栈以恢复其原始值，保证foo()能够正常执行。用伪汇编代码表示如下：

In the above code snippet, suppose that foo() uses registers A, B, C, and D; foo() calls bar(), and suppose that bar() uses registers A, B, and C. Because registers A, B and C are overlapped in foo() and bar(), bar() need to save values of A, B, and C into stack before it starts execution. Also, it needs to restore these 3 registers from stack before it ends execution, to make sure foo() can work correctly. Let’s look at some pseudo code:

// foo() function函数

foo:

// 将A、B、C、D入栈，保存它们的原始值

//push A、B、C、D, store their values

入栈 {A, B, C, D}

// 使用A ~ D

移动 A, #1 // A = 1

移动 B, #2 // B = 2

移动 C, #3 // 你猜猜这行是什么意思？

调用 bar

移动 D, global\_var0

// global\_var1 = A + B + C + D

相加 A, B // A = A + B，注意此处A的值

相加 A, C // A = A + C，还要注意此处A的值

相加 A, D // 你再猜猜这行是什么意思？

移动 global\_var1, A

// 将A、B、C、D出栈，恢复它们的原始值

出栈 {A-D}

返回

// bar()函数

bar:

// 将A、B、C入栈，保存它们的原始值A == 1，B == 2，C == 3

入栈 {A-C}

// 使用A ~ C

移动 A, #2 // 还需要注释吗？

移动 B, #5

移动 C, A

相加 C, B // C = 7

// global\_var0 = A + B + C (== 2 \* C)

相加 C, C

移动 global\_var0, C // A = 2，B = 5，C = 14

// 现在你知道入栈和出栈的重要意义了吗？

出栈 {A-C}

返回

// foo()

foo:

// push A, B, C, D into stack, save their values

push {A, B, C, D}

// use A ~ D

move A, #1 // A = 1

move B, #2 // B = 2

move C, #3 // C = 3

call bar

move D, global\_var0

// global\_var1 = A + B + C + D

add A, B // A = A + B，notice A’s value

add A, C // A = A + C，notice A’s value

add A, D // A = A + D，notice A’s value

move global\_var1, A

// pop A, B, C, D out of stack, restore their values

pop {A-D}

return

// bar()

bar:

// push A、B、C into the stack, store their values

push {A-C}

// use A ~ C

move A, #2 // Do you know what this instruction do?

move B, #5

move C, A

add C, B // C = 7

// global\_var0 = A + B + C (== 2 \* C)

add C, C

move global\_var0, C // A = 2，B = 5，C = 14

// Do you get the meaning of push and pop now?

pop {A-C}

return

简单解释一下这段伪代码：foo()先将A、B、C分别设置为1、2、3，然后调用bar()，bar()改变了A、B、C的值，并将全局变量global\_var0的值设置为ABC三者之和。如果把此时的A、B、C直接用于foo()，计算出的另一个全局变量global\_var1的值就是错的，因此在bar()执行前先要让A、B、C入栈，保存它们的值，执行完成后再出栈，使得foo()能够得到正确的global\_var1。注意一点，出于同样的目的，foo()在执行前后也对A、B、C、D执行了入栈和出栈操作，所以foo()的调用者也能够正常工作。

Let’s shortly explain this snippet of pseudo code: firstly, foo() sets registers A, B and C to 1, 2 and 3 respectively, then calls bar(), which changes values of A, B and C as well sets global\_var0, a global variable, to the sum of registers ABC. If we directly use the current values of A, B and C to calculate the value of global\_var1 for now, then the result would be wrong. So before executing bar(),values of A, B and C should be pushed into stack first, and pop them out after the execution of bar() for restoration, then we can get a correct global\_var1. Notice that, for the same reason, foo() has done the same operations on A, B, C and D, which saves its callers’ days.

### 特殊用途的寄存器

### Preserved registers

ARM处理器中的部分寄存器有特殊用途，如下所示：

Some registers in ARM processors must preserve their values after a function call, as shown below:

R0-R3 传递参数与返回值

R7 帧指针，指向母函数与被调用子函数在栈中的交界

R9 在iOS 3.0以前被系统保留

R12 内部过程调用寄存器，dynamic linker会用到它

R13 SP寄存器

R14 LR寄存器，保存函数返回地址

R15 PC寄存器

R0-R3 Passes arguments and return values

R7 Frame pointer, which pointsto the previously saved stack frame and the saved link register

R9 Reserved by system before iOS 3.0

R12 IP register，used by dynamic linker

R13 Stack Pointer, i.e. SP

R14 Link Register, i.e. LR, saves function return address

R15 Program Counter, i.e. PC

因为现在还没有开始自己写汇编代码，所以对上述知识有简单了解就足够了。

We’re not writing ARM assembly yet, so treat the above table as a reference would be enough.

### 分支跳转与条件判断

### Branchs

处理器中名为“program counter”（简称PC）的寄存器用于存放下一条指令的地址。一般情况下，计算机一条接一条地顺序执行指令，处理器执行完一条指令后将PC加1，让它指向下一条指令，如图6-2所示。

A process saves the address of the next instruction in PC register. Usually, CPU will execute instructions in order. When it has done with one instruction, PC will increase 1 to point to the next instruction, as shown in figure 6-2.

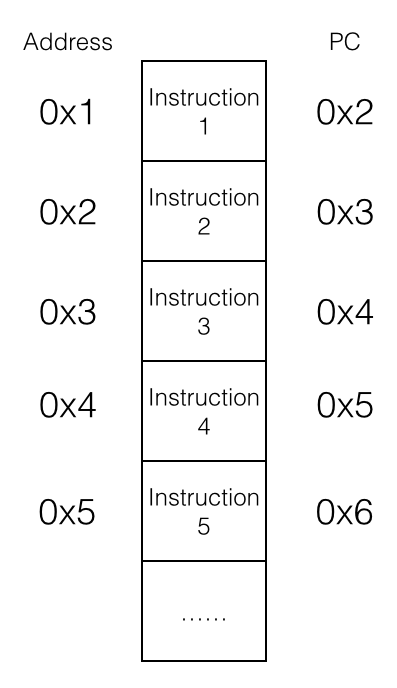


图6-2 顺序执行指令

Figure 6-2 Execute instructions in order

处理器顺序执行指令1到指令5，稀松平常、沉闷无聊。但是如果把PC的值变一变，指令的执行顺序就完全不同了，如图6-3所示。

The processor will execute instructions from 1 to 5 in a plain and trivial way. However, if we change the value of PC, the execution order will be very different, as shown in figure 6-3.

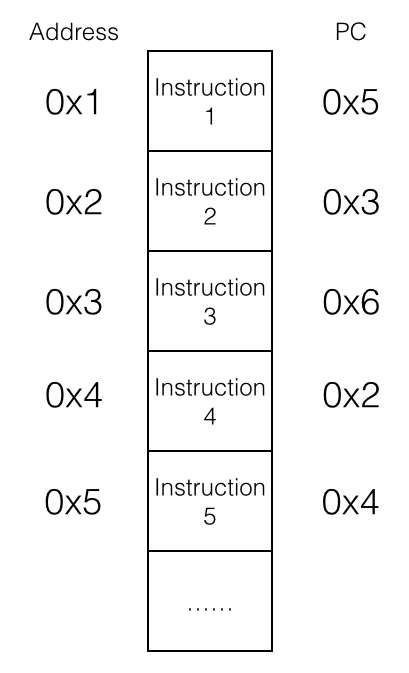


图6- 3 乱序执行指令

Figure 6-3 Execute instructions out of order

指令的执行顺序被打乱，变成指令1、指令5、指令4、指令2、指令3、指令6，光怪陆离、百花齐放。这种“乱序”的学名叫“分支”（branch），或“跳转”（jump），它使得循环和subroutine成为可能，例如：

The instructions’execution order has been disordered to 1, 5, 4, 2, 3, and 6, which is bizarre and remarkable. This kind of “disorder” is officially called “branch” or “jump”, which makes loop and subroutine possible. For example:

// endless()函数

endless:

操作 操作数1, 操作数2

分支 endless

返回 // 死循环，执行不到这里啦！

// endless()

endless:

operate op1, op2

branch endless

return // dead loop, we cannot reach here!

在实际情况中，满足一定条件才得以触发的分支是最实用的，这种分支称为条件分支。if else和while都是基于条件分支实现的。在ARM汇编中，分支的条件一般有4种：

In actual cases, conditional branches, which are triggered under some specific conditions, are the most practical branches. “If else” and “while” are both based on conditional branches. In ARM assembly, there are 4 kinds of conditional of branches:

* 操作结果为0（或不为0）；
* The result of operation is zero (or non-zero).
* 操作结果为负数；
* The result of operation is negative.
* 操作结果有进位；
* The result of operation has carry.
* 运算溢出（比如两个正数相加得到的数超过了寄存器位数）。
* The operation overflows (for example, the sum of two positive numbers exceeds 32 bits).

这些条件的判断准则（flag）存放在程序状态寄存器（Program Status Register，简称PSR）中，数据处理相关指令会改变这些flag，分支指令再根据这些flag决定是否跳转。下面的伪代码展示了一个for循环：

These operation results are often represented as flags and are saved in the Program Status Register (PSR). Some instructions will change these flags according to their operation results, and conditional branches decide whether to branch according to these flags. The pseudo code below shows an example of for loop:

for:

相加 A, #1

比较 A, #16

不为0则跳转到for

for:

add A, #1

compare A, #16

bne for // if A-16 != 0 then jump to for

此循环将A和#16作比较，如果两者不相等，则将A加1，继续比较。如果两者相等，则不再循环，继续往下执行。

The above code compares A and #16, if they’re not equal, increase A by 1 and compare again. Otherwise break out the loop and go on to the next instruction.

### 6.1.2 ARM/THUMB指令解读

### 6.1.2 Interpretation of ARM/THUMB instructions

ARM处理器用到的指令集分为ARM和THUMB两种；ARM指令长度均为32bit，THUMB指令长度均为16bit。所有指令可大致分为3类，分别是数据操作指令、内存操作指令和分支指令。

ARM processors use 2 different instruction sets: ARM and THUMB. The length of ARM instructions is universally 32 bits, whereas it’s 16 bits for THUMB instructions. Broadly, both sets have 3 kinds of instructions: data processing instructions, register processing instructions, and branch instructions.

### 数据操作指令

### Data processing instructions

数据操作指令有2条规则：

There’re 2 rules in data processing instructions:

* 1. 所有操作数均为32bit；
  2. All operands are 32 bits.
  3. 所有结果均为32bit，且只能存放在寄存器中。
  4. All results are 32 bits, and can only be stored in registers.

总的来说，数据操作指令的基本格式是：

In a nutshell, the basic syntax of data processing instructions is:

op{cond}{s} Rd, Rn, Op2

其中，“cond”和“s”是两个可选后缀；“cond”的作用是指定指令“op”在什么条件下执行，共有17种条件：

“cond” and “s” are two optional suffixes. “cond” decides the execution condition of “op”, and there are 17 conditions:

EQ 结果为0（EQual to 0）

NE 结果不为0（Not Equal to 0）

CS 有进位或借位（Carry Set）

HS 同CS（unsigned Higher or Same）

CC 没有进位或借位（Carry clear）

LO 同CC（unsigned LOwer）

MI 结果小于0（MInus）

PL 结果大于等于0（PLus）

VS 溢出（oVerflow Set）

VC 无溢出（oVerflow Clear）

HI 无符号比较大于（unsigned HIgher）

LS 无符号比较小于等于（unsigned Lower or Same）

GE 有符号比较大于等于（signed Greater than or Equal）

LT 有符号比较小于（signed Less Than）

GT 有符号比较大于（signed Greater Than）

LE 无符号比较小于等于（signed Less than or Equal）

AL 无条件（ALways，默认）

EQ The result equals to 0 (EQual to 0)

NE The result doesn’t equal to 0 (Not Equal)

CS The has carry or borrow (Carry Set)

HS Same to CS (unsigned Higher or Same)

CC The operation has no carry or borrow (Carry Clear)

LO Same to CC (unsigned LOwer)

MI The result is negative (MInus)

PL The result is greater than or equal to 0 (PLus)

VS The operation overflows (oVerflow Set)

VC The operation doesn’t overflow (oVerflow Clear)

HI If operand1 is unsigned HIgher than operand2

LS If operand1 is unsigned Lower or Same than operand2

GE If operand1 is signed Greater than or Equal to operand2

LT If operand1 is signed Less Than operand2

GT If operand1 is signed Greater Than operand2

LE If operand1 is signed Less than or Equal operand2

AL ALways，this is the default

“cond”的用法很简单，例如：

“cond” is easy to use, for example:

比较 R0, R1

移动 GE R2, R0

移动 LT R2, R1

compare R0, R1

moveGE R2, R0

moveLT R2, R1

比较R0和R1的值，如果R0大于等于R1，则R2 = R0；否则R2 = R1。

Compare R0 with R1, if R0 is greater than or equal to R1, then R2 = R0, otherwise R2 = R1.

“s”的作用是指定指令“op”是否设置flag，共有4种flag：

“s” decides whether “op” sets flags or not, there are 4 flags:

N（Negative）

如果结果小于0则置1，否则置0；

Z（Zero）

如果结果是0则置1，否则置0；

C（Carry）

对于加操作（包括CMN）来说，如果产生进位则置1，否则置0；对于减操作（包括CMP）来说，Carry相当于Not-Borrow，如果产生借位则置0，否则置1；对于有移位操作的非加/减操作来说，C置移出值的最后一位；对于其它的非加/减操作来说，C的值一般不变；

V（oVerflow）

如果操作导致溢出，则置1，否则置0。

N (Negative)

If the result is negative then assign 1 to N, otherwise assign 0 to N.

Z (Zero)

If the result is zero then assign 1 to Z, otherwise assign 0 to Z.

C (Carry)

For add operations (including CMN), if they have carry then assign 1 to C, otherwise assign 0 to C; for sub operations (including CMP), Carry acts as Not-Borrow, if borrow happens then assign 0 to C, otherwise assign 1 to C; for shift operations (excluding add or sub), assign C the last bit to be shifted out; for the rest of operations, C stays unchanged.

V (oVerflow)

If the operation overflows then assign 1 to V, otherwise assign 0 to V.

需要注意一点，C flag表示无符号数运算结果是否溢出；V flag表示有符号数运算结果是否溢出。

One thing to note, C flag works on unsigned calculations, whereas V flag works on signed calculations.

数据操作指令可以大致分为以下4类：

Data processing instructions can be divided into 4 kinds:

* 算术操作
* Arithmetic instructions

ADD R0, R1, R2 ; R0 = R1 + R2

ADC R0, R1, R2 ; R0 = R1 + R2 + C(arry)

SUB R0, R1, R2 ; R0 = R1 - R2

SBC R0, R1, R2 ; R0 = R1 - R2 - !C

RSB R0, R1, R2 ; R0 = R2 - R1

RSC R0, R1, R2 ; R0 = R2 - R1 - !C

算术操作中，ADD和SUB为基础操作，其他均为两者的变种。RSB是“Reverse SuB”的缩写，仅仅是把SUB的两个操作数调换了位置而已；以“C”（即Carry）结尾的变种代表有进位和借位的加减法，当产生进位或没有借位时，将Carry flag置1。

All arithmetic instructions are based on ADD and SUB. RSB is the abbreviation of “Reverse SuB”, which just reverse the two operands of SUB; instructions end with “C” stands for ADD with carry or SUB with borrow, and they will assign 1 to C flag when there is carry or no borrow.

* 逻辑操作
* Logical operation instructions

AND R0, R1, R2 ; R0 = R1 & R2

ORR R0, R1, R2 ; R0 = R1 | R2

EOR R0, R1, R2 ; R0 = R1 ^ R2

BIC R0, R1, R2 ; R0 = R1 &~ R2

MOV R0, R2 ; R0 = R2

MVN R0, R2 ; R0 = ~R2

逻辑操作指令没什么多说的，它们的作用都已经用C操作符表示出来了，大家应该很熟悉；但是C操作符里的移位操作并没有对应的逻辑操作指令，因为ARM采用了桶式移位，共有4种指令：

There is not much to explain about these instructions with their corresponding C operators. You may have noticed that there’s no shift instruction, because ARM uses barrel shift with 4 instructions:

LSL 逻辑左移，见图6-4

LSL Logic Shift Left, as shown in figure 6-4

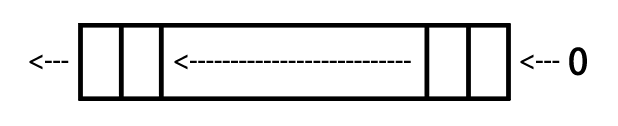


图6- 4 逻辑左移

Figure 6-4 LSL

LSR 逻辑右移，见图6-5

LSR Logic Shift Right, as shown in figure 6-5

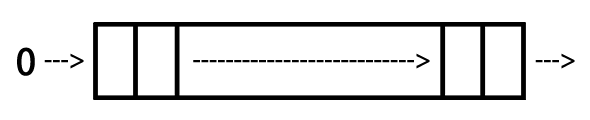


图6- 5 逻辑右移

Figure 6-5 LSR

ASR 算术右移，见图6-6

ASR Arithmetic Shift Right, as shown in figure 6-6

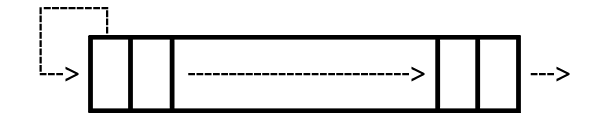


图6- 6 算术右移

Figure 6-6 ASR

ROR 循环右移，见图6-7

ROR ROtate Right, as shown in figure 6-7

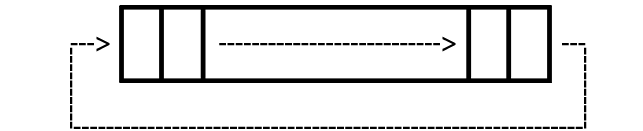


图6- 7 循环右移

Figure 6-7 ROR

* 比较操作
* Compare instructions

CMP R1, R2 ; 执行R1 - R2并依结果设置flag

CMN R1, R2 ; 执行R1 + R2并依结果设置flag

TST R1, R2 ; 执行R1 & R2并依结果设置flag

TEQ R1, R2 ; 执行R1 ^ R2并依结果设置flag

CMP R1, R2 ; Set flag according to the result of R1 - R2

CMN R1, R2 ; Set flag according to the result of R1 + R2

TST R1, R2 ; Set flag according to the result of R1 & R2

TEQ R1, R2 ; Set flag according to the result of R1 ^ R2

比较操作其实就是改变flag的算术操作或逻辑操作，只是操作结果不保留在寄存器里而已。

Compare instructions are just arithmetic or logical operation instructions that change flags, but they don’t save the results to registers.

* 乘法操作
* Multiply instructions

MUL R4, R3, R2 ; R4 = R3 \* R2

MLA R4, R3, R2, R1 ; R4 = R3 \* R2 + R1

乘法操作的操作数必须来自寄存器。

The operands of multiply instructions must come from registers.

### 内存操作指令

### Register processing instructions

内存操作指令的基本格式是：

The basic syntax of register processing instructions is:

op{cond}{type} Rd, [Rn, Op2]

其中Rn是基址寄存器，用于存放基地址；“cond”的作用与数据操作指令相同；“type”指定指令“op”操作的数据类型，共有4种：

Rn, the base register, stores base address; the function of “cond” is the same to data processing instructions; “type” decides the data type which “op” operates, there are 4 types:

B（unsigned Byte）

无符号byte（执行时扩展到32bit，以0填充）；

SB（Signed Byte）

有符号byte（仅用于LDR指令；执行时扩展到32bit，以符号位填充）；

H（unsigned Halfword）

无符号halfword（执行时扩展到32bit，以0填充）；

SH（Signed Halfword）

有符号halfword（仅用于LDR指令；执行时扩展到32bit，以符号位填充）。

B (unsigned Byte)

Unsigned byte (Extends to 32 bits when executing，filled with 0).

SB (Signed Byte)

Signed byte (For LDR only；extends to 32 bits when executing，filled with the sign bit).

H (unsigned Halfword)

Unsigned halfword (Extends to 32 bits when executing，filled with 0).

SH (Signed Halfword)

Signed halfword (For LDR only；extends to 32 bits when executing，filled with the sign bit).

如果不指定“type”，则默认数据类型是word。

The default data type is word if no “type” is specified.

ARM内存操作基础指令只有两个：LDR（LoaD Register）将数据从内存中读出来，存到寄存器中；STR（STore Register）将数据从寄存器中读出来，存到内存中。两个指令的使用情况如下：

There are only two basic register processing instructions: LDR (LoaD Register), which reads data from memory then write to register; and STR (STore Register), which reads data from register then write to memory. They’re used like this:

* LDR

LDR Rt, [Rn {, #offset}] ; Rt = \*(Rn {+ offset})，{}代表可选

LDR Rt, [Rn, #offset]! ; Rt = \*(Rn + offset); Rn = Rn + offset

LDR Rt, [Rn], #offset ; Rt = \*Rn; Rn = Rn + offset

LDR Rt, [Rn {, #offset}] ; Rt = \*(Rn {+ offset}), {} is optional

LDR Rt, [Rn, #offset]! ; Rt = \*(Rn + offset); Rn = Rn + offset

LDR Rt, [Rn], #offset ; Rt = \*Rn; Rn = Rn + offset

* STR

STR Rt, [Rn {, #offset}] ; \*(Rn {+ offset}) = Rt

STR Rt, [Rn, #offset]! ; \*(Rn {+ offset}) = Rt; Rn = Rn + offset

STR Rt, [Rn], #offset ; \*Rn = Rt; Rn = Rn + offset

此外，LDR和STR的变种LDRD和STRD还可以操作双字（Doubleword），即一次性操作2个寄存器，其基本格式是：

Besides, LDRD and STRD, the variants of LDR and STR, can operate doubleword, namely, LDR or STR two registers at once. The syntax of them is:

op{cond} Rt, Rt2, [Rn {, #offset}]

其用法与原型类似，如下：

The use of LDRD and STRD is just like LDR and STR:

* STRD

STRD R4, R5, [R9,#offset] ; \*(R9 + offset) = R4; \*(R9 + offset + 4) = R5

* LDRD

LDRD R4, R5, [R9,#offset] ; R4 = \*(R9 + offset); R5 = \*(R9 + offset + 4)

除了LDR和STR外，还可以通过LDM（LoaD Multiple）和STM（STore Multiple）进行块传输，一次性操作多个寄存器。块传输指令的基本格式是：

Beside LDR and STR, LDM (LoaD Multiple) and STM (STore Multiple) can process several registers at the same time like this:

op{cond}{mode} Rd{!}, reglist

其中Rd是基址寄存器，可选的“!”指定Rd变化后的值是否写回Rd；reglist是一系列寄存器，用大括号括起来，它们之间可以用“,”分隔，也可以用“-”表示一个范围，比如，{R4 – R6, R8}表示寄存器R4、R5、R6、R8；这些寄存器的顺序是按照自身的编号由小到大排列的，与大括号内的排列顺序无关。

Rd is the base register, and the optional “!” decides whether the modified Rd is written back to the original Rd if “op” modifies Rd; reglist is a list of registers, which are curly braced and separated by “,”, or we can use “-” to represent a scope, such as {R4 – R6, R8} stands for R4, R5, R6 and R8; these registers are ordered according to their numbers, regardless of their positions inside the braces.

需要特别注意的是，LDM和STM的操作方向与LDR和STR完全相反：LDM是把从Rd开始，地址连续的内存数据存入reglist中，STM是把reglist中的值存入从Rd开始，地址连续的内存中。此处特别容易混淆，大家一定要注意！

Attention, the operation direction of LDM and STM is opposite to LDR and STR; LDM reads memory starting from Rd then write to reglist, while STM reads from reglist then write to memory starting from Rd. This is a little confusing; please don’t mess up.

“cond”的作用与数据操作指令相同。“mode”指定Rd值的4种变化规律，如下：

The function of “cond” is the same to data processing instructions. And, “mode” specifies how Rd is modified, including 4 cases:

IA（Increment After）

每次传输后增加Rd的值；

IB（Increment Before）

每次传输前增加Rd的值；

DA（Decrement After）

每次传输后减少Rd的值；

DB（Decrement Before）

每次传输前减少Rd的值。

IA (Increment After)

Increment Rd after “op”.

IB (Increment Before)

Increment Rd before “op”.

DA (Decrement After)

Decrement Rd after “op”.

DB (Decrement Before)

Decrement Rd before “op”.

这是什么意思呢？下面以LDM为代表，举一个简单的例子，相信大家一看就明白了。在图6-8中，R0指向的值是5。

What do they mean? We will use LDM as an example. As figure 6-8 shows, R0 points to 5 currently.

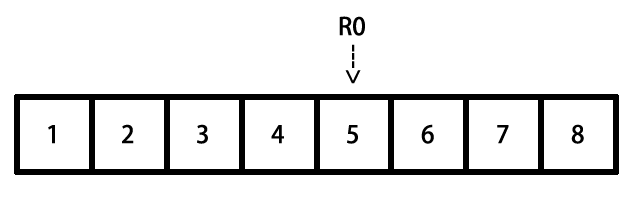


图6- 8 块传输指令模拟环境

Figure 6-8 Simulation of LDM

执行以下命令后，R4、R5、R6的值分别变成：

After executing the following instructions, R4, R5 and R6 will change to:

foo():

LDMIA R0, {R4 – R6} ; R4 = 5, R5 = 6, R6 = 7

LDMIB R0, {R4 – R6} ; R4 = 6, R5 = 7, R6 = 8

LDMDA R0, {R4 – R6} ; R4 = 5, R5 = 4, R6 = 3

LDMDB R0, {R4 – R6} ; R4 = 4, R5 = 3, R6 = 2

STM指令的作用方式与此类似，不再赘述。再次提醒，LDM和STM的操作方向与LDR和STR完全相反，切记切记！

STM works similarly. Notice again, the operation direction of LDM and STM is just opposite to LDR and STR.

### 分支指令

### Branch instructions

分支指令可以分为无条件分支和条件分支两种。

Branch instructions can be divided into 2 kinds: unconditional branches and conditional branches.

* 无条件分支
* Unconditional branches

B Label ; PC = Label

BL Label ; LR = PC – 4; PC = Label

BX Rd ; PC = Rd ,and switch instruction set

无条件分支很简单，举一个小例子就会了解：

Unconditional branches are easy to understand, for example:

foo():

B Label ; 跳转到Label处往下执行

....... ; 得不到执行

Label:

.......

foo():

B Label ; Jump to Label to keep executing

....... ; Can’t reach here

Label:

.......

* 条件分支
* Conditional branches

条件分支的cond是依照6.2.1节提到的4种flag来判断的，它们的对应关系如下：

The “cond” of conditional branches are decided by the 4 flag mentioned in section 6.2.1, their relations are:

cond flag

EQ Z = 1

NE Z = 0

CS C = 1

HS C = 1

CC C = 0

LO C = 0

MI N = 1

PL N = 0

VS V = 1

VC V = 0

HI C = 1 & Z = 0

LS C = 0 | Z = 1

GE N = V

LT N != V

GT Z = 0 & N = V

LE Z = 1 | N != V

在条件分支指令前会有一条数据操作指令来设置flag，分支指令根据flag的值来决定代码走向，例如：

Before every conditional branch there will be a data processing instruction to set the flag, which determines if the condition is met or not, hence influence the code execution flow.

Label:

LDR R0, [R1], #4

CMP R0, 0 ; 如果R0 == 0，Z = 1；否则Z = 0

BNE Label ; Z == 0则跳转

Label:

LDR R0, [R1], #4

CMP R0, 0 ; If R0 == 0 then Z = 1; else Z = 0

BNE Label ; If Z == 0 then jump

### THUMB指令

### 4. THUMB instructions

THUMB指令集是ARM指令集的一个子集，每条THUMB指令均为16bit；因此THUMB指令比ARM指令更节省空间，且在16位数据总线上的传输效率更高。有得必有失，除了“b”之外，所有THUMB指令均无法条件执行；桶式移位无法结合其他指令执行；大多数THUMB指令只能使用R0~R7这8个寄存器等。相对于ARM指令，THUMB指令的特点是：

THUMB instruction set is a subset of ARM instruction set. Every THUMB instruction is 16 bits long, so THUMB instructions are more space saving than ARM instructions, and can be faster transferred on 16-bit data bus. However, you can’t make an omelet without breaking eggs. All THUMB instructions except “b” can’t be executed conditionally; barrel shift can’t cooperate with other instructions; most THUMB instructions can only make use of registers R0 to R7, etc. Compared with ARM instructions, the features of THUMB instructions are:

* 指令数量减少
* There’re less THUMB instructions than ARM instructions

既然THUMB只是一个子集，指令数量必然会减少。例如，乘法指令中只有MUL保留了下来，其他的都被精简了。

Since THUMB is just a subset, the number of THUMB instructions is definitely less. For example, among all multiply instructions, only MUL is kept in THUMB.

* 没有条件执行
* No conditional execution

除分支指令外，其他指令无法条件执行。

Except branch instructions, other instructions cannot be executed conditionally.

* 所有指令默认附带“s”
* All THUMB instructions set flags by default

即所有THUMB指令都会设置flag。

* 桶式移位无法结合其它指令执行
* Barrel shift cannot cooperate with other instructions

移位指令只能单独执行，无法与其他指令结合执行。即，可以：

Shift instructions can only be executed alone, say:

LSL R0 #2

而不可：

But cannot:

ADD R0, R1, LSL #2

* 寄存器使用受限
* Limitation of registers

除非显式声明，否则THUMB指令只能使用R0~R7寄存器；但也有例外：ADD、MOV和CMP指令可以将R8~R15作为操作数使用；LDR和STR可以使用PC或SP寄存器；PUSH可以使用LR，POP可以使用PC；BX可以使用所有寄存器。

Unless with explicit declaration, THUMB instructions can only make use of R0 to R7. However, there are exceptions: ADD, MOV, and CMP can use R8 to R15 as operands; LDR and STR can use PC or SP; PUSH can use LR, POP can use PC; BX can use all registers.

* 立即数和第二操作数使用受限
* Limitation of immediate values and the second operand

大多数THUMB数据操作指令的形式是“op Rd, Rm”，只有移位指令、ADD、SUB、MOV和CMP是例外。

Most of THUMB instructions’ format is “op Rd, Rm”, excluding shift instructions, ADD, SUB, MOV and CMP.

* 不支持数据写回
* Doesn't support data write back

除了LDMIA和STMIA外，其他THUMB指令均不支持数据写回，即“!”不可用。

All THUMB instructions do not support data write back i.e. “!”, except LDMIA and STMIA.

我们在iOS逆向工程初级阶段经常会碰到以上指令，如果你对前两节的内容还是感到一知半解，没关系，自己动手分析两个程序就熟悉了。这一节的内容只是一个引子，在实际操作中如果对指令作用不清楚，ARM的官方文档http://infocenter.arm.com永远是最好的教科书，<http://bbs.iosre.com>上的讨论也很有参考价值。

We will read the instructions mentioned above a lot during the junior stage of iOS reverse engineering. If you only have a smattering of the knowledge so far, take it easy. Get your hands dirty and analyze several binaries from now on, you will gradually get familiar with ARM assembly. This section is just an introduction, if you have any questions about instructions in practice, ARM® Architecture Reference Manual on <http://infocenter.arm.com> will always be the best reference for you. Of course, things discussed on <http://bbs.iosre.com> are also worth to have a look.

### 6.1.3 ARM调用规则

### 6.1.3 ARM calling conventions

了解了常用的ARM指令后，相信大家已经能够基本读懂一个函数的汇编代码了。当一个函数调用另一个函数时，常常需要传递参数和返回值；如何传递这些数据，称为ARM汇编的调用规则。

After a brief look at the commonly used ARM instructions, I believe you can barely read the assembly of a function for now. When a function calls another function, arguments and return values need to be passed between the caller and the callee. The rule of how to pass them is called ARM calling conventions.

### 前言与后记

### 1. Prologs and epilogs

在6.1.1节提到，“在执行一块代码时，其前后栈地址应该是不变的”，这个操作是通过被执行代码块的前言（prologs）和后记（epilogs）完成的。前言所做的工作主要有：

We’ve mentioned in section 6.1.1 that “before and after a block of code is executed, SP should stay the same, otherwise there will be a fatal problem”. This goal is achieved by the cooperation of prolog and epilog of this code block. Generally, prolog does these:

* 将LR入栈；
* 将R7入栈；
* R7 = SP；
* 将需要保留的寄存器原始值入栈；
* 为本地变量开辟空间。
* PUSH LR;
* PUSH R7;
* R7 = SP;
* PUSH registers that must be preserved;
* Allocates space in the stack frame for local storage.

后记所做的主要工作跟前言正好相反：

And epilog does an opposite job to prolog:

* 释放本地变量占用的空间；
* 将需要保留的寄存器原始值出栈；
* 将R7出栈；
* 将LR出栈，PC = LR。
* Deallocates space that the prolog allocates;
* POP preserved registers;
* POP R7;
* POP LR, and PC = LR.

前言和后记中的这些工作并不是必须的，如果这块代码压根儿就没有用到栈，就不需要“保留寄存器原始值”这一步了。在逆向工程中，前言与后记的影响主要体现在SP的变化上，此处稍作了解即可，第10章的例子中会有详细的解答。

However, the work of prolog and epilog is not indispensable. If the code block doesn’t make use of a register at all, then there is no need to push it onto stack. In iOS reverse engineering, prologs and epilogs may change the value of SP, which deserves our attention. We’ll come across this situation in chapter 10, review this section when you get there.

### 传递参数与返回值

### 2. Pass arguments and return values

如果想详细了解参数传递规则，可以通读http://infocenter.arm.com/help/topic/com.arm.doc.  
ihi0042e/IHI0042E\_aapcs.pdf。一般情况下，记住最重要的一个金句就好：

If you want to delve deeper into how arguments and return values are passed, you can read [http://infocenter.arm.com/help/topic/com.arm.doc.ihi0042e/IHI0042E\_aapcs.pdf](http://infocenter.arm.com/help/topic/com.arm.doc.ihi0042e/ihi0042e_aapcs.pdf). However, in the majorty of cases, you just need to remember “sentence of the book”:

“函数的前4个参数存放在R0到R3中，其他参数存放在栈中；返回值放在R0中。”

“The first 4 arguments are saved in R0, R1, R2 and R3; the rest are saved on the stack; the return value is saved in R0.”

这句话的意思很好理解，为了加深印象，我们看一个例子：

A concise but informative sentence, right? To make a deeper impression, let’s see an example:

// clang -arch armv7 -isysroot `xcrun --sdk iphoneos --show-sdk-path` -o MainBinary main.m

#include <stdio.h>

int main(int argc, char \*\*argv)

{

printf("%d, %d, %d, %d, %d", 1, 2, 3, 4, 5);

return 6;

}

把这段代码存成名为main.m的文件，用注释里的那句话编译它，然后把MainBinary拖进IDA，生成的main汇编代码如图6-9所示。

Save this code snippet as main.m, and compile it with the sentence in comments. Then drag and drop MainBinary into IDA and locate to main, as shown in figure 6-9.

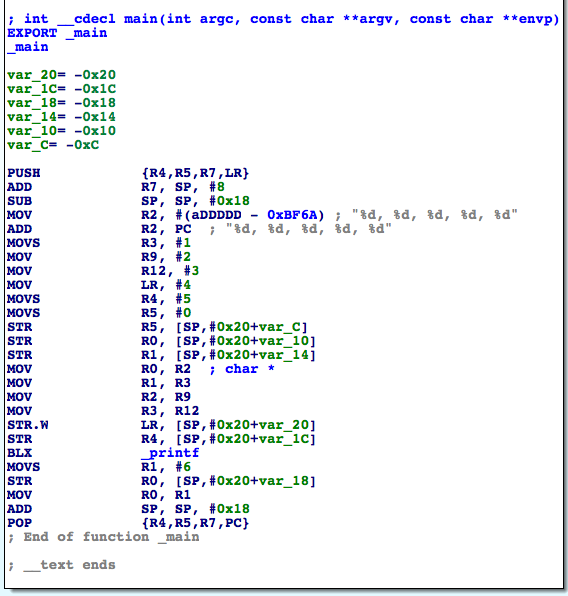


图6- 9 main的汇编代码

Figure 6-9 main in assembly

“BLX \_printf”执行printf函数，它的6个参数分别存放在R0、R1、R2、R3、[SP, #0x20 + var\_20]和[SP, #0x20 + var\_1C]中，返回值存放在R0里，其中var\_20 = -0x20，var\_1C = -0x1C，因此栈上的2个参数分别位于[SP]和[SP, #0x4]。

“BLX \_printf” calls printf, and its 6 arguments are stored in R0, R1, R2, R3, [SP, #0x20 + var\_20], and [SP, #0x20 + var\_1C] respectively; the return value is stored in R0. Because var\_20 = -0x20，var\_1C = -0x1C, 2 arguments in the stack are at [SP] and [SP, #0x4].

还需要更多解释吗？

I don’t think we need further explanation.

“函数的前4个参数存放在R0到R3中，其他参数存放在栈中；返回值放在R0中。”

“The first 4 arguments are saved in R0, R1, R2 and R3; the rest are saved on the stack; the return value is saved in R0.”

一定要牢记上面这句话！

Promise me you’ll remember “sentence of the book”, which is the key to most problems in iOS reverse engineering!

本节只是把iOS逆向工程用到的最基本的ARM汇编知识过了一遍，难免有遗漏，但说白了，只要记住刚才的“金句”，配合ARM官方网站，就已经可以开始分析程序了。接下来，就来实际动手，看看如何把刚刚学到的知识运用到iOS逆向工程中。

This section just walked you through the most basic knowledge about ARM assembly, there were omissions for sure. However, to be honest, with “sentence of the book” and the official site of ARM, you can start reversing 99% of all Apps. Next, it’s time for us to figure out how to use the knowledge we have just learned in practical iOS reverse engineering.