第 3 章　Cortex-M3 处理器的指令系统

【导读】 Cortex-M3 处理器支持的指令集包括 ARMv6 的大部分 16 位 Thumb 指令和 ARMv7 的 Thumb-2 指令集。本章给出了指令的一般格式，并详细说明了存储器访问指令、数据处理指令、乘法除法指令、位操作指令和分支控制指令等使用，最后在说明符号定义伪指令、数据定义伪指令、汇编控制伪指令和宏指令的使用方法后，举例说明了汇编程序的编写方法。目前嵌入式开发主要以高级语言为主，汇编语言作为性能调优和底层初始化使用，本章介绍的指令集旨在让读者了解指令的种类、功能和基本使用方法，读懂汇编程序代码。

3.1　Cortex-M3 处理器的指令系统

3.1.1　指令系统基本概念

1. 指令和指令集

冯·诺依曼体系结构采用存储程序的原则，即事先将程序存储到计算机中，程序是由计算机可执行的指令组成的，计算机的控制器根据程序指令控制计算机自动运行，即计算机执行程序的过程就是执行一条条指令的过程。

指令是指 CPU 能直接识别并执行的控制命令，它的表现形式是二进制编码。指令通常由操作码和操作数两部分组成，操作码指出该指令所要完成的操作，即指令的功能，操作数指出参与运算的数据或其存储地址等。

【思考题：为何指令中要使用数据存储地址而非直接使用数据?】

由于指令与 CPU 紧密相关，不同类型的 CPU 所对应的指令也有所不同，一台计算机所能执行的各种不同指令的全体，称为计算机的指令系统。一般同一系列的 CPU 指令集具有兼容性，即新的指令系统必须包括先前同系列 CPU 的指令系统，这样先前开发的各类程序在新 CPU 上才能正常运行。

机器语言是用来直接描述指令的最佳语言，是 CPU 能直接识别的唯一一种计算机语言，但机器语言书写大规模程序不容易维护。汇编语言使用助记符来代替和表示特定机器语言操作，相对机器语言更容易理解和维护，且汇编语言中可使用标签和符号等代表操作数或功能模块，使得程序更加灵活。一般汇编语言和其特定的机器语言描述的指令集是一一

对应的，但 CPU 无法识别汇编语言，因此需要汇编器进行转换。汇编语言目前已不像十几年前被广泛用于程序设计，只在操作系统、驱动程序等底层硬件操作和极高要求的程序优化场合下使用。

指令按照不同的分类可划分不同的种类。按照指令实现的功能，指令可分为：数据传送指令、算术运算指令、逻辑运算指令、程序控制指令、系统控制指令等。数据传送指令用来实现主存与 CPU 寄存器以及寄存器与寄存器之间的数据传输，例如 Thumb-2 的取一个字到寄存器指令 LDR、寄存器数据存储到主存指令 STR，寄存器间数据传送指令 MOV 等。算术运算是计算机能够执行的基本数值计算，算术运算指令包括加法 ADD、减法 SUB、乘法 MUL、除法 DIV 等指令。逻辑运算是对数据进行逻辑操作，包括逻辑与 AND、逻辑或 ORR、逻辑非 MVN 等三种基本操作以及同或、异或等组合逻辑操作。程序控制指令主要包括：转移指令 BL、断点指令 BKRT、分支指令 IT 等。系统控制指令包括休眠指令 WFI、WFE、空操作指令 NOP、开关中断指令 CPSIE、CPSID 等。

按照数据存取方式，指令可分为寄存器-寄存器型、寄存器-存储器型和存储器-存储器型。寄存器-寄存器型将数据存放在寄存器中进行操作，例如 Cortex-M3 的大多数数据传输和算术逻辑运算指令，寄存器-寄存器型指令编码简单、执行速度快，指令周期相近。寄存器-存储器型指令可直接对存储器操作数进行访问，如访存操作 LDR、STR 等，指令周期相差较大，执行速度较慢。存储器-存储器型无需寄存器保存数据，其执行的访存较多，执行速度慢，Cortex-M3 绝大多数指令不采用存储器-存储器结构。

2. Cortex-M3 指令的编码方式

Cortex-M3 支持 16 位和 32 位的 Thumb-2 指令集，一个典型的 16 位指令的编码如图 3-1 所示。

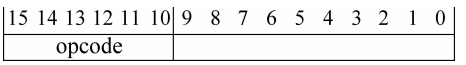

图 3-1　Thumb-2 16 位指令编码格式

16 位指令的操作码部分通过 6 个 bit 进行分类，如表 3-1 所示，每一类指令根据具体情况进行二次编码，因此，Cortex-M3 的指令操作码部分是不等长的，但可以通过多级译码实现，每一级译码的操作码是等长的，由此实现了指令的灵活性和复杂性的均衡。

表 3-1　opcode 分类定义

操作码 opcode	指令或指令类
00xxxx	立即数寻址移位、加法、减法、送数和比较指令
010000	数据处理指令
010001	特殊的数据、分支和交换指令
01001x	寄存器偏移寻址加载指令
0101xx 011xxx 100xxx	单数据加载/存储指令

续表

操作码 opcode	指令或指令类
10100x	PC 相关寻址类指令
10101x	SP 相关寻址类指令
1011xx	16 位的其他指令
11000x	多寄存器存储治理，如 STM，STMIA，STMEA
11001x	多寄存器加载指令，如 LDM，LDMIA，LDMFD
1101xx	有条件转移、SVC 指令
11100x	无条件转移指令

对于 32 位指令，高 16 位的操作码 opcode 的取值为 11101、11110 和 11111，此时处理器会将下一个 16 位和当前 16 位组合成一个 32 位指令，如图 3-2 所示（即 op1 的取值为 01，10 和 11，当 op1 为 00 时，表明是一条 16 位指令）。

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	op1				op2									op															

图 3-2　Thumb-2 32 位指令的编码格式

两个寄存器数据的逻辑与 ADD 和逻辑或 ORR 指令编码如图 3-3 所示，从表 3-1 中可以看出逻辑与和逻辑或属于数据处理指令类，数据处理指令又采用了 4 个比特进行了二次编码。

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	0	0	0	Rm			Rdn		

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	1	1	0	0	Rm			Rdn		

图 3-3　逻辑与和逻辑或的 16 位指令编码

【思考题：Thumb2 指令编码是否违反了 RISC 设计思想？】

3.1.2　指令格式

Cortex-M3 不支持 ARM 指令集，支持的指令集包括 ARMv6 的大部分 16 位 Thumb 指令和 ARMv7 的 Thumb-2 指令集。Thumb-2 指令集是一个 16/32 位混合的指令系统。

指令的一般格式如下：

```
<opcode>{<cond>}{S}{.N|.W} <Rd>,<Rn>{,<operand2>}
```

- opcode 是操作码，如 ADD、LDR 和 STR 等，规定所执行的具体操作；

- cond 是可选的条件码，如 EQ、NE 和 CS 等，规定指令执行所满足的条件，条件码的说明见表 3-2；
- S 是可选后缀，若指定 S，则需要根据指令执行结果去更新程序状态寄存器 xPSR 相应的标志位；
- .N 表示 16 位指令，.W 表示 32 位指令，默认为 16 位指令；
- Rd 是目标寄存器；
- Rn 是存放第 1 个操作数的寄存器；
- operand2 是第 2 个操作数。

这里，符号{ }和< >的含义如下：

- { }表示可选的，例如{<cond>}表示条件码是可选的，可以有条件码也可无条件码；
- < >表示必需的，例如<Rd>表示必须有目标寄存器。

表 3-2 条件码定义

后 缀	标 志	含 义
EQ	Z=1	相等
NE	Z=0	不相等
CS or HS	C=1	无符号数大于或等于
CC or LO	C=0	无符号数小于
MI	N=1	负的
PL	N=0	正的或为 0
VS	V=1	溢出
VC	V=0	无溢出
HI	C=1 and Z=0	无符号数大于
LS	C=0 or Z =1	无符号数小于或等于
GE	N=V	有符号数大于或等于
LT	N!=V	有符号数小于
GT	Z =0 and N=V	有符号数大于
LE	Z =1 or N!=V	有符号数小于或等于
AL		无条件执行

条件执行是 ARM 处理器的一个优化程序速度的典型方式，可以减少不必要的跳转。如图 3-4 所示的 C 语言代码的 ARM 汇编结果：

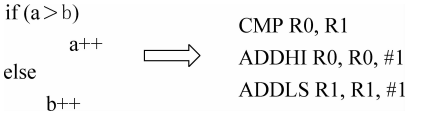

图 3-4 条件执行汇编指令

3.1.3　寻址方式

寻址方式是根据指令中给出的操作数字段来实现寻找真实操作数的方式，Cortex-M3 支持 8 种寻址方式。

1）寄存器寻址

操作数的值在寄存器中，指令中的地址码字段给出的是寄存器编号，寄存器的内容是操作数，指令执行时直接取出寄存器值操作，例如：

```
MOV  R1,R2          ;R1←R2
SUB  R0,R1,R2       ;R0←R1-R2
```

2）立即数寻址

数据就包含在指令当中，立即寻址指令的操作码字段后面的地址码部分就是操作数本身，取出指令也就取出了可以立即使用的操作数（也称为立即数）。立即数要以“#”为前缀，表示十六进制数值时以“0x”表示，例如：

```
ADD  R0,R0,#1       ;R0←R0+1
MOV  R0,#0xff00     ;R0←0xff00
```

3）寄存器移位寻址

寄存器移位寻址是把第 2 个寄存器操作数移位之后送给第 1 个操作数，例如：

```
MOV  R0,R2,LSL #3          ;R2 的值左移 3 位，结果放入 R0，即 R0=R2X8。
AND  R1,R1,R2,LSL R3       ;R2 的值左移 R3 位，然后和 R1 相与操作，结果放入 R1。
```

可采用的移位操作如下：

- LSL：逻辑左移（Logical Shift Left），寄存器中字的低端空出的位补 0。
- LSR：逻辑右移（Logical Shift Right），寄存器中字的高端空出的位补 0。
- ASR：算术右移（Arithmetic Shift Right），移位过程中保持符号位不变，即如果源操作数为正数，则字的高端空出的位补 0，否则补 1。
- ROR：循环右移（Rotate Right），由字的低端移出的位填入字的高端空出的位。
- RRX：带扩展的循环右移（Rotate Right extended by 1 place），操作数右移一位，高端空出的位用进位标志 C 的值填充。

4）寄存器间接寻址

指令中的地址码给出的是一个通用寄存器编号，所需要的操作数保存在该寄存器的值作为地址的存储单元中，即寄存器为操作数的地址指针，操作数存放在存储器中，例如：

```
LDR  R0,[R1]   ;R0←[R1](将 R1 的值作为地址，取出此地址中的数据保存在 R0 中)
STR  R0,[R1]   ;[R1]←R0(将 R0 的值写入到以 R1 的值为地址的存储器空间中)
```

5）变址寻址

变址寻址是将基址寄存器的内容与指令中给出的偏移量相加，形成操作数的有效地址，

变址寻址用于访问基址附近的存储单元，常用于查表，数组操作，外设控制器的内部寄存器访问等，例如：

```
LDR  R2,[R3,#4]    ;R2←[R3+4](将 R3 的数值加 4 作为地址,取出此地址的数值保存在 R2 中)
STR  R1,[R0,#-2]   ;[R0-2]←R1(将 R0 的数值减 2 作为地址,把 R1 中的内容保存到此地址的
                   ;存储单元中)
```

6）多寄存器寻址

采用多寄存器寻址方式，一条指令可以完成多个寄存器值的传送，这种寻址方式用一条指令最多可以完成 16 个寄存器值的传送，例如：

```
LDMIA  R0,{R1,R2,R3,R5}  ;将 R0,R0+4,R0+8,R0+12 地址处的数据分别送到寄存器 R1,R2,
                         ;R3 和 R5 中,R0 的值保持不变
STMIA  R0!,{R1-R7}       ;将 R1~R7 的数据保存到存储器中,存储器指针 R0 在保存第一个值
;之后增加,增长方向为向上增长,即 R1~R7 的值存储在 R0,R0+4,R0+8,R0+12,R0+16,R0+20,R0
;+24 的地址中,指令执行完后,R0 的值变成 R0+24
STMDA  R0!,{R1-R7}       ;将 R1~R7 的数据保存到存储器中,存储器指针 R0 在保存第一个值
;之后减少,增长方向为向下增长,即 R1~R7 的值存储在 R0-24,R0-20,R0-16,R0-12,R0-8,R0-4,
;R0 的地址中,指令执行完后,R0 的值变成 R0-24
```

7）栈寻址

栈寻址是通过栈指针 R13 进行数据读写的方式，如 POP 和 PUSH 操作等，其数据存储和加载的地址由 SP 寄存器隐含给出，例如：

```
PUSH  {R0-R3}            ;将 R0,R1,R2,R3 四个寄存器的值压入栈中
POP   {R0-R2}            ;将栈顶的数据依次读取到 R0,R1,R2 中
LDMIA  SP!,{R1,R2,R3,R5} ;将栈顶的数据依次读入到 R1,R2,R3,R5 中
```

8）相对寻址

相对寻址是变址寻址的一种变通，由程序计数器 PC 提供基准地址，指令中的地址码字段作为偏移量，两者相加后得到的地址即为操作数的有效地址，例如：

```
    LDR  R2,[PC,#4]      ;R2←[PC+4](将 PC 加 4 作为地址,取出此地址的数保存在 R2 中)
    BL   ROUTE1          ;调用到 ROUTE1 子程序,等价于 LDR  PC,[PC,#6]
    BEQ  LOOP            ;条件跳转到 LOOP 标号处,等价于 LDR  PC,[PC,#2]
LOOP
    MOV R2,#2
ROUTE1
    MOV R1,#3
```

3.1.4 数据传送指令

最基本的数据传送指令是寄存器间的数据传送，此外，还包括立即数加载到寄存器，特殊寄存器的读写指令等。数据传送指令包括 MOV、MVN、MRS 和 MSR 等，具体指令格式

及其功能如表 3-3 所示。

表 3-3　数据传送指令

示　　例	功 能 描 述
MOV ＜Rd＞，＃＜immed_8＞	将 8 位立即数传到目标寄存器
MOV ＜Rd＞，＜Rn＞	将寄存器值传给低目标寄存器
MVN ＜Rd＞，＜Rm＞	寄存器值取反后传给目标寄存器
MOV{S}. W ＜Rd＞，＃＜immed_12＞	将 12 位立即数传送到寄存器中
MOV{S}. W ＜Rd＞，＜Rm＞{，＜shift＞}	将移位后的寄存器值传到寄存器
MOVT. W ＜Rd＞，＃＜immed_16＞	将 16 位立即数传送到寄存器的高半字[31:16]
MOVW. W ＜Rd＞，＃＜immed_16＞	16 位立即数传到寄存器的低半字[15:0]，将高半字[31:16]清零
MRS ＜Rd＞，　＜SReg＞	读特殊功能寄存器 SReg 到寄存器 Rd
MSR ＜SReg＞，＜Rn＞	写 Rn 到特殊功能寄存器 SReg

例如：

```
MOV  R0, #8              ;R0=8
MOV  R1,  R0             ;R1=R0=8
MVN  R2,  R1             ;R2=0xFFFFFFF7
MOV.W  R4,  R0 LSL, #2   ;R4=32
MOVW.W  R1,#0x1234       ;R1 =0x1234
MOVT.W  R1,#0x5678       ;R1 =0x56781234
MRS  R1  APSR            ;将 R1 的值写入到状态寄存器 APSR
```

3.1.5　存储器访问指令

存储器访问指令包括存储器寄存器传输指令 LDR、STR；多寄存器加载指令 LDM、多寄存器存储指令 STM 以及压栈指令 PUSH 和出栈指令 POP 等。

1）加载指令 LDR 和存储指令 STR

LDR 实现从存储器中加载操作数到寄存器，STR 实现从寄存器存储数据到存储器。LDR 和 STR 的语法格式为：

语法格式：

```
op {type}{cond} Rt, [Rn{, #offset}]     ;立即偏移
op {type}{cond} Rt, [Rn, #offset]!      ;前变址
op {type}{cond} Rt, [Rn], #offset       ;后变址
```

这里，op 是 LDR 或 STR。type 指定传送的是字节还是半字，缺省为传送一个字。cond 是可选条件码，Rt 是指定的加载或存储的寄存器，Rn 是存储器地址存放的寄存器，

offset 是偏移量。

type 可以是：

- B：传送无符号的字节；
- SB：传送有符号的字节；
- H：传送无符号的半字；
- SH：传送有符号的半字。

LDR 和 STR 指令进行数据加载和存储时，涉及三种寻址方式，分别为立即偏移的基址变址寻址方式、前变址的基址变址寻址方式和后变址的基址变址寻址方式。这里，基址寄存器为 Rn。

- 立即偏移的基址变址寻址方式，例如 LDR R0，[R1，#4]，这种寻址方式是将基址寄存器 R1 的内容和偏移量 4 相加形成操作数的有效地址；操作完成后，基址寄存器 R1 的内容不变。
- 前变址的基址变址寻址方式，例如 LDR R0，[R1，#4]!，这种寻址方式是将基址寄存器 R1 的内容和偏移量 4 相加形成操作数的有效地址；操作完成后，基址寄存器 R1 的内容更新为新的地址，即 R1＝R1＋4。
- 后变址的基址变址寻址方式，例如 LDR R0，[R1]，#4，这种寻址方式直接将基址寄存器 R1 的内容作为操作数的有效地址；操作完成后，基址寄存器 R1 的内容更新为原有内容和偏移量之和，即 R1＝R1＋4。

使用举例：

① LDR R6, [R10]

存储器操作数是寄存器间接寻址方式，直接将 R10 寄存器的内容作为操作数的有效地址，该指令实现将有效地址的存储器操作数加载到 R6 寄存器；

② LDRNE R6, [R5, #960]!

这是带条件码的指令，当程序状态寄存器的 Z 标志位为 0 时才执行该指令。存储器操作数是前变址的基址变址寻址方式，将 R5 寄存器的内容和偏移量＃960 相加作为操作数的有效地址，该指令实现将有效地址的存储器操作数加载到 R6 寄存器；操作完成后，R5 寄存器的内容增加 960。

③ STRH R6, [R4], #4

这是一个无符号半字的存储指令。存储器操作数是后变址的基址变址寻址方式，直接将 R4 寄存器的内容作为操作数的有效地址，该指令实现将有效地址的存储器操作数（无符号半字）加载到 R6 寄存器；操作完成后，R4 寄存器的内容增加 4。

如果需要对双字进行存取，指令的格式为：

```
opD {cond} Rt, Rt2, [Rn{, #offset}]     ;立即偏移, 双字指令
opD {cond} Rt, Rt2, [Rn, #offset]!      ;前变址, 双字指令
opD {cond} Rt, Rt2, [Rn], #offset       ;后变址, 双字指令
```

其中，op 为 STM 或 LDR，Rt 和 Rt2 为双字存取的寄存器，Rn 为地址寄存器，例如：STRD R1，R2，[R8]，#-16，这是一个双字存储指令。存储器操作数是后变址的基址变址寻址方式，将 R1 的内容存储到 R8 寄存器所指定的有效地址中，将 R2 的内容存储到 R8 寄存器的内容＋4 所指定的有效地址中；操作完成后，R8 寄存器的内容减少 16。存储器访问指令的指令格式和功能详见表 3-4。

表 3-4　存储器数据加载和存储指令

示　例	功 能 描 述
LDRB Rd，[Rn，#offset]	从地址 Rn＋offset 处读取一个字节送到 Rd
LDRH Rd，[Rn，#offset]	从地址 Rn＋offset 处读取一个半字送到 Rd
LDR Rd，[Rn，#offset]	从地址 Rn＋offset 处读取一个字送到 Rd
LDRD Rd1，Rd2，[Rn，#offset]	从地址 Rn＋offset 处读取一个双字(64 位整数)送到 Rd1(低 32 位)和 Rd2(高 32 位)中
STRB Rd，[Rn，#offset]	把 Rd 中的低字节存储到地址 Rn＋offset 处
STRH Rd，[Rn，#offset]	把 Rd 中的低半字存储到地址 Rn＋offset 处
STR Rd，[Rn，#offset]	把 Rd 中的低字存储到地址 Rn＋offset 处
STRD Rd1，Rd2，[Rn，#offset]	把 Rd1(低 32 位)和 Rd2(高 32 位)表达的双字存储到 Rn＋offset 处
LDR.W Rd，[Rn，#offset]!	字的带预索引加载
LDRB.W Rd，[Rn，#offset]!	字节的带预索引加载(不扩展符号位)
LDRH.W Rd，[Rn，#offset]!	半字的带预索引加载(不扩展符号位)
LDRD.W Rd1，Rd2，[Rn，#offset]!	双字的带预索引加载(不扩展符号位)
LDRSB.W Rd，[Rn，#offset]! LDRSH.W Rd，[Rn，#offset]!	字节/半字的带预索引加载，并且在加载后执行带符号扩展成 32 位整数
STR.W Rd，[Rn，#offset]!	字/字节/半字/双字的带预索引存储
STRB.W Rd，[Rn，#offset]!	字节的带预索引存储
STRH.W Rd，[Rn，#offset]!	半字的带预索引存储
STRD.W Rd1，Rd2，[Rn，#offset]!	双字的带预索引存储

2) 批量加载指令 LDM 和批量存储指令 STM

LDM 实现由基址寄存器指示的一片连续存储器中的数据批量加载到多个寄存器，STM 实现将多个寄存器中的数据批量存储到由基址寄存器指示的一片连续存储器。

语法格式：

```
op {addr_mode} {cond} Rn{!}, reglist
```

这里，op 是 LDM 或 STM，addr_mode 是地址变化模式，cond 是可选条件码，Rn 是基址寄存器，! 为可选的回写后缀，若选用该后缀，则数据传送完毕后，将最后的地址写入基址寄

存器，reglist 是多个数据加载或存储的寄存器列表。

addr_mode 可以取下列值：

- IA(Increment After)意味着在每一次访问之后地址递增，如图 3-5(a)所示。
- IB(Increment Before)意味着每一次访问之前地址递增，如图 3-5(b)所示。
- DA(Decrement After)意味着每一次访问之后地址递减，如图 3-5(c)所示。
- DB(Decrement Before)意味着在每一次访问之前地址递减，如图 3-5(d)所示。

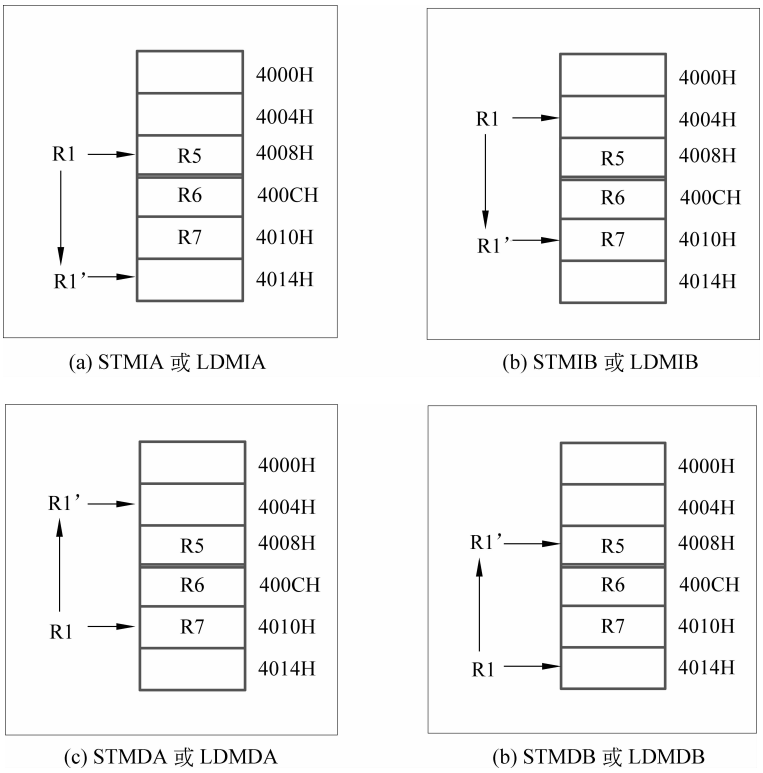

图 3-5　批量加载和批量存储指令中的地址模式

指令举例如下：

① LDM R8, {R0,R2,R9}

该指令实现将 R8 指示的存储器的连续内容加载到三个寄存器 R0、R2 和 R9。R8 的值保持不变。

② STMDB R1!,{R3-R6,R11,R12}

该指令实现将 R3-R6、R11 和 R12 寄存器中的内容存储到 R1 指示的存储器中，并将最后一个存储单元的地址回写到 R1 寄存器。

在多寄存器存储指令中，基址寄存器不能是 PC，寄存器列表中不能含有 SP；若是 STM 指令，寄存器列表一定不能含有 PC；若是 LDM 指令，若寄存器列表中含有 LR 则一定不能

含有 PC；如果增加回写后缀，则寄存器列表中一定不能含有基址寄存器。多寄存器存储指令的指令格式和功能详见表 3-5。

表 3-5　多寄存器访存指令

示　　例	功 能 描 述
LDMIA Rd!，{寄存器列表}	16 位指令，从 Rd 处读取多个字，并依次送到寄存器列表中的寄存器。每读一个字后 Rd 自增一次
STMIA Rd!，{寄存器列表}	16 位指令，存储寄存器列表中各寄存器的值依次存储到 Rd 给出的地址。每存一个字后 Rd 自增一次
LDMIA. W Rd!，{寄存器列表}	32 位指令，从 Rd 处读取多个字，并依次送到寄存器列表中的寄存器。每读一个字后 Rd 自增一次
LDMDB. W Rd!，{寄存器列表}	32 位指令，从 Rd 处读取多个字，并依次送到寄存器列表中的寄存器。每读一个字前 Rd 自减一次
STMIA. W Rd!，{寄存器列表}	32 位指令，依次存储寄存器列表中各寄存器的值到 Rd 给出的地址。每存一个字后 Rd 自增一次
STMDB. W Rd!，{寄存器列表}	32 位指令，存储多个字到 Rd 处。每存一个字前 Rd 自减一次

3）压栈指令 PUSH 和出栈指令 POP

当栈指针指向最后压入栈的数据时，称为满栈（Full Stack）；当栈指针指向下一个将要放入数据的空位置时，称为空栈（Empty Stack）。压栈时，栈由低地址向高地址生长时，称为递增栈（Ascending Stack），栈由高地址向低地址生长时，称为递减栈（Descending Stack）。由此可以组合四种栈模型：递增满栈、递增空栈、递减满栈、递减空栈。Cortex-M3 使用的是向下生长的满栈模型。栈指针 SP 指向最后一个被压入栈的 32 位数值。在下一次压栈时，SP 先自减 4，再存入新的数值，如图 3-6 所示。

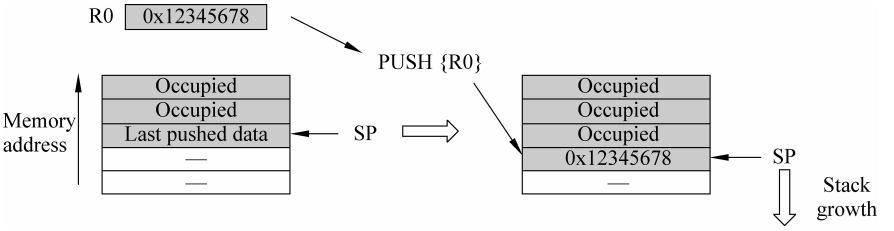

图 3-6　压栈操作过程

出栈操作刚好相反：先从 SP 指针处读出上一次被压入的值，再把 SP 指针自增 4，如图 3-7 所示。

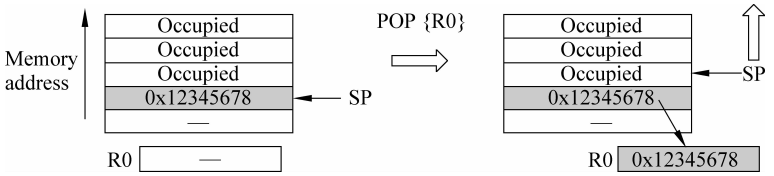

图 3-7　出栈操作过程

PUSH 实现以递减满栈方式的压栈操作；POP 实现以递减满栈方式的出栈操作。因此，PUSH 相当于指令 STMDB 的功能，POP 相当于指令 LDMIA 的功能。

语法格式：

```
PUSH {cond} reglist
POP  {cond} reglist
```

这里，cond 是可选条件码；reglist 是压栈或出栈需要的寄存器列表。

寄存器列表中一定不能含有 SP；对于 PUSH 指令，寄存器列表一定不能含有 PC；对于 POP 指令，如果寄存器列表中含有 LR，则一定不能含有 PC。

使用举例：

① `PUSH {R0, R4-R7}`

该指令实现将寄存器列表 R0、R4、R5、R6 和 R7 中数据压入栈。

② `POP {R0,R6,PC}`

该指令实现将寄存器列表 R0、R6 和 PC 中的数据弹出栈。

3.1.6 算术运算指令

算术运算指令主要包括加减乘除操作，其指令格式和功能如表 3-6 所示。

表 3-6 算术运算指令

示 例		功 能 描 述
ADD Rd, Rn, Rm	; Rd = Rn+Rm	常规加法
ADD Rd, Rm	; Rd += Rm	
ADD Rd, #imm	; Rd += imm	im8(16 位指令)或 im12(32 位指令)
ADC Rd, Rn, Rm	; Rd = Rn+Rm+C	带进位的加法， Im8 或 im12
ADC Rd, Rm	; Rd += Rm+C	
ADC Rd, #imm	; Rd += imm+C	
SUB Rd, Rn	; Rd -= Rn	常规减法
SUB Rd, Rn, #imm3	; Rd = Rn-imm3	
SUB Rd, #imm8	; Rd -= imm8	
SUB Rd, Rn, Rm	; Rd = Rn-Rm	
SBC Rd, Rm	; Rd -= Rm+C	带借位的减法
SBC.W Rd, Rn, #imm12	; Rd = Rn-imm12-C	
SBC.W Rd, Rn, Rm	; Rd = Rn-Rm-C	

续表

示　例		功 能 描 述
RSB. W Rd, Rn, #imm12	; Rd = imm12−Rn	反向减法
RSB. W Rd, Rn, Rm	; Rd = Rm−Rn	
MUL Rd, Rm	; Rd *= Rm	常规乘法
MUL. W Rd, Rn, Rm	; Rd = Rn * Rm	
MLA Rd, Rm, Rn, Ra	; Rd = Ra+Rm * Rn	乘加
MLS Rd, Rm, Rn, Ra	; Rd = Ra−Rm * Rn	乘减
UDIV Rd, Rn, Rm	; Rd = Rn/Rm	无符号除法，硬件支持的除法，余数被丢弃
SDIV Rd, Rn, Rm	; Rd = Rn/Rm	带符号除法，硬件支持的除法，余数被丢弃
SMULL RL, RH, Rm, Rn	;[RH: RL]= Rm * Rn	带符号的 64 位乘法
SMLAL RL, RH, Rm, Rn	;[RH: RL]+= Rm * Rn	
UMULL RL, RH, Rm, Rn	;[RH: RL]= Rm * Rn	无符号的 64 位乘法
UMLAL RL,RH, Rm, Rn	;[RH: RL]+= Rm * Rn	

1) ADD、ADC、SUB、SBC 和 RSB

- ADD 实现第 2 个操作数 Operand2 或立即数 imm 与第 1 个操作数 Rn 相加，并将结果送至 Rd 目标寄存器。
- ADC 实现第 2 个操作数 Operand2 与第 1 个操作数 Rn 以及进位标志相加，并将结果送至 Rd 目标寄存器。
- SUB 实现第 1 个操作数 Rn 与第 2 个操作数 Operand2 或立即数 imm 相减，并将结果送至 Rd 目标寄存器。
- SBC 实现第 1 个操作数 Rn 与第 2 个操作数 Operand2 相减，再减去 C 条件标志位的反码，并将结果送至 Rd 目标寄存器。
- RSB 实现第 2 个操作数 Operand2 与第 1 个操作数 Rn 相减，并将结果送至 Rd 目标寄存器。

使用举例：

① ADD R2, R1, R3

该指令实现 R1 寄存器和 R3 寄存器的内容相加，并将结果送至 R2 寄存器，即 R2=R1+R3。

② SUBS R7, R5, #256

该指令实现 R5 寄存器的内容与立即数 #256 相减，并将结果送至 R7 寄存器，即 R7=R5-256。同时，根据运算结果影响标志位。

③ RSB R8, R8, #240

该指令实现立即数 240 与 R8 寄存器的内容相减，并将结果送至 R8 寄存器，即 R8＝240-R8。

④ ADCHI R10, R0, R3

当 C＝1 且 Z＝0 成立时，执行该指令。该指令实现将 R0 寄存器与 R3 寄存器的内容以及进位标志相加，并将结果送至 R10 寄存器，即 R10＝R0＋R3＋C。

2）MUL、MLA 和 MLS

- MUL 指令实现将 Rn 寄存器和 Rm 寄存器的内容相乘，并将结果存入 Rd 寄存器。
- MLA 指令实现将 Rn 寄存器和 Rm 寄存器的内容相乘，并将乘法结果和 Ra 寄存器的内容相加，再将最终结果存入 Rd 寄存器。
- MLS 指令实现将 Rn 寄存器和 Rm 寄存器的内容相乘，并将 Ra 寄存器中的内容与乘法结果相减，再将最终结果存入 Rd 寄存器。

使用举例：

① MUL R7, R2, R5

该指令实现将 R2 寄存器的内容和 R5 寄存器的内容相乘，然后将乘法结果存入 R10 寄存器，即 R10＝R2 * R5。

② MLA R9, R3, R4, R6

该指令实现将 R3 寄存器的内容和 R4 寄存器的内容相乘，然后将乘法结果再与 R6 寄存器内容相加，最后将计算结果存入 R9 寄存器，即 R9＝R3 * R4＋R6。

③ MLS R10, R5, R6, R7

该指令实现将 R5 寄存器的内容和 R6 寄存器的内容相乘，然后将 R7 寄存器的内容与乘法结果相减，最后将计算结果存入 R10 寄存器，即 R10＝R7-R5 * R6。

3）SDIV 和 UDIV

- SDIV 实现有符号整数相除，将 Rn 寄存器的内容与 Rm 寄存器的内容相除，计算结果存入 Rd 寄存器。
- UDIV 实现无符号整数相除，将 Rn 寄存器的内容与 Rm 寄存器的内容相除，计算结果存入 Rd 寄存器。

除法指令同样不能使用 SP 或 PC，对条件标志位没有影响。

使用举例：

① SDIV R0, R1, R2

该指令实现 R1 寄存器内容与 R2 寄存器内容的有符号数相除，计算结果存入 R0 寄存器，即 R0＝R1/R2。

② UDIV R7, R7, R3

该指令实现 R1 寄存器内容与 R2 寄存器内容的有无号数相除，计算结果存入 R7 寄存器，即 R7＝R7/R3。

3.1.7　逻辑运算指令

逻辑运算指令包括与、或、非基本操作及其扩展，其指令格式及功能描述见表 3-7 所示。

表 3-7　逻辑运算指令

示　　例		功 能 描 述
AND　Rd，Rn AND.W Rd，Rn，＃imm12 AND.W Rd，Rm，Rn	；Rd &= Rn ；Rd ＝ Rn & imm12 ；Rd ＝ Rm & Rn	按位与
ORR Rd，Rn ORR.W Rd，Rn，＃imm12 ORR.W Rd，Rm，Rn	；Rd |= Rn ；Rd ＝ Rn | imm12 ；Rd ＝ Rm | Rn	按位或
BIC Rd，Rn BIC.W Rd，Rn，＃imm12 BIC.W Rd，Rm，Rn	；Rd &= ～Rn ；Rd ＝ Rn & ～imm12 ；Rd ＝ Rm & ～Rn	位清零 Rn 与 Operand2 的反码按位逻辑与
ORN.W Rd，Rn，＃imm12 ORN.W Rd，Rm，Rn	；Rd ＝ Rn | ～imm12 ；Rd ＝ Rm | ～Rn	按位或反码
EOR Rd，Rn EOR.W Rd，Rn，＃imm12 EOR.W Rd，Rm，Rn	；Rd ^= Rn ；Rd ＝ Rn ^ imm12 ；Rd ＝ Rm ^ Rn	按位异或

- AND 实现第 1 个操作数 Rm 与第 2 操作数 Operand2 的逻辑与操作。
- ORR 实现第 1 个操作数 Rm 与第 2 操作数 Operand2 的逻辑或操作。
- EOR 实现第 1 个操作数 Rm 与第 2 操作数 Operand2 的逻辑异或操作。
- BIC 实现第 1 个操作数 Rm 与第 2 操作数 Operand2 的反码的逻辑与操作，一般可用来清除第 1 个操作数 Rm 的相应位。
- ORN 实现第 1 个操作数 Rm 与第 2 操作数 Operand2 的反码的逻辑或操作。

使用举例：

① `AND R1, R1, #0x00FF`

该指令实现 R1 寄存器的内容与立即数＃0x00FF 进行逻辑与操作，并将结果送至 R1 寄存器，即实现 R1 寄存器的高 16 位清零。

② `BIC R2, R2, #0x000B`

该指令实现 R2 寄存器的内容与立即数＃0x000B 的反码进行逻辑与操作，并将结果送至 R2 寄存器，即实现 R2 寄存器的第 0、1 和 3 位的清零。

逻辑运算中逻辑非的操作由送数指令 MVN 实现。

3.1.8 移位和循环指令

移位运算包括逻辑移位、算术移位以及循环移位等特殊形式，具体指令格式和功能见表 3-8。

表 3-8 移位和循环指令

示 例		功 能 描 述
LSL Rd, Rn, #imm5 LSL Rd, Rn LSL.W Rd, Rm, Rn	; Rd = Rn<<imm5 ; Rd <<= Rn ; Rd = Rm<<Rn	逻辑左移
LSR Rd, Rn, #imm5 LSR Rd, Rn LSR.W Rd, Rm, Rn	; Rd = Rn>>imm5 ; Rd >>= Rn ; Rd = Rm>>Rn	逻辑右移
ASR Rd, Rn, #imm5 ASR Rd, Rn ASR.W Rd, Rm, Rn	; Rd = Rn·>>imm5 ; Rd ·>> = Rn ; Rd = Rm·>> Rn	算术右移
ROR Rd, Rn ROR.W Rd, Rm, Rn	; Rd >> = Rn ; Rd = Rm >> Rn	循环右移
RRX.W Rd, Rn	; Rd=(Rn>>1)+(C<<31)	带进位的右移一位

- ASR 算术右移指令实现对 Rm 寄存器中的数进行右移 Rn 或 imm5 位操作，左端使用第 31 位值来补充，如图 3-8(a)所示。
- LSL 逻辑左移指令实现对 Rm 寄存器中的数进行左移 Rn 或 imm5 位操作，低位使用 0 填充，如图 3-8(b)所示。
- LSR 逻辑右移指令实现对 Rm 寄存器中的数进行右移 Rn 或 imm5 位操作，左端使用 0 填充，如图 3-8(c)所示。
- ROR 循环右移指令实现对 Rm 寄存器中的数进行右移 Rn 或 imm5 位操作，左端使用右端移出的位来进行填充，如图 3-8(d)所示。
- RRX 带进位的循环右移，左端使用进位标志 C 来进行填充，如图 3-8(e)所示。

使用举例：

① `ASR R1, R2, #9`

该指令实现将 R2 寄存中的内容算术右移 9 位(左端使用第 31 位值来填充)，然后将结果填入 R1 寄存器。

② `LSLS R3, R4, #3`

该指令附加标志位 S，计算结果可能影响标志位。该指令实现将 R4 寄存器逻辑左移 3 位(低位使用 0 值来填充)，然后将结果填入 R3 寄存器。

③ `ROR R5, R6, #6`

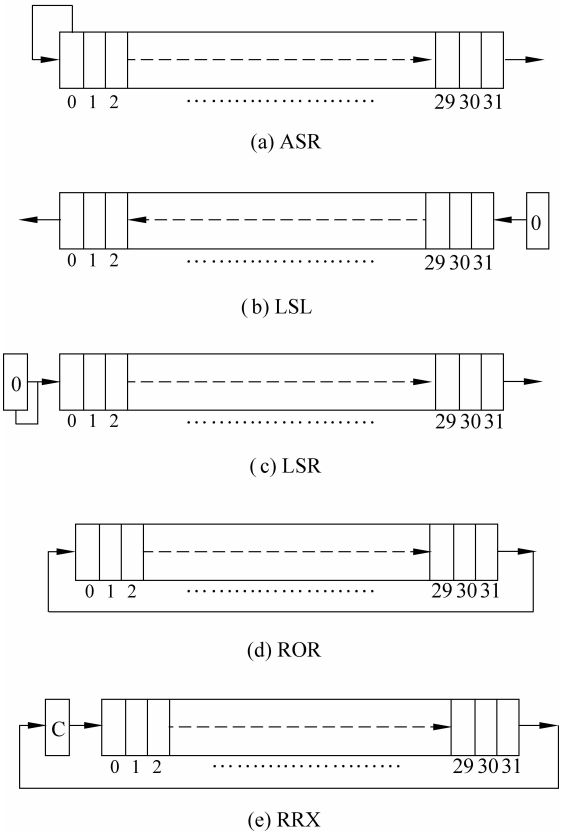

图 3-8　移位操作

该指令实现将 R6 寄存器循环右移 6 位(左端使用右端移出的位进行填充),然后将结果填入 R5 寄存器。

3.1.9　比较指令

CMP 和 CMN 是比较操作指令,其具体指令格式和功能见表 3-9。

表 3-9　比较指令

示　例	功 能 描 述
CMN <Rn>, <Rm>	将 Rm 取二进制补码后再与 Rn 比较
CMP <Rn>, #<immed_8>	Rn 与 8 位立即数比较,并根据结果更新标志位的值
CMP <Rn>, <Rm>	Rn 与 Rm 比较,并根据结果更新标志位的值
CMN. W <Rn>, #< immed_12>	Rn 与 12 位立即数取补后的值比较
CMN. W <Rn>, <Rm>{, <shift>}	Rn 与移位后的 Rm 取补后的值比较
CMP. W <Rn>, #< immed_12>	Rn 与 12 位立即数比较
CMP. W <Rn>, <Rm>{, <shift>}	Rn 与按需移位后的 Rm 比较,Rm 的值不变

CMP 指令实现将 Rn 寄存器的内容减去第 2 个操作数 Operand2，计算结果影响标志位。该指令类似于 SUBS 指令，所不同的是 CMP 指令丢弃减法计算结果，而 SUBS 指令需要保存减法计算结果。

CMN 指令实现将第 2 个操作数加到 Rn 寄存器中，计算结果影响标志位。该指令类似于 ADDS 指令，所不同的是 CMN 指令丢弃加法计算结果，而 ADDS 指令需要保存加法计算结果。

比较指令根据计算结果影响标志位 N、Z、C 和 V。

使用举例：

① `CMP R1, #6400`

该指令将 R1 寄存器的内容减去立即数 #6400，计算结果影响标志位。

② `CMN R2, R1`

指令将 R1 寄存器的内容加到 R2 寄存器，计算结果影响标志位。

3.1.10 分支控制指令

分支控制指令(Branch and Control Instructions)包括直接跳转指令 B 和 BL、间接跳转指令 BX 和 BLX，具体指令格式和功能见表 3-10.

表 3-10 跳转指令

示 例	功 能 描 述
B<cond> <target address>	按<contd>条件决定是否分支
B<tartet address>	无条件分支
BL <Rm>	带链接分支
B{cond}. W <label>	条件分支
BL. W <label>	带链接的分支
BL. W<c> <label>	带链接的分支(立即数)
B. W <label>	无条件分支

- B 是直接跳转指令，label 可以是 24 位的有符号数，跳转范围是－16～16MB。
- BL 除了可以直接跳转到 label 处外，在跳转前会将 PC 内容保存到 LR 寄存器。
- BX 是间接跳转指令，跳转到的目标地址存放在 Rm 寄存器。
- BLX 也是一个间接跳转指令，同样在跳转前会将 PC 内容保存到 LR 寄存器。

使用举例：

```
① B loop          ; 直接跳转到 loop 处;
② BLE ng          ; 当 Z=1 或 N!=V,即有符号数小于或等于时,直接跳转到 ng 处;
③ BEQ target1     ; 当 Z =1,即相等时,直接跳转到 target1 处
④ BX LR           ; 返回函数调用处
```

3.1.11　其他指令

主要包括位操作指令、符号扩展指令、字节交换指令等，指令的具体格式和功能见表 3-11。

表 3-11　其他指令

示　　例	功 能 描 述
BFC. W Rd，Rn，＃＜width＞	位区清零
BFI. W Rd，Rn，＃＜lsb＞，＃＜width＞	将一个寄存器的位区插入另一个寄存器中
SBFX. W Rd，Rn，＃＜lsb＞，＃＜width＞	复制位段，并带符号扩展到 32 位
SBFX. W Rd，Rn，＃＜lsb＞，＃＜width＞	复制位段，并无符号扩展到 32 位
REV. W Rd，Rn	在字中反转字节序
REV16. W Rd，Rn	在高低半字中反转字节序
REVSH. W	在低半字中反转字节序，并做带符号扩展
SXTB Rd，Rm{，＜rotation＞}	Rd = Rm 把带符号字节整数扩展到 32 位
SXTH Rd，Rm{，＜rotation＞}	Rd = Rm 把带符号半字整数扩展到 32 位

1）BFC 和 BFI

- BFC 指令实现 Rd 寄存器中从 1sb 开始的 width 位数的位清零。
- BFI 指令实现 Rn 寄存器中从 0 开始的 width 位拷贝到 Rd 寄存器中从 1sb 开始的 width 位。

使用举例：

① `BFC R1, #8, #13`

该指令实现对 R1 寄存器中从第 8 位开始的 13 位（即到第 20 位）的数据进行清零。

② `BFI R2, R3, #7, #11`

该指令实现将 R3 寄存器中从第 0 位到第 10 位的数据拷贝到 R2 寄存器的第 7 位到第 17 位。

2）SBFX 和 UBFX

- SBFX 指令实现将 Rn 寄存器中从第 1sb 位开始的 width 位抽取出来，然后进行有符号位扩展到 32 位并将结果存入 Rd 寄存器。
- UBFX 指令实现将 Rn 寄存器中从第 1sb 位开始的 width 位抽取出来，然后进行无符号位扩展到 32 位并将结果存入 Rd 寄存器。

使用举例：

① `SBFX R1, R2, #10, #4`

该指令实现将 R2 寄存器中的第 10 位到第 13 位抽取出来并进行有符号扩展到 32 位，

然后将结果存入 R1 寄存器。

② UBFX R3, R4, #9, #10

该指令实现将 R4 寄存器中的第 9 位到第 18 位抽取出来并进行无符号扩展到 32 位，然后将结果存入 R3 寄存器。

3）SXTB、SXTH、UXTB 和 UXTH

- SXTB 指令实现将 Rm 寄存器的低 8 位，或 Rm 寄存器经过循环右移 rotation 位后的低 8 位，有符号扩展到 32 位，然后存入 Rd 寄存器。
- UXTH 指令实现将 Rm 寄存器的低 16 位，或 Rm 寄存器经过循环右移 rotation 位后的低 16 位，无符号扩展到 32 位，然后存入 Rd 寄存器。

使用举例：

① SXTH R1, R2, ROR #16

该指令首先将 R2 寄存器的内容进行循环右移 16 位，然后取出低 16 位的半字并进行有符号扩展到 32 位，最后将结果存入 R1 寄存器。

② UXTB R3, R10

该指令首先取出 R10 寄存器的低 8 位，然后进行无符号扩展到 32 位，最后将结果存入 R3 寄存器。

4）REV、REV16、REVSH

- REV 实现一个字 Rn 的四个字节大小端转换，复制到 Rd 中。
- REV16 实现 Rn 的两个半字内部的大小端转换，复制到 Rd 中。
- REVSH 将 Rn 低半字内的字节反转，再把反转后的值带符号位扩展到 32 位后，复制到 Rd 中。

字节交换指令的原理见图 3-9。

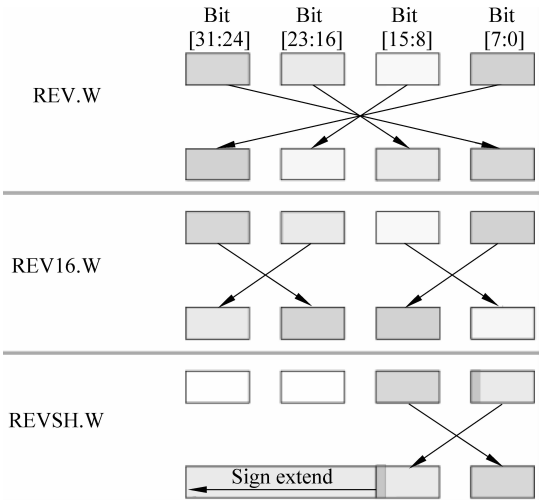

图 3-9　字节交换指令交换顺序