跟涛哥一起学嵌入式 08：ARM跳转指令深度剖析

文档说明	作者	日期
来自微信公众号：宅学部落(armLinuxfun)	wit	2018.7.22
嵌入式视频教程淘宝店：https://wanglitao.taobao.com/		
联系微信：brotau(宅学部落)		

顺序、选择、循环是构建程序的基本结构，任何一个逻辑复杂的程序基本上都可以由这三种程序结构组合而成。而跳转指令，则在子程序调用、选择、循环程序结构中被大量使用。程序的跳转是如何实现的呢？在了解这个机制之前，我们需要先了解一下程序计数器PC。

程序计数器 PC，是 CPU 的寄存器列表中最重要的一个寄存器。它就像一杆枪，指哪打哪：你给 PC 指针赋值哪个地址，CPU 就会到 PC 指针指向的这个地址去取指令、翻译指令、执行指令。一般情况下，当你没有给 PC 指针赋新地址时，CPU 在 PC 指针指向的地址取完指令后，PC 计数器会自动加一，指向下一条指令，程序可以自动执行下去。当我们需要跳转时，可以直接给 PC 指针赋一个新地址，于是 CPU 就会跳转到新地址去执行了。

在 ARM 中，常见的跳转指令有 B、BL、MOV、LDR 等。不同的指令，它们的使用条件、使用场合是不同的，今天就给大家总结一下它们的区别及各自使用的场合。

1. B 指令

B 指令是 ARM 中最基本的跳转指令，它的使用方法如下：

```
B label
```

上面语句表示跳转到 label 的标号处去执行。B 跳转指令是 ARM 中最简单的指令，只是单纯的跳转，而且是相对跳转。它可以跳到以当前位置 PC 为基址，前后 32MB 的地址空间范围，所以 B 指令只是在临近的代码块、标号之间跳转。

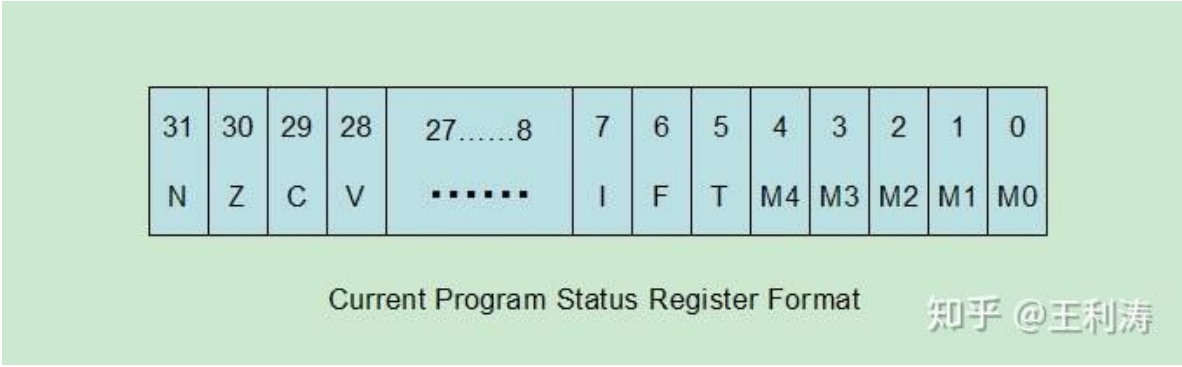
B 指令跳转，大多数时候是单向的，跳过去就不再返回来了。但是我们可以通过添加一些标号来实现一些控制逻辑：比如循环、选择程序结构：

```
;循环结构示例
LOOP
    SUB R0,R0,#1
    ...
    CMP R0,#0
    BNE LOOP
;选择结构示例
    MOV R1,#10
    MOV R2,#20
    CMP R1,R2
    BEQ HERE
    ...
    B END
```

```
HERE
    ...
END
    ...
```

在上面的程序中，我们使用B跳转指令实现了选择、循环这两种基本的程序结构。B 指令像 ARM 的其它指令一样，可以根据 CPSR 状态寄存器的标志位，有条件的执行。这样，可以减少指令数目、提高代码密度和运行效率。如 BNE、BEQ 就是当结果相等、不相等时的条件跳转。

当前程序状态寄存器：



各种各样的条件码：

条件码	CPSR标志位	含义	条件码	CPSR标志位	含义
EQ	Z=1	相等	HI	C置位，Z清零	无符号数大于
NE	Z=0	不相等	LS	C清零，Z置位	无符号小于或等于
CS/HS	C=1	无符号大于或等于	GE	N=V	有符号大于或等于
CC/LO	C=0	无符号数小于	LT	N!=V	有符号数小于
MI	N置位	负数	GT	Z清零，N=V	有符号数大于
PL	N清零	正数或零	LE	Z置位，N!=V	有符号小于或等于
VS	V置位	溢出	AL	忽略	无条件执行
VC	V清零	未溢出	NV	忽略	从不执行

2. BL 指令

BL 指令跟 B 不同：在跳转之前，会先将当前指令的下一条指令地址保存到 LR 寄存器中，然后才跳转到标号执行。这样做的好处是：当我们想从标号地方返回时，可以直接将 LR 寄存器中的返回地址赋值给 PC，程序就可以返回到原来的程序中继续执行了。

BL 跳转指令一般用在子程序的调用中。无论是汇编语言子程序，还是C语言子程序，在跳转到子程序之前，都要将返回地址保存起来。当子程序执行完毕，将LR寄存器保存的返回地址，重新赋值给 PC，处理器就可以返回到主程序继续执行了。

```
BEGIN
    MOV R0,#SRC
    MOV R1,#DST
    MOV R2,#100
    BL COPY
    NOP
    ...
 COPY
    SUB R2,R2,#1
    LDR R3,[R0],#1
    STR R3,[R1],#1
    CMP R2,#0
    BNE COPY
    MOV PC,LR
```

上面的汇编代码段，我们定义了一个汇编子程序 COPY，实现了数据拷贝的功能。当我们使用 BL 指令调用这个子程序 COPY 时，CPU 会首先将当前指令的下一条指令：NOP 的地址保存到 LR 寄存器中，然后才跳转到COPY 子程序去执行。在 COPY 子程序中，处理完数据搬运后，通过

```
MOV PC,LR
```

这条语句，将保存在 LR 寄存器中的返回地址，重新赋值给 PC，这样我们就可以返回到原来的程序中继续执行了。

在上面的汇编代码中，LR，即 R14，连接寄存器，常用来存放程序的返回地址；PC，即 R15，程序计数器，表示当前指令地址。LR 和 PC 都是 ARM 汇编器为了方便程序员编程，预定义的一些宏。你在程序中使用这些助记符其实就是相当于操作 R14 和 R15 寄存器。除此之外，ARM 中常用的助记符有：

- FP：栈帧基址寄存器，即 R12
- SP：栈指针寄存器，即 R13
- LR：链接寄存器，即 R14
- PC：程序计数器，即 R15

同样，在 C 语言调用子函数的过程中，在跳转子函数执行之前，CPU 也会将当前指令的下一条指令地址保存到LR 寄存器中，然后再跳转到子函数中执行。因为在子函数运行过程中，也有可能会用到 ARM 的一些寄存器，也有可能会调用其它的子函数，会覆盖掉保存在LR寄存器中的返回地址，所以，我们一般在运行子函数之前，会首先将LR寄存器压入子函数的栈帧，相当于将返回地址保存到了栈上。当子函数运行结束时，再通过出栈操作，将保存在栈中的返回地址弹出到 PC 指针中，这样程序就成功从子程序中返回了，直接返回到原来的函数中继续执行。

```
int main(void)
{
    func();
    printf("Hello!\n");
    return 0;
}
;对应的汇编代码
main
    BL func
    BL printf
func
    PUSH LR
    ...
    pop pc
;func子函数返回
```

3. MOV 指令

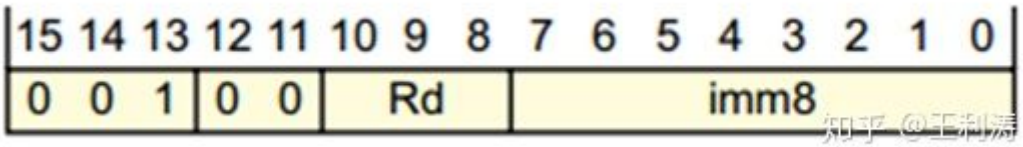
通过上面的学习，我们可以看到，无论是 B 指令、还是 BL 指令，都是相对寻址。其本质都是以当前指令地址PC为基址，然后加上一个 [0,32M] 的偏移，达到修改 PC 的目的。

除此之外，我们也可以直接给PC指针赋值，达到跳转的目的。如上面的 func 子程序返回，就是直接通过

```
MOV PC,LR
```

这条指令，将LR寄存器中的返回地址，直接赋值给PC，直返回到原来的主函数去执行。

MOV 指令主要用来在寄存器之间传输数据，或者将一个立即数传送到寄存器。但是 MOV 指令有一个硬伤，就是传递的立即数只能是 8 位数，有大小的限制。这是为什么呢？很简单，ARM 是 RISC 架构，在一个 32 位的 ARM 中，指令通常都是 32 位的。而一个指令中，通常要包括操作码+操作数，如下图



一条指令，总共有 32 个 bit 空间，MOV 这个操作码要占几位吧，Rd 寄存器编码要占据几位吧，剩下的留给立即数的空间就不多了，所以这也就限定了 MOV 指令能传递的立即数的大小了。而一般的 32 位程序中，无论是变量还是函数，它们的地址一般都是32位的，如果使用 MOV 指令，将他们的地址传送到 PC，使用下面的形式：

```
MOV PC,#0x30008000
```

你会发现，立即数#0X30008000这个地址就已经32位了，在加上MOV指令这个操作码，已经超过32位了，编译器是无法翻译这个指令的，所以说，当一个变量或函数地址为32位时，使用MOV指令给PC直接赋值，行不通，那怎么办呢？

4. LDR 伪指令

办法总是有的，比如，我们就可以通过伪指令 LDR，直接将一个 32 位的立即数地址，传送到 PC：

```
LDR PC,=0x30008000
```

LDR 伪指令的功能和 MOV 一样，都可以将一个立即数传送到寄存器。唯一区别的就是，MOV 指令只能传送8位的，而 LDR 可以传送一个 32 位的立即数或地址。

这里需要注意一下，立即数 0X30008000 的前面有一个等于号 “=”，这表示前面的 LDR 指令是一个伪指令。除此之外，在 ARM 中，LDR 还有另外一个意思，用来将内存中的数据加载到寄存器。我们知道，ARM 是 RISC 架构，使用 LDR/STR 架构，不能直接修改内存中的数据。如果我们要修改内存中的一个变量，要首先使用 LDR 指令将内存中的变量加载到寄存器中，接着对寄存器进行操作，最后再使用 STR 指令将寄存器中的变量回写到内存中。所以，LDR 可以看作是一个伪指令，也可以看做是普通的一个 LDR 指令，判定他们的区别就是看前面的等于号。

普通的 LDR 指令主要使用寄存器间接寻址，常用的使用方式如下：

```
LDR R0,[R1]
LDR R0,0x30008000
```

这里注意后面一句，是将地址 0x30008000 地址上的内容传动到寄存器 R0，而不是直接将这个地址传送到 R0，这里一定要注意其跟 LDR 伪指令的区别，这一点没有注意到，你在分析程序时就可能误入歧途了。

在《 C 语言嵌入式 Linux 高级编程》第二期中，我们已经探讨了 CPU、指令集、伪指令的基本概念，这里就不赘述了。简单来说，伪指令并不是真正的ARM指令，并不属于ARM指令集中的标准指令。它只是编译器为了方便我们程序员开发程序，定义的一些助记符。在编译时，这些伪指令还是会使用指令集中的标准指令来实现。

比如上面的 LDR 伪指令，程序在编译时，看到这个伪指令，会使用ARM指令集中标准的指令实现。如果 LDR 伪指令中的立即数小于8位，它就会转换为 MOV 指令来实现：

```
LDR R0,=200
MOV R0,#200
```

如果 LDR 伪指令中，立即数大于 8 bit 表示的数据范围，比如说是一个32位的立即数或地址，那就不能使用MOV指令来实现了，可以采用文字池的形式，先将这个地址常量单独存放在存储单元中，然后使用相对寻址，曲线救国，完成这个32位地址或立即数与寄存器之间的传输，这些细节在教程视频中都有讲到，就不再赘述了。

5. 小结

通过上面的学习，我们基本上理清了 ARM 系统中常见的几种跳转指令，以及它们的区别。只有彻底理解他们的底层机制及实现细节，才有可能在使用反汇编分析程序时，达到事半功倍的效果，从而大大提高我们的工作效率。否则，这些基本的细节和概念搞不清，将会永远成为你学习和工作上的障碍。


ARM