14｜汇编代码学习（一）：熟悉CPU架构和指令集

2021-09-08 宫文学

《手把手带你写一门编程语言》 课程介绍 >

讲述：宫文学
时长 21:40 大小 19.85M

你好，我是宫文学。

经过了上一节课的学习，你已经对物理机的运行期机制有了一定的了解。其中最重要的知识点就是，为了让一个程序运行起来，硬件架构、操作系统和计算机语言分别起到了什么作用？对这些知识的深入理解，是让你进入高手行列的关键。

接下来，就让我们把程序编译成汇编代码，从而生成在物理机上运行的可执行程序吧！

慢着，不要太着急。为了让你打下更好的基础，我决定再拿出一节课来，带你了解一下 CPU 架构和指令集，特别是 ARM 和 X86 这两种使用最广泛的 CPU 架构，为你学习汇编语言打下良好的基础。

首先，我们讨论一下什么是 CPU 架构，以及它对学习汇编语言的作用。

掌握汇编语言的关键，是了解 CPU 架构

提到汇编语言，很多同学都会觉得很高深、很难学。其实这是个误解，汇编语言并不难掌握。

为什么这么说呢？其实前面在实现虚拟机的时候，我们已经接触了栈机的字节码。你觉得它难吗？JVM 的字节码理论上不会超过 128 条，而我们通过前面几节课已经了解了其中的好几十条指令，并且已经让他们顺利地运转起来了。

而且，汇编代码作为物理机的指令，也不可能有多么复杂。因为 CPU 的设计，就是要去快速地执行一条条简单的指令，所以这些指令不可能像高级语言那样充满复杂的语义。

我们可以把学习汇编语言跟学习高级语言做一下类比。如果你接触过多门计算机语言，很快就能找到这些计算机语言的相似性，比如都有基础数据类型、都支持加减乘除等各种运算、都支持各种表达式和语句等等。抓住这些基本规律以后，学习一门新语言的速度会很快。

汇编语言也是一样的。不同的 CPU 有不同的指令集，但它们的指令的格式有一些共性，比如都包含操作码的助记符和操作数，而操作数通常也都包含立即数、寄存器和内存地址这三个类别。它们也都包含数据处理、内存读写、跳转、子过程调用等几种大类的指令。所以，你也可以很快掌握这些规律或模式，做到触类旁通，游刃有余。

说了那么多，那么学习汇编语言的关键是什么呢？

由于汇编语言是跟具体的 CPU 打交道的，所以不同的 CPU 架构，它们的汇编语言就会有所差别。如果你能够深刻了解某款 CPU 的架构，自然也就会为它编写汇编代码了。

这里提到了一个词，**架构**。所谓架构，是指一个处理器的功能规范，定义了 CPU 在什么情况下会产生什么行为。你也可以把它理解成软件和硬件之间的一个桥梁，规定了硬件如何提供功能被软件所调用。所以，如果你要搞编译技术，就必须要了解目标 CPU 的架构。

我把 CPU 架构里的内容整合在这张表里，你可以保存起来：

一个CPU的架构一般包含哪些内容?	
指令集	每条指令的功能是什么，以及它在内存中的存储格式，也就是编码
寄存器集	一共有多少寄存器，每个寄存器的位数、功能和初始状态
异常模式	我们之前提到的中断机制就是异常的一种，它会停下CPU的正常执行流程，去执行一些异常处理的代码。在这里，异常模式要规定出异常的类型、有几个优先级以及触发异常和从异常返回时，分别会发生哪些事情
内存模式	内存访问的顺序、高速缓存是如何工作的，以及软件什么时候必须介入来维护高速缓存的正确状态。我们在上一节课描述过，在并发等情况下，编译器需要产生相应的代码，保证多个内核中对同一个数据的版本保持一致
其他	为Debug、Trace和Profile所提供的支持

极客时间

看上去，要深入了解一个 CPU 架构，涉及的知识还蛮多的。不过，**作为初学者，我们最重要的是关注两个方面就行了：指令集和寄存器集，因为它们跟汇编代码的关系最密切。**

那么如何了解一个 CPU 的架构呢？正如我在第 12 节课说的那样，其实最重要的方式，就是阅读 CPU 的手册。

那么这节课我们就分析两种主流的 CPU 架构，看看能获得哪些知识点。首先我们就来看一下 ARM 架构的 CPU 的特点，它是目前大多数智能手机所采用的 CPU。

了解 ARM 架构

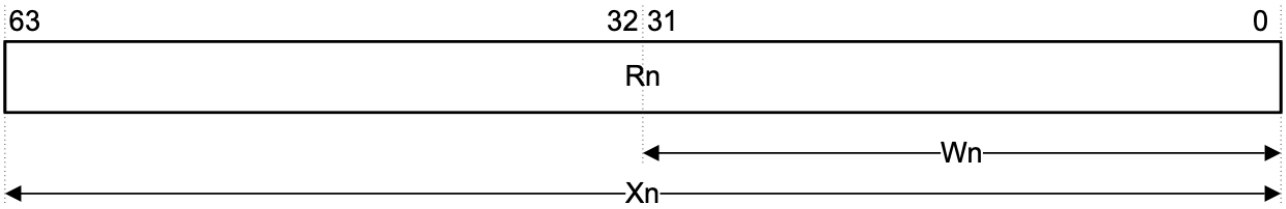
ARM 处理器是 ARM 公司推出的一系列处理器的名称。ARMv8 是它比较新的架构，当前大多数高端智能手机都是采用这个架构。了解这个架构的方法呢，当然是下载ARMv8 的手册。

不过这本手册比较厚，有 8000 多页。为了加速你的理解，我挑其中最有用、跟编写汇编代码最相关的几个知识点跟你聊聊。

首先，ARMv8 支持 32 位和 64 位运行模式，分别叫做 AArch32 和 AArch64。在 64 位模式下，它的指令集叫做 A64。

接着，我们看看 ARMv8 的寄存器。在 AArch64 架构下，它的寄存器有下面这几个（参见手册的 B1.2.1 部分）。

R0-R30：是 31 个通用寄存器。当它们被用于 64 位计算或 32 位计算的时候，分别被叫做 X0-X30（X 表示 64 位），以及 W0-W30（W 是 Word 的意思，表示 32 位）。这 31 个处理器是我们用做数据处理的主力。



SP 寄存器：64 位的栈指针寄存器，用于指向栈顶的地址。

PC 寄存器：64 位的程序代码寄存器，PC 是 Program Code 的意思。这个寄存器记录了内存中当前指令的地址，CPU 会从这个地址读取指令并执行。当程序执行跳转指令、进入异常或退出异常的时候，这个寄存器的值会被自动修改。

另外，还有一组寄存器是用于处理浮点数运算和矢量计算的，你可以去官方手册看看。

这些就是我们的汇编代码中会涉及到的寄存器，还有一些寄存器是系统级的寄存器，我们平常的应用代码用不上，就先不管了。

谈到寄存器，我插个话题。我注意到，你如果在网上搜索某个 CPU 架构的文章，往往得到的是模棱两可的、甚至是错误的信息，你千万要注意不要被它们误导了。

比如，我看到有的文章说在某个架构的 CPU 中，哪些寄存器是用来放返回值的，哪些寄存器是用来传参数的，而哪些寄存器又分别是由调用者或被调用者保护的，等等，还配了图做说明。

但如果你对调用约定或 ABI 的概念有所了解的话马上就会知道，这些其实都是软件层面上的一些约定，不是 CPU 架构层面上的规定。如果你还想了解得更具体一些，可以参考涉及到 ARM 架构的一些ABI 规范文档，特别是其中的“Procedure Call Standard for the Arm® 64-bit”，这篇文档就规定了如何使用这些通用寄存器等信息。但作为语言的作者，你其实可以设计自己的 ABI，你拥有更大的自由度。

与寄存器相关的一个概念，叫做 Process State，或者 PSTATE，它是 CPU 在执行指令时形成的一些状态信息，这些状态信息在物理层面是保存在一些特殊目的寄存器里。

PSTATE 有什么用呢？比如，它的用途之一是辅助跳转指令的运行。当我们执行一个条件跳转指令之前，会先执行一个比较指令，这个比较指令就会设置某个状态信息，而后续的跳转指令就可以基于这个状态信息进行正确的跳转了。

此外，PSTATE 还可以用于判断算术运算是否溢出、是否需要进位等等。

最后，我们终于谈到 CPU 架构中的主角，**指令集**了。你先看一下 A64 指令集中的一些常见的指令：

指令	说明
add	加法运算
sub	减法运算，subs会修改flag寄存器中的标志位
mul	乘法运算
udiv、sdiv	除法运算（无符号、有符号）
mov	把立即数或寄存器的值赋值到目的寄存器
ldr	就是Load，把数据从内存加载到寄存器
str	就是Store，把数据从寄存器保存回内存
b、bl	无条件跳转指令，跳到另一个地址执行 这个时候PC寄存器的值会发生变化
cmp	比较两个操作数，并根据比较结果设置CPU的PSTATE
beq、bne、bgt、bgt、bge、ble、blt	有条件的跳转指令，该指令会查询CPU的PSTATE，决定是否跳转

你看这些指令，是不是跟前面学过的 Java 字节码的指令集有很多相似之处？我们来比较一下看，上面的指令大概可以分为四组。

第一组指令，是加减乘除等算术运算的指令。

回忆一下，在我们之前学栈机的时候，是不是也有这些运算指令？但栈机和寄存器机的指令有一些差别。栈机的运算指令，是不需要带操作数的，因为操作数已经在操作数栈里。这几个指令会从栈顶取出两个操作数，做完加减乘除运算后，再放回栈顶。这样，下一条指令就可以把这个值作为操作数，继续进行计算。

但寄存器机上没有操作数栈，典型的寄存器机，所有运算都发生在寄存器里。我们来看看下面这个示例程序：

复制代码

```
int foo(int a){
    return a + 10;
}
```

你可以用下面这个命令生成 ARM64 指令集的汇编代码：

复制代码

```
clang -arch arm64 -S  foo.c -o foo_arm64.s -O2
```

汇编代码的主要部分是下面这几行：

复制代码

```
_foo:                                   ; @foo
    add w0, w0, #10                     ; =10
    ret
```

其中“add w0, w0, #10”的意思，是把 w0 寄存器的值加上 10，结果仍然放到 w0。

在 CPU 指令里，我们把常量一般叫做立即数（immediate）。在这条代码里，#10 就是个立即数。根据当前的调用约定，当我们调用 foo 函数的时候，参数 a 是传入到 w0 寄存器的。然后，这个寄存器的值又加上了 10，返回值也放在 w0 里就可以了，函数调用者会从 w0 里获取这个返回值。

第二组指令，只有一个 mov 指令，它能把数据从一个寄存器拷贝到另一个寄存器，也可以把一个立即数设置到目标寄存器。

第三组指令，是内存读写的指令。

在 ARM 指令集中，算术运算性的指令只能基于寄存器，但是原始数据通常来自于内存，计算结果最后也通常保存回内存，所以我们还需要一组指令把数据从内存加载到寄存器，以及从寄存器再写回内存，这就是 load 和 store 指令。

这两个指令我们也很熟悉，因为它们在栈机的指令里也出现过。在栈机里，我们是用 load 命令把本地变量和常量加载到操作数栈，用 store 命令操作数栈中的数据写回到本地变量。而在寄存器机里，操作数栈变成了寄存器。

比如，“str　　w0, [sp, #12]”的意思，是把 w0 的值保存到一个内存地址，这个内存地址是 sp 寄存器的值加 12。sp 是寄存器指向栈顶的指针，操作系统会根据这个寄存器的值自动为栈分配内存。

第四组指令，是各种跳转指令。

跳转指令的基本原理，是修改程序计数器的值，让 CPU 去执行另一个地方的代码。

高级语言中的 if 语句、for 循环语句等，翻译成汇编代码后，就会生成跳转指令。在跳转之前，一般要做一个数值的比较，依据比较结果再做跳转，做比较的指令是 CMP。比较完之后，就可以根据比较结果做跳转了，比如 jeq 表示两边相等就可以跳转了。

而函数调用、方法调用，我们都把它们统称为子过程调用，本质上也是做指令的跳转。只不过做子过程调用的时候，要建立新的栈帧并保证原来的栈帧不被破坏，一些寄存器的值也要保护起来，并且还要记下返回地址，所以我们通常要花费多条指令才能完成子过程调用的工作。

上面四组指令是各类不同 CPU 的指令集都具备的。当然，我当前给你选的都是一些常用的，也是一看就容易明白的指令。明白了这些之后，你可以进一步研究更多的指令了。

在初步研究了 ARM 架构以后，我们再接再厉，了解一下 X86 架构。因为我们的台式机和服务器，目前大多采用的还都是 X86 架构的 CPU。对于 64 位的 CPU，这个系列的架构也叫做 X64 架构，或者是 AMD64 架构。并且，你还可以把它跟 ARM 架构做对比，了解它们的相同点和不同点，对比着学习，你可以有更多的收获。

了解 X86 架构

X86 架构的 CPU 有着很长的历史，贯穿了整个 PC 的发展史。要了解 X86 架构，当然也是看手册就行了，手册名称叫做《Intel® 64 and IA-32 Architectures Software Developer’ s Manual》。对于初学者，只需要看第一卷就行了，这一卷不是太厚，但已经足够你获得丰富的信息了。

首先，我们仍然看看寄存器方面的信息，X86 的架构下的寄存器可以分为下面几种。

通用寄存器：随着 X86 架构在不断地演化，它的可用的寄存器的数量也在不断增加。在 64 位模式下，一共有 16 个通用寄存器可以使用，这些寄存器可以用 8 位、16 位、32 位和 64 位的模式访问，分别拥有不同的名称，但实际上对应的是同一个物理寄存器。

字节数	位数	寄存器名称
1字节	8位	AL、BL、 CL、DL、DIL、SIL、BPL、SPL、R8B – R15B
2字节	16位	AX、BX、CX、DX、DI、SI、BP、SP、R8W – R15W
4字节	32位	EAX、EBX、ECX、EDX、EDI、ESI、EBP、ESP、R8D – R15D
8字节	64位	RAX、RBX、RCX、RDX、RDI、RSI、RBP、RSP、R8 – R15

段寄存器：在 32 位模式下，CPU 使用多个段寄存器，分别指向内存中不同段的起始地址，比如代码段、数据段和栈，来帮助计算代码、数据和栈操作的最后的地址，但在 64 位模式下，段寄存器就没有用了。

指令指针（EIP/RIP）：在 32 位模式下，EIP 寄存器保存着下一条要执行的指令在代码段中的偏移量。**换句话，代码段的段寄存器的地址加上 EIP 的地址，就是指令的逻辑地址。**当然，在 64 位模式下，段寄存器没有用，RIP 里的地址就是下一条要执行的指令的地址，它就相当于 ARM 中的 PC 寄存器。

EFLAGS 寄存器：这个寄存器有点像 ARM 中的 PSTATE，也是保存了指令执行过程中产生的状态信息，用于执行跳转指令等场景。

这里我补充一下，我前面费这么大的劲介绍段寄存器和指令指针，就是为了让你看出来，虽然不同的 CPU 架构都能够知道下一条指令的地址，但它的实现机制却是不同的。

接着，我们再看看 X86 的指令集。

X86 架构的 CPU 采用的是复杂指令集（CISC），而前面介绍的 ARM 和 RISC-V 指令集，都是精简指令集（RISC）。这两者的差别，体现在多个方面，但这些差别不是我们这节课的重点，我建议你自己去查阅一下资料，这里我们主要研究一下指令上的差别。

我在下面这张表里列出了 X86 指令集中的一些常见的指令，这些指令基本上从字面上也一看就懂。

指令	操作数	说明
mov	源：立即数、内存地址、寄存器；目标：寄存器（或者是，源：立即数、寄存器；目标：内存地址）	把数据从源拷贝到目的
add	源：立即数、内存地址、寄存器；目标：寄存器	把源的值加到目的上
sub		从目的减去源的值
imul		把源的值乘到目的上（有符号乘法）
mul		无符号乘法
idiv		从目的除以源的值（有符号除法）
div		无符号除法
and		逻辑与运算
or		逻辑或运算
not	1个操作数：内存地址、寄存器	逻辑Not运算
inc		值增加1
dec		值减少1
neg		值取负
push	1个操作数：寄存器	把寄存器的值压到栈里，并减少栈顶寄存器的值，让栈扩大
pop		把栈里的值取出到寄存器里，并增加栈顶寄存器的值，让栈缩小
call	1个操作数：被调用的过程的名称	把返回地址压到栈里，然后跳转到被调用的过程的地址
ret	无操作数	从栈桢里找到返回地址，并跳转过去
jmp	1个操作数：跳转的地址或标签	无条件跳转指令
cmp	两个操作数：需要比较的内容	比较指令
jz、je、jnz、jne、 jge、jg、jle、jl等	1个操作数：跳转的地址或标签	有条件跳转指令

你要注意的是，这些指令存在 8 位、16 位、32 位和 64 位的模式，你在汇编指令中使用时要带上不同的后缀：

字节数	英文	位数	后缀	举例
1字节	byte	8位	b	movb
2字节	world	16位	w	movw
4字节	long	32位	l	movl
8字节	quadword	64位	q	movq

接下来，我们再分析一下 X86 指令的特点。

首先，大多数指令的源和目标，都支持使用内存地址，而不像精简指令集那样，一般只用 load 和 store 指令来做内存的读写。在计算机发展的早期，寄存器的数量是很少的，这样的指令使用起来更方便。但是现在的手机芯片，都支持比较多的通用寄存器（比如 16 个甚至更高），可以尽量减少内存读写操作，所以 RISC 指令的设计方式就更有优势了。

第二，mov 指令的不同。在 ARM 的指令中，mov 只支持对寄存器的操作。而 X86 的 mov 指令，由于支持内存地址作为操作数，所以实际上支持把数据从内存加载到寄存器，从寄存器保存到内存，实现了 RISC 指令集中的 load 和 store 指令的功能。

第三，对于栈的维护和子过程的调用，X86 也有一些特别的指令，方便我们编程。

比如，通过 push 指令，我们可以把一个寄存器的值压到栈里，同时修改栈顶指针的值。所以，一条 push %rbx 指令，相当于下面两条指令，但花费的时间更少。

复制代码

```
1 subq $8, %rsp          #把%rsp的值减8，也就是栈增长8个字节，从高地址向低地址增长
2 movq %rbp, (%rsp)      #把%rbp的值写到当前栈顶指示的内存位置
```

pop 指令则是 push 指令的反向操作。

类似的还有一对指令：call 和 ret。前者是先把返回地址压到栈里，然后跳转到被调用的子过程的地址。比如“callq _foo”其实相当于下面两条指令：

复制代码

```
pushq %rip  # 保存下一条指令的地址，用于函数返回继续执行
jmp _foo   # 跳转到函数_fun1
```

ret 指令则是一个反向的操作。从栈里取出返回地址，并设置到 rip 中，让程序跳回来。

关于 X86 架构及其指令集，我们就先了解到这里。其实我们还有不少细节没有讲到，不过因为下一节课我们就要为 X86 架构的 CPU 生成汇编代码了，我们会继续介绍更多的知识点。

课程小结

今天的内容就是这些了，我希望你在这一节课记住这些重要的知识点。

首先，汇编语言是针对具体的 CPU 架构的，而每种架构的 CPU，都有不同的设计，所以它们的汇编语言也都是不同的。CPU 架构相当于针对软件开发所提供的一个接口规范，包括了寄存器、指令集、异常处理、内存模式等内容。学习 CPU 架构相关的知识，最重要的就是学会下载和阅读手册。

我们这一节课初步研究了 ARM 架构和 X86 架构的 CPU，你会发现它们有一些共性。比如，在寄存器方面，都提供了一些通用寄存器，也都有用于计算代码地址的寄存器，并都能够提供一种机制，用于保存指令执行过程中产生的一些标志信息（或状态信息），用于指令跳转等目的。在指令集方面，它们也都有用于数据处理、数据拷贝（mov）、跳转等类别的指令。

当然，它们也有一些差别，比如 X86 的数据处理类的指令也可以直接把内存地址作为操作数，等等。

我建议你可以再多看几个不同的指令集，对比着来分析。学习不同的指令集，能避免你的思维被限制在一种 CPU 设计模式里，这样也可以加深你对各种不同架构的理解。在这里，我特别建议你关注 **RISC-V 指令集**。由于它开源开放的特点，未来在我国会拥有巨大的发展潜力。运行你的程序的下一个设备，采用的可能就是 RISC-V 指令集。

好了，今天的要点就是这些。下一节课，我们将开始为 X86 架构生成汇编代码，继续在底层机制上深钻。

思考题

我们今天学到了指令集的概念，其实你当前使用的 CPU 可能会同时支持多个指令集。那你知道怎么来查询你的电脑的 CPU 支持的指令集吗？你可以把查询方法和这些指令集分享出来，并且查一查这些指令集都有什么用途，欢迎分享在留言区。

感谢你和我一起学习，也欢迎你把这节课分享给更多对汇编代码感兴趣的朋友。我是宫文学，我们下节课见。

资源链接

1. Arm Architecture Reference Manual - Armv8, for A profile Architecture

上一篇 13 | 物理机上程序运行的软件环境是怎么样的？

下一篇 15 | 汇编语言学习（二）：熟悉X86汇编代码

精选留言 (3)

奋斗的蜗牛
2021-09-08

汇编一直是弱项，看来还是要多学

展开

作者回复: 看手册，多动手。

qinsi

2021-09-08

A64更接近x86_64。相比之下A32更有趣一些，比如每条指令都自带4bit标志位来实现条件执行；又比如指令中的立即数只有12bit，却能用来表示一些32bit的整数。

作者回复: 是，我注意到risc-v也有类似的特点。

罗乾林

2021-09-08

Linux: cat /proc/cpuinfo
Windows:https://docs.microsoft.com/en-us/sysinternals/downloads/coreinfo
手上的ARM开发板Features：
fp asimd evtstrm aes pmull sha1 sha2 crc32
...
展开

作者回复: Great!