CHAPTER 3

第3章 ARM9指令集和汇编

ARM处理器是基于RISC(精简指令运算集)设计的,ARM9支持32位的ARM指令集和16位的Thumb指令集。ARM指令集效率高,但是代码密度低,而Thumb指令集具有更好的代码密度,却仍然保持ARM的大多数性能上的优势,它是ARM指令集的子集。通过本章的学习,读者可以获得以下知识点:

(1) RISC的基本知识。
(2) ARM状态和Thumb状态。
(3) ARM指令寻址方式。
(4) ARM指令和Thumb指令。

3.1 ARM指令集概述

3.1.1 RISC简介

ARM的全称是Advanced RISC Machine,这个名称很明确的说明了ARM是一种RISC处理器。所谓的RISC是计算机中央处理器的一种设计模式,全称Reduced Instruction Set Computer,中文称为精简指令集计算机。1974年,IBM研究中心的研究人员发现,计算机不同指令的执行密度有非常大的差异,其中有20%的指令会被频繁的使用到,承担了几乎80%的计算任务。当时计算机处理器普遍采用的是CISC(Complex Instruction set Computer)指令结构,CISC的指令类型很多,总共包含300多条指令,而且单条指令功能也很复杂,比如同一条指令中会进行取操作数、乘除法计算、回写结果等复杂操作。这样做的目的是为了支持高级语言、应用程序的复杂功能,当然导致的必然结果便是增加了处理器结构的复杂性,提高了生产成本。但实际情况是,这种复杂性增加的成本要远大于复杂指令所带来的效益,鉴于这种情况,RISC的概念被提了出来,RISC选取使用最为频繁的简单指令及部分复杂指令,而且指令等长,通常指令为16位或32位,指令格式更加规

格化和简单化，并采用高效的流水线操作，提高了数据和指令的处理速度。同时，RISC 结构采用大量的寄存器，大部分操作都在寄存器之间进行，以提高效率。存储器访问指令被独立出来，避免 CISC 结构中指令频繁的内存访问操作。

除了 ARM 以外，还有很多处理器都采用 RISC 结构，像高档服务器中应用的 HP 公司的 PA-RISC、IBM 的 Power PC 以及 SUN 公司的 Sparc 等。RISC 也并不是没有缺点，比如当用一系列指令完成某个简单任务时，由于取指令的次数变多，会导致执行时间变长。现在的 CPU 发展趋势是融合 CISC 和 RISC 各自的优点，比如超长指令集的应用。Intel 现在的 CISC 处理器已经具有了明显的 RISC 特性。

3.1.2　ARM 状态和 Thumb 状态

如 2.1.2 节所介绍的，从 v4 版本开始，ARM 引入了 Thumb 指令集。Thumb 指令为 16 位，能完成的功能是 32 位 ARM 指令的子集。对应这两类指令，ARM 处理器支持两种运行状态：ARM 状态和 Thumb 状态。ARM 指令必须在 ARM 状态下执行；同样，Thumb 指令也必须处于 Thumb 状态下执行。ARM 处理器可以在两种状态下进行切换。只要遵循 ATPCS 调用规则，Thumb 子程序和 ARM 子程序之间可以进行相互调用。

ARM 指令和 Thumb 指令并存可以增加系统的灵活性。ARM 指令在 32 位的存储下性能较高；而 Thumb 指令具有较高的指令密度，可以有效降低存储器功耗，并且在 16 位的存储器下具有较好的性能。

在一些情况下是必须使用 ARM 指令的：ARM 处理器启动的第一句指令必须是 ARM 指令，随后可以根据需要切换到 Thumb 状态下执行 Thumb 指令。访问程序状态寄存器 CPSR 或协处理器时必须是 ARM 指令。ARM 在处理异常中断时会自动切换到 ARM 状态，执行中断处理程序入口处的 ARM 指令，之后程序也可以切换到 Thumb 状态，但在中断程序返回时，会再次自动切换到 ARM 状态。

ARM 状态和 Thumb 状态切换可以通过 BX(Branch eXchange)指令来实现。BX 指令将通用寄存器 Rn(R0-R15)的值复制到程序寄存器 PC 中来实现 4G 地址范围的绝对跳转。BX 指令通过判断 Rn 所存储的目标地址的最后一位来判断要跳转到什么状态：若 Rn[0]=0，则跳转到 ARM 状态，若 Rn[0]=1，则跳转到 Thumb 状态。在 2.2.1 节中，本书介绍了状态寄存器 CPSR 的结构，可以看到 CPSR 的位 5 是 T 位，即状态控制位，它的值决定了处理器的运行状态。当 T=1 时，处理器处于 Thumb 状态，当 T=0 时，则处于 ARM 状态。可以通过直接修改 CPSR 的 T 位来达到切换运行状态的目的，但是这样有时会出现问题，因为 ARM9 采用 5 级流水线结构，在执行过程中，流水线上会存在多条预取指令。若修改了 CPSR 的 T 位，状态的切换会导致预取指令执行出错。而 BX 指令实现状态切换时，会清除流水线上的预取指令，保证在新状态下重新进行指令预取，从而避免了上述问题。

在 v4 版本中的函数调用中，如果调用过程不涉及状态的切换，情况比较简单，只需要用到 BL 指令就可以实现了，此时 R14，即连接寄存器 LR 会保存函数的返回地址，当程序结束时只要用 LR 来恢复 PC 就能实现函数返回了。但如果函数调用时需要进行状态切换，情况会复杂一点，BL 指令不能进行状态切换，需要执行 BX 指令，但 BX 指令不能自动保存函数的返回地址，需要在调用 BX 指令前保存 LR。当函数返回时，需要用指令：

```
BX LR
```

具体过程可以用图 3-1 表示。

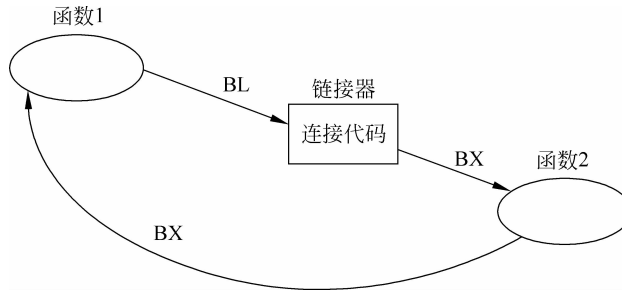

图 3-1　不同状态间的函数调用

到了 ARM v5 版本后，引入了一条新的指令 BLX，从命名规则就可以看出，它结合了 BL 和 BX 指令各自的功能特点，它使得交互的函数调用通过一条指令就可以实现。所以在 ARM9E 中可以通过执行 BLX 来进行跨状态的函数调用。v5 版本还能将 PC 加载值的最低位自动地送到 CPSR 的 T 位，这样就能通过给 PC 赋值来实现状态的切换。

3.1.3　ARM9 指令类型和指令的条件域

ARM 指令集属于加载/存储型指令，指令的操作数都储存在寄存器中，处理结果直接放回到目的寄存器中，而想要访问存储器需要使用专门的存储器访问指令。ARM9 指令集可以分为 6 类，分别是跳转指令、数据处理指令、存储器访问指令、协处理器指令、杂项指令和饱和算术指令。基本的指令如下面的表所示。

跳转指令

助记符	功能描述	助记符	功能描述
B	跳转指令	BLX	带链接和状态切换的跳转指令
BL	带链接的跳转指令	BX	带状态切换的跳转指令

数据处理指令

助记符	功能描述	助记符	功能描述
MOV	数据传输指令	ORR	逻辑或指令
MVN	数据取反传输指令	EOR	异或指令
ADD	加法指令	BIC	位清零指令
SUB	减法指令	CMP	比较指令
RSB	逆向减法指令	CMN	比较反指令
ADC	带进位加法指令	TST	位测试指令
SBC	带借位减法指令	TEQ	相等测试指令
RSC	带借位逆向减法指令	MUL	32 位乘法指令
AND	逻辑与指令	MLA	32 位乘加指令

存储器访问指令

助记符	功能描述	助记符	功能描述
LDR	存储器到寄存器的数据传输指令	STC	协处理器寄存器写入存储器指令
STR	寄存器到存储器的数据传输指令	STM	批量内存字写入指令
LDM	加载多个寄存器指令	SWP	交换指令

协处理器指令

助记符	功 能 描 述
CDP	协处理器数据操作指令
LDC	协处理器从存储器读取数据指令
STC	协处理器寄存器写入存储器指令
MCR	ARM 寄存器到协处理器寄存器的数据传输指令
MRC	协处理器寄存器到 ARM 寄存器的数据传输指令

杂项指令

助记符	功 能 描 述
SWI	软件中断指令
MRS	传送 CPSR 或 SPSR 的内容到通用寄存器指令
MSR	传送通用寄存器到 CPSR 或 SPSR 的指令
BKPT	断点指令

饱和算术指令

助记符	功能描述	助记符	功能描述
QADD	饱和加法	QDADD	SAT(Rm+SAT(Rn*2))
QSUB	饱和减法	QDSUB	SAT(Rm−SAT(Rn*2))

上述指令在 3.3 节中会有详细的介绍。

ARM 指令一般由操作码、目的寄存器、操作数几部分组成，并可以配合条件码，S 后缀等可选项目，以完成更复杂操作，它的格式一般为：

```
<opcode>{<cond>}{S} <Rd>,<Rn> {, <shift_op2>}
```

指令中<>内的项目是必需的，比如 opcode，Rd，Rn 等，{}内的项目是可选的，可以根据功能需求选择，各个项目的具体含义如表 3-1 所示。

表 3-1　ARM 指令格式

opcode	操作码，即指令助记符，如 BL，ADD
cond	条件码，描述指令执行的条件，在下文会有详细介绍
S	可选后缀，若在指令后加上“S”，在指令完毕后会自动更新 CPSR 中条件码标志位的值
Rd	ARM 指令中的目标操作数总是一个寄存器，通常用 Rd 表示
Rn	存放第 1 操作数的寄存器
opcode2	第 2 操作数，它的使用非常灵活，不仅可以是寄存器，还能使用立即数，而且能够使用经过位移运算的寄存器和立即数，这在下文也会介绍

ARM 指令集不同寻常的特征是几乎每条指令(除了某些 v5T 指令)都可以是条件执行的。ARM 指令的最高 4 位[31:28]称为条件码,它指定了指令要执行所需要满足的条件。而条件是否满足,需要根据当前程序状态寄存器 CPSR 中的条件码标志位[31:28]的复制情况决定。ARM 指令的条件码以两个字符表示,可以添加到指令助记符的后面和指令同时使用,比如常见的 BEQ 指令:B 是跳转指令,EQ 是指令条件域。约束指令只有在"相等"的情况下才会跳转,而是否"相等"则要参照 CPSR 中的位[30]中 Z 标志的值决定。条件码共 4 位,总共可以表示 16 种情况,在 ARM9 中,第 16 种(1111)情况属于系统保留。条件码的具体描述如表 3-2 所示。

表 3-2　ARM 指令条件码

ARM 指令条件码	助记符	描　　述	CPSR 条件码标志位的值
0000	EQ	相等,运行结果为 0	Z 置位
0001	NE	不相等,运行结果不为 0	Z 清零
0010	CS/HS	无符号数大于等于	C 置位
0011	CC/LO	无符号数小于	C 清零
0100	MI	负数	N 置位
0101	PL	非负数	N 清零
0110	VS	上溢出	V 置位
0111	VC	没有上溢出	V 清零
1000	HI	无符号数大于	C 置位且 Z 清零
1001	LS	无符号数小于等于	C 清零且 Z 置位
1010	GE	带符号数大于等于	N=V
1011	LT	带符号数小于	N! =V
1100	GT	带符号数大于	Z 清零且 N=V
1101	LE	带符号数小于等于	Z 置位且 N!=V
1110	AL	无条件执行 1111	系统保留

＜shift_op2＞形式非常灵活,共有 11 种形式。这是 ARM 的一个显著特点,就是在第 2 操作数进入算术逻辑单元之前可以先对操作数进行各种方式的左移或右移。具体形式如表 3-3 所示。

表 3-3　＜shift_op2＞的各种形式

语　　法	含　　义
#＜immediate＞	立即数寻址
＜Rm＞	寄存器寻址
＜Rm＞, LSL #＜shift_imm＞	立即数逻辑左移
＜Rm＞, LSL ＜Rs＞	寄存器逻辑左移
＜Rm＞, LSR #＜shift_imm＞	立即数逻辑右移
＜Rm＞, LSR ＜Rs＞	寄存器逻辑右移
＜Rm＞, ASR #＜shift_imm＞	立即数算术右移
＜Rm＞, ASR ＜Rs＞	寄存器算术右移
＜Rm＞, ROR #＜shift_imm＞	立即数循环右移
＜Rm＞, ROR ＜Rs＞	寄存器循环右移
＜Rm＞, RRX	寄存器扩展循环右移

从上表可以看出，ARM 指令集有 5 种形式的位移操作，分别是 LSL 逻辑左移，LSR 逻辑右移，ASR 算术右移，ROR 循环右移和 RRX 带扩展的循环右移。许多人对这些概念会产生混淆，有必要在这里讲清楚这几种操作的概念。

逻辑左移(Logical Shift Left)在移位操作时，用 0 补足低位；而逻辑右移(Logical Shift Right)移动的方向相反，并用 0 补足高位。

算术右移(Arithmetic Shift Right)在移位操作时，根据符号位来补足高位，若原数符号位是 1，即当原数为负数时，移位空出的高位都用 1 补足，反之则用 0 补足。

循环右移(ROtate Right)可以将数字看做首尾相接的“环形”，当最低位被移出后，它会绕到数组的最高位去，继续参与移位操作。

带扩展的循环右移(Rotate Right one bit with eXtended)较前面的几种移位方法复杂一些，它需要用到 CPSR 中的 C 位。当最低位被向右移出后，最高位有 C 位的值补足，然后被移出的最低位被放到 C 位中。

C 位在一些指令加了 S 后缀并有移位操作通常会被影响，如 MOV、MVN、AND、ORR、EOR 或 BIC。最后一位被移出的值会放到 C 位中。而指令 TEQ 和 TST 则不需要 S 位就能影响到 C 位。

关于表 3-3 中＃＜immediate＞有一点需要非常注意，立即数并不是任意数都是合法的。在立即数寻址中，分配给立即数的空间是 12 位，8 位用于保存一个常数，4 位用于保存循环右移基数，而循环右移每次需要移动偶数位，即右移的位数是基数＊2。假设常数为 A，循环右移位数为 N，则最后得到的立即数＝A 循环右移(N＊2 位)。举个例子：0x3FC 立即数合法，因为此时可以取到一组值：A＝0b11111111，N＝0d15。但是 0x1FE 不合法，因为无法找到一组值可以使得 A 和 N 同时满足条件。

但对于一些立即数来说，虽然本身“不合法”，但是当它取逆或取负时却能够变成合法的立即数。而 ARM 中有一些指令对，除了操作数存在互逆或互负的关系，其他都是相同的，如 ADD 和 SUB、ADC 和 SBC、AND 和 BIC、MOV 和 MVN、CMP 和 CMN。所以在一些时候，这些指令对会通过这样的方法得到合法的立即数并完成和原指令相同的功能，这个变换被称为指令替换(Instruction Substitution)。

3.2　ARM 指令的寻址方式

ARM 指令有 9 种寻址方式，所谓寻址方式是指处理器根据指令给出的地址信息来寻找物理地址的方式，下面将具体介绍这 9 种寻址方式。

3.2.1　立即寻址

立即寻址也可被称为立即数寻址，这种方式比较特别，其实并不需要真正的“寻址”，因为操作数本身已经包含在指令中了，读取指令后可以立即得到操作数，而不需要去物理内存得到相应内容。这个给出的操作数叫立即数，它一般以“＃”为前缀，“＃0x”、“＃0d”、“＃0b”开头的计数用来表示十六进制、十进制和二进制。举例：

```
ADD R1, R1, #0x1 ; R1 <- R1 + 1
```

3.2.2 寄存器寻址

寄存器寻址也是一种不需要访问存储器内容的寻址方式，指令中直接指明操作数所在的寄存器，执行时处理器直接访问寄存器获取操作数，如下面指令：

```
ADD R1, R1, R2 ; R1 <- R1 + R2
MOV R1, R0; R1 <- R0
```

3.2.3 寄存器偏移寻址

寄存器偏移寻址是 ARM 指令特有的一种寻址方式，它利用了＜shift_op2＞形式的灵活性，如 3.1.2 所介绍。第 2 操作数可以在与第 1 操作数结合之前，进行各种形式的移位操作，举几个简单的寄存器偏移寻址的例子：

```
ADD R1, R1, R2 , ROR #0x2;          ;R2 循环右移两位后与 R1 相加,结果放入 R1 中
MOV R1, R0, LSL R2;                 ;R0 逻辑左移 R2 位后放入 R1 中
```

3.2.4 寄存器间接寻址

寄存器间接寻址的指令中虽然也是指定寄存器，但并不是直接拿寄存器中的值来进行运算操作，此时寄存器中储存的是地址，处理器需要根据这个地址从存储器中获取操作数。所以寄存器间接寻址是需要进行存储器访问的，所以执行效率比寄存器寻址要慢。相应的指令举例如下：

```
STR R1, [R2]                        ;将 R1 的值存入以 R2 内容为地址的存储器中
SWP R1, R1, [R2]                    ;交换以 R2 为地址的存储器内容和 R1 内容
```

3.2.5 基址变址寻址

基址变址寻址与寄存器间接寻址相似，但此时从寄存器取出的内容需要加上指令所给定的偏移量，这样才构成操作数的有效地址。变址寻址方式通常用于访问基地址附近的地址单元，常用于查表、数组操作、功能部件寄存器访问等。通常基址变址寻址有以下四种形式：

```
op Rd, [Rn, R1]
op Rd, [Rn, FlexOffset]
op Rd, [Rn, FlexOffset]!
op Rd, [Rn], FlexOffset
```

按照顺序依次解释一下：第一种形式称为零偏移(Zero offset)，Rn＋R1 的结果便是有效的操作数地址；第二种形式被称为前索引偏移(Pre-Indexed)，指令首先 Rn＋FlexOffset 得到有效的操作数地址，然后完成指令操作；第三种形式被称为带写回的前索引偏移(Pre-Indexed with Writeback)，它在完成第二种形式的操作后，需要在最后将操作数地址存入 Rn 寄存器中，“!”后缀的作用就是完成 Rn 寄存器的自增功能，这种寻址方式适合数组，因为会自动进行数组下标的更新；第四种形式称为后索引偏移(Post-Indexed)，它首先根据 Rn 的值寻址操作数，在完成指令操作后，计算 Rn＋FlexOffset 的值并将其存入 Rn 寄存器中。

FlexOffset 可以被称为灵活的偏移量，它有两种形式：

```
#expr
{-}Rm{, shift}
```

可以看到它的形式和 3.1.3 节中介绍的灵活的第二操作数的形式很相似，但还是有一些不同的地方：首先，Expr 表示的整数范围为－4095 到＋4095，而不存在 8 位结构的合法性问题。第二，此时 Rm 不允许是 R15，在第二操作数中没有这个限制。第三，在书写指令时，FlexOffset 有{－}选项，第二操作数没有。关于 shift 移位操作可以完全参照第二操作数的介绍。

3.2.6　多寄存器寻址

多寄存器寻址方式可以在同一条指令中完成多个寄存器数据的传送，最多可以传送 16 个通用寄存器。举两个例子：

```
LDMIA R0, {R1, R2, R3, R4, R5} ; R1 <- R0, R2 <- R0 + 4, …, R5 <- R0 + 16
STMIA R0, {R2 - R5, R7}; R0 <- R2, R0 + 4 <- R3, … , R0 + 12 <- R5, R0 + 16 <- R7
```

LDM 和 STM 指令后缀 IA 的作用是每次加载/存储操作后，R0 的值按字长度增加，从而完成连续存储单元和多个寄存器之间内容的传递。寄存器的顺序一般都是由大到小排列，连续的寄存器可用“－”连接，不连续的寄存器之间用“，”分隔。

3.2.7　堆栈寻址

堆栈是一个后进先出的数据结构，堆栈寻址方式会有一个指针，始终指向存储单元的栈顶，这个指针需要用一个专门的寄存器来存放，这个寄存器一般是 R13，当然用户也可以自己指定。如果堆栈指针总是指向最后压入堆栈的数据，称为满堆栈(Full Stack)；当堆栈指针指向下一个空位置时，称为空堆栈(Empty Stack)。按照地址增长方式，堆栈又可以分成递增堆栈(Ascending Stack)和递减堆栈(Descending Stack)。递增堆栈从低地址向高地址生长，递减堆栈则相反。通过组合，共有 4 种堆栈类型：满递增堆栈(Full Ascending，指令如 LDMFA、STMFA)，空递增堆栈(Empty Ascending，指令如 LDMEA、STMEA)，满递减堆栈(Full Descending，指令如 LDMFD、STMFD)，空递减堆栈(Empty Descending，指令如 LDMED、STMED)，如图 3-2 所示。

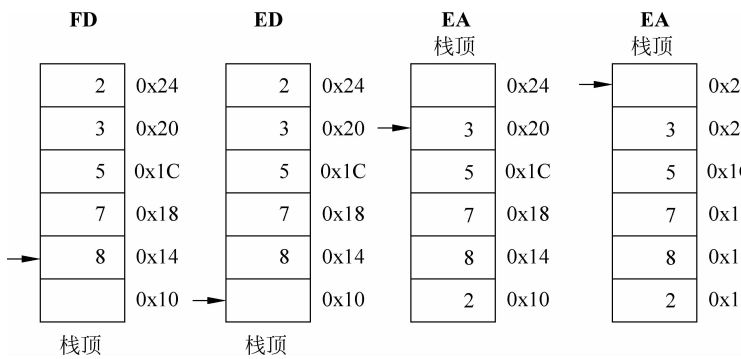

图 3-2　四种堆栈类型

堆栈寻址的例子如下：

```
STMFD SP!, {R1 - R7, LR}              ;将 R1 - R7,LR 存放到堆栈中,这条指令一般用来保护现场
```

3.2.8 相对寻址

相对寻址可以看做是寄存器变址寻址方式的一个特例，因为此时包含基地址的寄存器特指程序计数器 PC，通过 PC 值与指令中的偏移量结合，生成有效的操作数地址。一般这种寻址方式用于指令跳转，如：

```
BL Label                              ;转跳到 Label 标签处
…
Label:
…
```

3.3 ARM 指令简介

本节将具体介绍 ARM 指令集中各类常用指令的用法和注意点。

3.3.1 跳转指令

ARM 转跳指令主要用于：向后跳转实现循环；通过条件判断实现现在跳转；子程序调用；切换处理器工作状态。ARM 实现程序跳转有两种方法：

第一种方法是将当前的程序寄存器 PC 值改写为跳转的目的地址，此时可以实现 4G 地址范围内的长跳转。通常使用的方法有两种，可以使用指令：

```
MOV PC, #immediate; PC<- immediate
```

此处要注意的问题还是立即数的合法性问题，也正是如此，这种方法并不能做到跳转到任意地址，而另一种方法则可以保证跳转的任意性：

```
LDR PC, [PC, #offset]; PC<- [PC + offset]
```

此时跳转的目标地址被预先存放于存储器中，通过存储器读取指令将其赋值给 PC。但这个方法也有不足之处，就是存储单元的地址距当前的指令地址不能超过 4KB 的范围，这是因为给偏移量 offset 分配的空间只有 12 位的大小(MOV 指令和 LDR 指令的具体用法会在下文详细介绍)。

第二种实现程序跳转的方法就是使用专门的跳转指令实现，在 ARM 中包括 B、BL、BX 和 BLX 指令。其实在 3.1.3 介绍 ARM 状态和 Thumb 状态时，读者已经初步认识了 BL、BX 和 BLX 指令的作用和实现方式，在这里将更详细的进行说明。

B：(Branch)

B 指令是基本的转跳指令，它的格式为：

```
B{cond}, Label
```

cond 表示指令的条件域，它可以是 2.1.3 节中列出的 15 种可能条件中的一种。Label

并不是一个绝对跳转地址，而只是表示相对于当前指令地址的偏移。它是一个 24 位的带符号数，在实际寻址过程中，由于 ARM 采用 32 位对齐方式，故 Label 会左移两位，然后符号扩展到 32 位，实际有效的寻址空间范围为：当前指令地址±32M，即使用 B 指令可以在当前指令的前后 32M 范围内实现跳转。但这个当前指令的地址是多少需要注意，ARM9 采用 5 级流水线，计算偏移在第三级流水线上，因此在此条指令之后已经预取两条指令，故当前指令的地址是 PC-8。

BL：（Branch with Link）

BL 是带链接的跳转指令，所谓带链接是指在跳转过程发生之前，会先将下一条要执行的指令地址存放到链接寄存器 R14 中，这条指令一般用于函数的调用，当函数执行完成时，只要将 R14 中的值恢复到 PC 中，便可以实现函数的返回。BL 指令的格式和注意事项基本与 B 指令相同，格式如下：

```
BL{cond}, Label
```

BX：（Branch and eXchange）

BX 指令用于 ARM 状态和 Thumb 状态之间的切换，它将通用寄存器 Rm(R0-R15)的值复制到程序寄存器 PC 中来实现 4G 地址范围的绝对跳转。BX 指令通过判断 Rm 所存储的目标地址的最后一位来判断要跳转到什么状态。在 ARM 状态下，若 Rm[0]=1，BX 指令将 CPSR 的 T 位置位，然后跳转到 Thumb 状态；若 Rm[0]=0，则 Rm[1]必须为 0，以保证 ARM 指令的字对齐。BX 指令的格式如下：

```
BX{cond}, Rm
```

Rm 存储了目标跳转地址，具体跳转过程可以看一个具体的汇编程序例子：

```
CODE32
…
ADR R1, Label + 1
BX R1
Label2
…
CODE16
…
    LDR R2, = Label2
    BX R2
…
```

程序中 Label+1 是为了指明这个跳转指令要切换到 Thumb 状态。因为 ARM 指令是字对齐的，故指令地址最低两位[1:0]=0b00；而 Thumb 指令是半字对齐的，指令地址最低位[0]=0。所以无论指令属于哪种类型，指令地址的最低位必为 0，为确保这一地址特性，在执行 BX 指令时处理器会自动将 R1 的值和 0xFFFFFFFE 进行与操作，以得到合法的目标地址。但在执行这个与操作之前，处理器会判断指令最后一位是 0 或 1，以决定转跳的状态类型，所以在上面的程序中能够正确切换到 Thumb 状态。

BLX：（Branch with Link and eXchange）

这是 v5 版本后才出现的命令，它能够在一条指令内完成指令跳转、返回位置保存和处

理器工作状态切换三个动作。它有两种格式，一种是目标地址为任意绝对地址的带条件跳转，另一种是目标地址为当前程序相对地址的无条件跳转，格式如下：

```
BLX{cond}, Rm
BLX Label
```

第一种带条件跳转的注意事项和 BX 指令基本相同，而第二种无条件跳转是指执行这条指令肯定会引起状态切换。Label 同样是程序相对地址，转跳范围是：当前指令地址±32M。用法有：

```
BLX R2
BLXNE R2
BLX Thumblabel
```

注意一下指令用法是错误的，开始学习时容易犯这样的错误：

```
BLXMI Thumblabel                    ;相对地址跳转指令必须是无条件的
```

3.3.2 通用数据处理指令

ARM 的通用数据处理指令大致可以分为 4 类：数据传送指令、算术逻辑运算指令、比较指令和前导零计数指令。通用数据传送指令实现寄存器和存储器之间的双向传输；算术逻辑运算指令执行算术和逻辑运算，如加减、与或操作等；比较指令通常将一个寄存器的值与 32 位的常数进行比较或测试；前导零计数指令只有一条，即 CLZ，用于统计寄存器数据的前导零个数。ARM 数据处理指令可以选择使用 S 后缀，这样在执行指令时会同时影响 CPSR 的条件标志位。比较指令 CMP，CMN，TST 和 TEQ 无论加不加 S 都会影响标志位，所以不需要加 S 后缀。

通用数据处理指令有几个注意事项：

第一，当将 R15 作为 Rd 时，如 3.3.1 所述，可以完成跳转功能。若此时加上 S 后缀，SPSR 的当前模式会复制到 CPSR，这样能够完成从异常模式的返回。但必须注意，因为用户模式和系统模式不属于异常模式，所以不用在这两种模式下使用 S 后缀，否则会造成不可以预知的后果，因为汇编编译器在编译阶段是不会报告这个警告的。

第二，当指令中包含寄存器控制的移位操作时，不能够将 R15 用作 Rd 或是任何的操作数。

第三，由于 ARM9 采用 5 级流水线，执行数据操作的 ALU 在第三级流水线上，因此在此条指令之后已经预取两条指令，当 R15 用作 Rn 时，它的值是当前指令地址加 8，即当前的 PC 值。

(1) 数据传送指令

MOV(MOVE)和 MVN(MOVE NOT)

这两条指令的格式为：

```
MOV{cond}{S} Rd, Operand2
MVN{cond}{S} Rd, Operand2
```

MOV 指令将 Operand2 的值复制到 Rd 寄存器中，而 MVN 指令会先将 Operand2 按位