



嵌入式系统原理及应用

第3章 ARM的指令系统

张帆


中南大学自动化学院



第3章 ARM的指令系统

- ❖ 3.1 ARM处理器的寻址方式
- ❖ 3.2 ARM处理器的指令集
- ❖ 3.3 小结
- ❖ 3.4 思考与练习

• ARM指令集与Thumb指令集的关系



ARM指令集支持ARM核所有的特性，具有高效、快速的特点

Thumb指令集具有灵活、小巧的特点

1.ARM程序的文件类型:

- ◆C程序: ARM开发中大部分程序使用C语言编写, 文件类型为“*.C”;
- ◆汇编程序: 涉及到硬件底层操作的代码有时必须使用汇编语言编写, 文件类型为“*.S”。

2.为什么学习ARM指令系统:

- ◆操作系统移植
- ◆编写启动代码
- ◆方便程序调试

• 寻址方式分类

寻址方式是根据指令中给出的地址码字段来实现寻找真实操作数地址的方式。ARM 处理器具有 8 种基本寻址方式。

1. 寄存器寻址;

2. 立即寻址;

3. 寄存器移位寻址;

4. 寄存器间接寻址;

5. 基址寻址;

6. 多寄存器寻址;

7. 堆栈寻址;

8. 相对寻址。

- 寻址方式分类——寄存器寻址

操作数的值在寄存器中，指令中的地址码字段指出的是寄存器编号，指令执行时直接取出寄存器值来操作。寄存器寻址指令举例如下：

MOV R1,R2 ;将R2的值存入R1

SUB R0,R1,R2 ;将R1的值减去R2的值，结果保存到R0

- 寻址方式分类——立即寻址

立即寻址指令中的操作码字段后面的地址码部分即是操作数本身，也就是说，数据就包含在指令当中，取出指令也就取出了可以立即使用的操作数（这样的数称为立即数）。立即寻址指令举例如下：

SUBS R0,R0,#1 ;R0减1，结果放入R0，并且影响标志位

MOV R0,#0xFF000 ;将立即数0xFF000装入R0寄存器

• 寻址方式分类——寄存器移位寻址

寄存器移位寻址是ARM指令集特有的寻址方式。当第2个操作数是寄存器移位方式时，第2个寄存器操作数在与第1个操作数结合之前，选择进行移位操作。寄存器移位寻址指令举例如下：

MOV	R0,R2,LSL #3	;R2的值左移3位，结果放入R0， ;即是 $R0=R2 \times 8$
ANDS	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

• 寻址方式分类——寄存器间接寻址

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的编号，所需的操作数保存在寄存器指定地址的存储单元中，即寄存器为操作数的地址指针。寄存器间接寻址指令举例如下：

LDR R1, [R2] ;将R2指向的存储单元的数据读出

 ;保存在R1中

SWP R1, R1, [R2] ;将寄存器R1的值和R2指定的存储

 ;单元的内容交换

• 寻址方式分类——基址寻址

基址寻址就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。基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、功能部件寄存器访问等。基址寻址指令举例如下：

LDR R2, [R3, #0x0C] ; 读取R3+0x0C地址上的存储单元
 ; 的内容，放入R2

STR R1, [R0, #-4] ! ; 先R0=R0-4，然后把R1的值寄存
 ; 到保存到R0指定的存储单元

• 寻址方式分类——多寄存器寻址

多寄存器寻址一次可传送几个寄存器值，允许一条指令传送16个寄存器的任何子集或所有寄存器。多寄存器寻址指令举例如下：

LDMIA R1!, {R2-R7, R12} ; 将R1指向的单元中的数据读出到
; R2~R7、R12中 (R1自动加1)

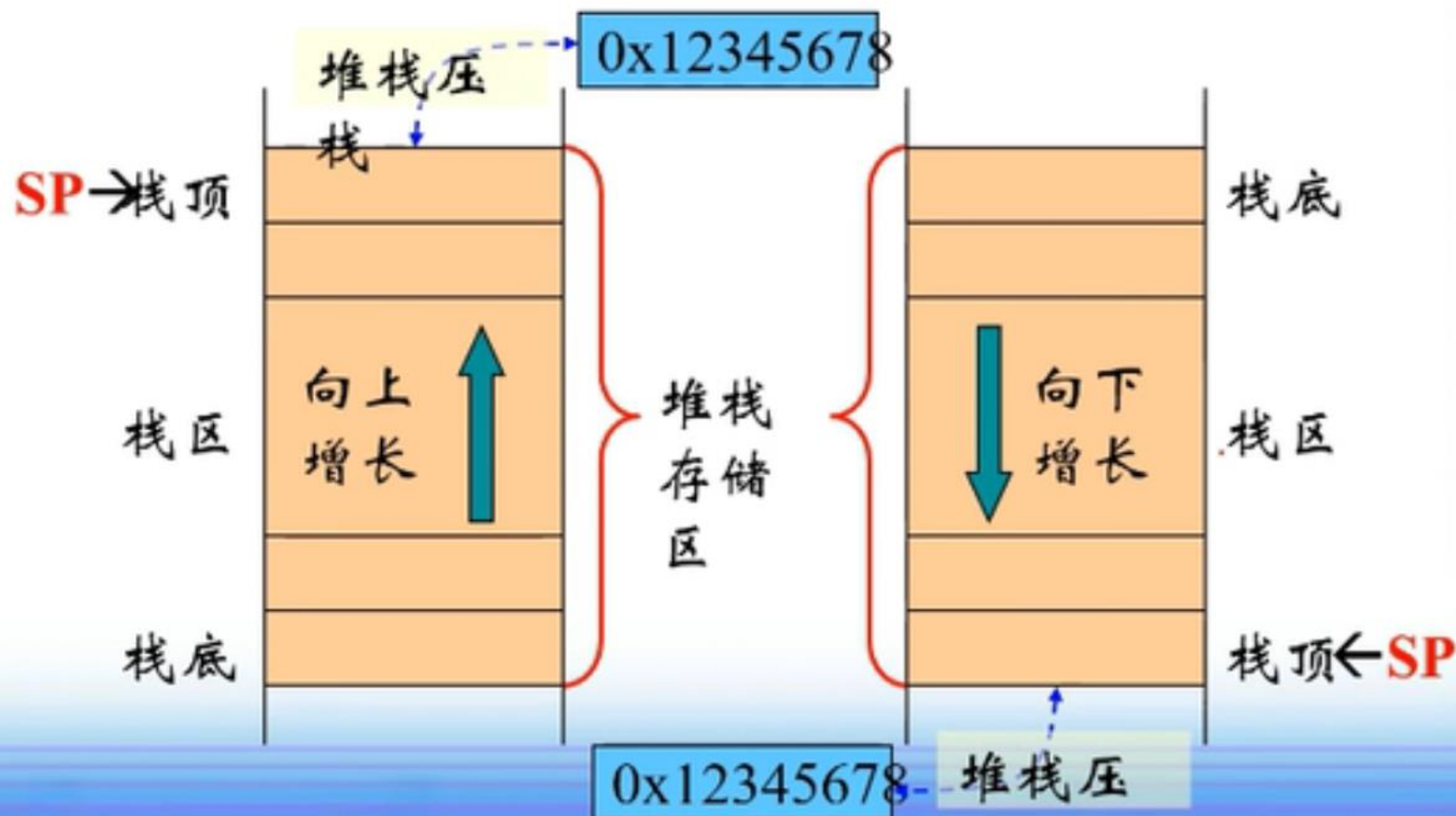
STMIA R0!, {R2-R7, R12} ; 将寄存器R2~R7、R12的值保
; 存到R0指向的存储单元中
; (R0自动加1)

• 寻址方式分类——堆栈寻址

堆栈是一个按特定顺序进行存取的存储区，操作顺序为“后进先出”。堆栈寻址是隐含的，它使用一个专门的寄存器(堆栈指针)指向一块存储区域(堆栈)，指针所指向的存储单元即是堆栈的栈顶。存储器堆栈可分为两种：

- 向上生长：向高地址方向生长，称为递增堆栈
- 向下生长：向低地址方向生长，称为递减堆栈

• 寻址方式分类——堆栈寻址



• 寻址方式分类——堆栈寻址

堆栈指针指向最后压入的堆栈的有效数据项，称为**满堆栈**；
堆栈指针指向下一个待压入数据的空位置，称为**空堆栈**。



• 寻址方式分类——堆栈寻址

所以可以组合出四种类型的堆栈方式：

■ **满递增**：堆栈向上增长，堆栈指针指向内含有效数据项的最高地址。指令如LDMFA、STMFA等；

■ **空递增**：堆栈向上增长，堆栈指针指向堆栈上的第一个空位置。指令如LDMEA、STMEA等；

■ **满递减**：堆栈向下增长，堆栈指针指向内含有效数据项的最低地址。指令如LDMFD、STMFD等；

■ **空递减**：堆栈向下增长，堆栈指针向堆栈下的第一个空位置。指令如LDMED、STMED等。

• 寻址方式分类——相对寻址

相对寻址是基址寻址的一种变通。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。相对寻址指令举例如下：

	BL	SUBR1	;调用到SUBR1子程序
	BEQ	LOOP	;条件跳转到LOOP标号处
	...		
LOOP	MOV	R6, #1	
	...		
SUBR1	...		

3.1 ARM处理器的寻址方式

❖ ARM指令集可以分为：

- 🔗 跳转指令、
- 🔗 数据处理指令、
- 🔗 程序状态寄存器传输指令、
- 🔗 Load/Store指令、
- 🔗 协处理器指令
- 🔗 异常中断产生指令。

❖ 根据使用的指令类型不同，指令的寻址方式分为数据处理指令寻址方式和内存访问指令寻址方式。

3.1.1 数据处理指令寻址方式

✎ 数据处理指令的基本语法格式如下：

<opcode> {<cond>} {S} <Rd>, <Rn>, <shifter_operand>

其中，<shifter_operand>有11种形式

↵	语 法↵	寻 址 方 式↵
1↵	# <immediate>↵	立即数寻址↵
2↵	<Rm>↵	寄存器寻址↵
3↵	<Rm>, LSL #<shift_imm>↵	立即数逻辑左移↵
4↵	<Rm>, LSL <Rs>↵	寄存器逻辑左移↵
5↵	<Rm>, LSR #<shift_imm>↵	立即数逻辑右移↵
6↵	<Rm>, LSR <Rs>↵	寄存器逻辑右移↵
7↵	<Rm>, ASR #<shift_imm>↵	立即数算术右移↵
8↵	<Rm>, ASR <Rs>↵	寄存器算术右移↵
9↵	<Rm>, ROR #<shift_imm>↵	立即数循环右移↵
10↵	<Rm>, ROR <Rs>↵	寄存器循环右移↵
11↵	<Rm>, RRX↵	寄存器扩展循环右移↵

3.1.1 数据处理指令寻址方式

🌿 数据处理指令寻址方式分类：

- ❖ (1) 立即数寻址方式；
- ❖ (2) 寄存器寻址方式；
- ❖ (3) 寄存器移位寻址方式。

(1) 立即数寻址

指令中的立即数是由一个8bit常数移动4bit偶数位得到的

🔗 下面列举了一些有效的立即数：

❖ 0xFF、0x104、0xFF0、0xFF00、0xFF000、0xFF000000、
0xF000000F

🔗 下面是一些无效的立即数：

❖ 0x101、0x102、0xFF1、0xFF04、0xFF003、0xFFFFFFFFF、
0xF000001F

🔗 应用立即数的指令：

- ❖ MOV R0, #0 ;送0到R0
- ❖ ADD R3, R3, #1 ;R3的值加1
- ❖ CMP R7, #1000 ;R7的值和1000比较
- ❖ BIC R9, R8, #0xFF00 ;将R8中8~15位清零，结果保存在R9中

(2) 寄存器寻址

寄存器的值可以被直接用于数据操作指令，这种寻址方式是各类处理器经常采用的一种方式，也是一种执行效率较高的寻址方式，如：

- ❖ MOV R2, R0 ;R0的值送R2
- ❖ ADD R4, R3, R2 ;R2加R3，结果送R4
- ❖ CMP R7, R8 ;比较R7和R8的值

(3) 寄存器移位寻址

寄存器的值在被送到ALU之前，可以事先经过**桶形移位寄存器**的处理。预处理和移位发生在同一周期内，所以有效地使用移位寄存器，可以增加代码的执行效率。

- ❖ ADD R2, R0, R1, LSR #5
- ❖ MOV R1, R0, LSL #2
- ❖ RSB R9, R5, R5, LSL #1
- ❖ SUB R1, R2, R0, LSR #4
- ❖ MOV R2, R4, ROR R0

3.1.2 内存访问指令寻址方式

内存访问指令的寻址方式可以分为以下几种。

- (1) 寄存器间接寻址方式。
- (2) 基址变址寻址方式；
- (3) 多寄存器寻址/快拷贝寻址方式
- (4) 相对寻址方式
- (5) 堆栈操作寻址方式。

(1) 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寄存器指定地址的存储单元中，即寄存器为操作数的地址指针。。

LDR|STR {<cond>} {B} {T} <Rd>, [Rm]

❖ LDR R1, [R2] ;

❖ STR R1, [R2] ;

(2) 寄存器基址变址寻址

将基址寄存器的值与偏移量相加，形成操作数的有效地址，基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、功能寄存器访问等。

LDR|STR {<cond>} {B} {T} <Rd>, <addressing_mode>

其中<addressing_mode>共有9种寻址方式

↺	格 式↺	模 式↺
1↺	[Rn, #±<offset_12>]↺	立即数偏移寻址 (Immediate offset) ↺
2↺	[Rn, ±Rm]↺	寄存器偏移寻址 (Register offset) ↺
3↺	[Rn, Rm, <shift>#< offset_12>]↺	带移位的寄存器偏移寻址 (Scaled register offset) ↺
4↺	[Rn, #±< offset_12>]!↺	立即数前索引寻址 (Immediate pre-indexed) ↺
5↺	[Rn, ±Rm]!↺	寄存器前索引寻址 (Register post-indexed) ↺
6↺	[Rn, Rm, <shift>#< offset_12>]!↺	带移位的寄存器前索引寻址 (Scaled register pre-indexed) ↺
7↺	[Rn], #±< offset_12>↺	立即数后索引寻址 (Immediate post-indexed) ↺
8↺	[Rn], ±<Rm>↺	寄存器后索引寻址 (Register post-indexed) ↺
9↺	[Rn], ±<Rm>, <shift>#< offset_12>↺	带移位的寄存器后索引寻址 (Scaled register post-indexed) ↺

❖ “!” 表示完成数据传输后要更新基址寄存器。

(2) 寄存器基址变址寻址

LDR R1, [R0, #0x0f]; 将R0中的数值加0x0f作为地址，取出此地址的数值保存在R1

STR R1, [r0, #-2]; 将R0中的数值减2作为地址，将R1中的内容保存在此地址位置

STR R1, [R0, +R2]; 将R0中的数值加上R2的值作为地址，将R1中的内容保存在此地址位置

LDR R3, [R1, #4]!;..... 更新R1, 即R1=R1+4

“!” 表示完成数据传输后要更新基址寄存器。

(3) 多寄存器寻址/块拷贝寻址

批量Load/Store指令将一片连续内存单元的数据加载到通用寄存器组中或将一组通用寄存器的数据存储在内存单元中。

批量Load/Store指令的寻址模式产生一个内存单元的地址范围，指令寄存器和内存单元的对应关系满足这样的规则，即编号低的寄存器对应于内存中低地址单元，编号高的寄存器对应于内存中的高地址单元。

LDM|STM {<cond>} <addressing_mode> <Rn>{!}, <registers><^>

↕	格 式↕	模 式↕
1↕	IA (Increment After) ↕	后递增方式↕
2↕	IB (Increment Before) ↕	先递增方式↕
3↕	DA (Decrement After) ↕	后递减方式↕
4↕	DB (Decrement Before) ↕	先递减方式↕

(3) 多寄存器寻址/块拷贝寻址

❖ STMIA R0! , {R1~R7} ;

将R1~R7的数据保存到R0所指向的存储器中，R0的值之后增加，增长方向为向上增长。类似于C语言的i++

❖ STMIB R0! , {R1~R7} ;

R0的值先增加，后将R1~R7的数据保存到R0所指向的存储器中，增长方向为向上增长。类似于C语言的++i

❖ STMDA R0! , {R1~R7} ;

将R1~R7的数据保存到R0所指向的存储器中，R0的值之后减少，增长方向为向下增长。类似于C语言的i--

❖ STMDB R0! , {R1~R7} ;

R0的值先减少，后将R1~R7的数据保存到R0所指向的存储器中，增长方向为向下增长。类似于C语言的--i

(4) 相对寻址

相对寻址是基址寻址的一种变通，由程序计数器PC提供基址地址，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

指令：B 和 BL

❖ BL FUN1 ; 调用到FUN1子程序

❖ B LOOP ; 条件跳转到LOOP标号处

(5) 堆栈寻址方式

类似于多寄存器Load/Store指令寻址方式

根据不同的寻址方式，将堆栈分为以下4种

- 🌀Full栈：堆栈指针指向栈顶元素 (last used location)
- 🌀Empty栈：堆栈指针指向第一个可用元素 (the first unused location)
- 🌀递减栈：堆栈向内存地址减小的方向生长
- 🌀递增栈：堆栈向内存地址增加的方向生长

根据堆栈的不同种类，将其寻址方式分为以下4种

- 🌀满递减FD (Full Descending)
- 🌀空递减ED (Empty Descending)
- 🌀满递增FA (Full Ascending)
- 🌀空递增EA (Empty Ascending)

堆栈寻址方式和批量Load/Store指令寻址方式对应关系

批量数据寻址方式↵	堆栈寻址方式↵	L 位↵	P 位↵	U 位↵
LDMDA↵	LDMFA↵	1↵	0↵	0↵
LDMIA↵	LDMFD↵	1↵	0↵	1↵
LDMDB↵	LDMEA↵	1↵	1↵	0↵
LDMIB↵	LDMED↵	1↵	1↵	1↵
STMDA↵	STMED↵	0↵	0↵	0↵
STMIA↵	STMEA↵	0↵	0↵	1↵
STMDB↵	STMFD↵	0↵	1↵	0↵
STMIB↵	STMFA↵	0↵	1↵	1↵

3.2 ARM处理器的指令集

❖ 3.2.1 数据操作指令

数据操作指令是指对存放在寄存器中的数据进行操作的指令。
包括

数据传送指令

算术指令

逻辑指令

比较与测试指令

乘法指令。

如果在数据处理指令使用S后缀，指令的执行结果将会影响CPSR中的标志位。

数据处理指令列表

助记符	操 作	行 为
MOV	数据传送	
MVN	数据取反传送	
AND	逻辑与	$Rd: =Rn \text{ AND } op2$
EOR	逻辑异或	$Rd: =Rn \text{ EOR } op2$
SUB	减	$Rd: =Rn - op2$
RSB	翻转减	$Rd: =op2 - Rn$
ADD	加	$Rd: =Rn + op2$
ADC	带进位的加	$Rd: =Rn + op2 + C$
SBC	带进位的减	$Rd: =Rn - op2 + C - 1$
RSC	带进位的翻转减	$Rd: =op2 - Rn + C - 1$
TST	测试	$Rn \text{ AND } op2$ 并更新标志位
TEQ	测试相等	$Rn \text{ EOR } op2$ 并更新标志位
CMP	比较	$Rn - op2$ 并更新标志位
CMN	负数比较	$Rn + op2$ 并更新标志位
ORR	逻辑或	$Rd: =Rn \text{ OR } op2$
BIC	位清 0	$Rd: =Rn \text{ AND NOT } (op2)$

1. MOV指令

MOV指令多用于设置初始值或者在寄存器间传送数据

(1) 指令的语法格式:

MOV {<cond>} {S} <Rd>, <shifter_operand>

(2) 指令举例

MOV R0, R0 ; R0 = R0... NOP 指令

MOV R0, R0, LSL#3 ; R0 = R0 * 8

MOV PC, R14 ; 退出到调用者, 用于普通函数返回,
PC即是R15

MOVS PC, R14 ; 退出到调用者并恢复标志位, 用于异常函数返回

1. MOV指令

(3) 指令的使用

- ① 将数据从一个寄存器传送到另一个寄存器
- ② 将一个常数值传送到寄存器中。
- ③ 实现无算术和逻辑运算的单纯移位操作，操作数乘以 2^n 可以用左移n位来实现
- ④ 当PC（R15）用作目的寄存器时，可以实现程序跳转。如“MOV PC, LR”，所以这种跳转可以实现子程序调用及从子程序返回，代替指令“B, BL”
- ⑤ 当PC作为目标寄存器且指令中S位被设置时，指令在执行跳转操作的同时，将当前处理器模式的SPSR寄存器的内容复制到CPSR中。这种指令“MOVS PC LR”可以实现从某些异常中断中返回。

2. MVN指令

❖ MVN指令多用于向寄存器传送一个负数或生成位掩码

❖ (1) 指令的语法格式:

❖ MVN {<cond>} {S} <Rd>, <shifter_operand>

❖ (2) 指令举例

MVN R0, #4 ; R0 = -5

MVN R0, #0 ; R0 = -1

这是逻辑非操作而不是算术操作，这个取反的值加1才是它的取负的值。

2. MVN指令

❖ 3) 指令的使用

- 🌀① 向寄存器中传送一个负数。
- 🌀② 生成位掩码（**Bit Mask**）。
- 🌀③ 求一个数的反码。

位掩码（BitMask），是”位（Bit）“和”掩码（Mask）“的组词。”位“指代着二进制数据当中的二进制位，而”掩码“指的是一串用于与目标数据进行按位操作的二进制数字。组合起来，就是”用一串二进制数字（掩码）去操作另一串二进制数字“的意思。

3. AND指令

AND指令将shifter_operand表示的数值与寄存器Rn的值按位（bitwise）做逻辑与操作，并将结果保存到目标寄存器Rd中，同时根据操作的结果更新CPSR寄存器。

（1）指令的语法格式：

AND {<cond>} {S} <Rd>, <Rn>, <shifter_operand>

3. AND指令

(2) 指令举例

- ① 保留R0中的0位和1位，丢弃其余的位。

AND R0, R0, #3

- ② $R2 = R1 \& R3$ 。

AND R2, R1, R3

- ③ $R0 = R0 \& 0x01$ ，取出最低位数据。

ANDS R0, R0, #0x01

4. EOR指令

EOR (Exclusive OR) 指令将寄存器Rn中的值和shifter_operand的值执行按位“异或”操作，并将执行结果存储到目的寄存器Rd中，同时根据指令的执行结果更新CPSR中相应的条件标志位。

(1) 指令的语法格式：

EOR {<cond>} {S} <Rd>, <Rn>, <shifter_operand>

(2) 指令举例

- ① 反转R0中的位0和1: EOR R0, R0, #3
- ② 将R1的低4位取反: EOR R1, R1, #0x0F
- ③ R2 = R1 \wedge R0: EOR R2, R1, R0

x	y	$x \oplus y$
0	0	0
0	1	1
1	0	1
1	1	0

5. SUB指令

SUB (Subtract) 指令从寄存器Rn中减去shifter_operand表示的数值，并将结果保存到目标寄存器Rd中，并根据指令的执行结果设置CPSR中相应的标志位。

(1) 指令的语法格式

 SUB {<cond>} {S} <Rd>, <Rn>, <shifter_operand>

(2) SUB指令举例

- ① SUB R0, R1, R2; $R0 = R1 - R2。$
- ② SUB R0, R1, #256; $R0 = R1 - 256。$
- ③ SUB R0, R2, R3, LSL#1; $R0 = R2 - (R3 \ll 1)。$

6. RSB指令

RSB (Reverse Subtract) 指令从寄存器shifter_operand中减去Rn表示的数值，并将结果保存到目标寄存器Rd中，并根据指令的执行结果设置CPSR中相应的标志位。

(1) 指令的语法格式

RSB{<cond>} {S} <Rd>, <Rn>, <shifter operand>

（2）RSB指令举例

下面指令序列可以求一个64位数值的负数。64位数放在寄存器R0与R1中，其负数放在R2和R3中。其中R0与R2中放低32位值。

RSBS	R2, R0, #0	(逆向减法指令)
------	------------	----------

RSC	R3, R1, #0	(反向带进位减)
-----	------------	----------

RSC{条件}{S} 目的寄存器, 操作数1, 操作数2

RSC指令用于把 操作数2减去操作数1，再减去CPSR中的C条件标志位的反码，并将结果存放到目的寄存器中。

7. ADD指令

ADD指令将寄存器shifter_operand的值加上Rn表示的数值，并将结果保存到目标寄存器Rd中，并根据指令的执行结果设置CPSR中相应标志位。

(1) 指令的语法格式

$$\text{ADD} \{ \langle \text{cond} \rangle \} \{ S \} \quad \langle \text{Rd} \rangle, \langle \text{Rn} \rangle, \langle \text{shifter_operand} \rangle$$

(2) ADD指令举例

ADD	R0, R1, R2	; R0 = R1 + R2
ADD	R0, R1, #256	; R0 = R1 + 256
ADD	R0, R2, R3, LSL#1	; R0 = R2 + (R3 << 1)

8. ADC指令

ADC指令将寄存器shifter_operand的值加上Rn表示的数值，再加上CPSR中的C条件标志位的值，将结果保存到目标寄存器Rd中，并根据指令的执行结果设置CPSR中相应的标志位。

(1) 指令的语法格式

$$\text{ADC } \{ \langle \text{cond} \rangle \} \{ S \} \quad \langle \text{Rd} \rangle, \langle \text{Rn} \rangle, \langle \text{shifter_operand} \rangle$$

8. ADC指令

ADC指令举例：两个128位的数相加

128位结果：寄存器R0、R1、R2和R3

第一个128位数：寄存器R4、R5、R6和R7

第二个128位数：寄存器R8、R9、R10和R11。

ADDS	R0, R4, R8	;加低端的字
ADCS	R1, R5, R9	;加下一个字, 带进位
ADCS	R2, R6, R10	;加第三个字, 带进位
ADCS	R3, R7, R11	;加高端的字, 带进位

9. SBC指令

SBC (Subtract with Carry) 指令用于执行操作数大于32位时的减法操作。该指令从寄存器Rn中减去shifter_operand表示的数值，再减去寄存器CPSR中C条件标志位的反码 [NOT (Carry flag)]，并将结果保存到目标寄存器Rd中，并根据指令的执行结果设置CPSR中相应的标志位。

(1) 指令的语法格式

SBC {<cond>} {S} <Rd>, <Rn>, <shifter_operand>

(2) SBC指令举例：下面的程序使用SBC实现64位减法，
(R1, R0) - (R3, R2)，结果存放到 (R1, R0)

SUBS R0, R0, R2

SBCS R1, R1, R3

10. RSC指令

RSC (Reverse Subtract with Carry) 指从寄存器shifter_operand中减去Rn表示的数值，再减去寄存器CPSR中C条件标志位的反码 [NOT (Carry Flag)]，并将结果保存到目标寄存器Rd中，并根据指令的执行结果设置CPSR中相应的标志位。

(1) 指令的语法格式

RSC {<cond>} {S} <Rd>, <Rn>, <shifter_operand>

(2) RSC指令举例

下面程序使用RSC指令实现求64位数值 (R1, R0) 的负数。

```
RSBS      R2, R0, #0
```

```
RSC       R3, R1, #0
```

11. TST测试指令

TST (Test) 测试指令用于将一个寄存器的值和一个算术值进行比较。条件标志位根据两个操作数做“逻辑与”后的结果设置。

(1) 指令的语法格式

TST {<cond>} <Rn>, <shifter_operand>

(2) TST指令举例

TST指令类似于CMP指令，不产生放置到目的寄存器中的结果。而是在给出的两个操作数上进行操作并把结果反映到状态标志上。使用TST指令来检查是否设置了特定的位。操作数1是要测试的数据字而操作数2是一个位掩码。经过测试后，如果匹配则设置Zero标志，否则清除它。与CMP指令一样，该指令不需要指定S后缀。

下面的指令测试在R0中是否设置了位0

```
TST    R0, #%1
```

12. TEQ指令

TEQ (Test Equivalence) 指令用于将一个寄存器的值和一个算术值做比较。条件标志位根据两个操作数做“逻辑异或”后的结果设置。以便后面的指令根据相应的条件标志来判断是否执行。

(1) 指令的语法格式

TEQ{<cond>} <Rn>,<shifter_operand>

(2) TEQ指令举例

下面的指令是比较R0和R1是否相等，该指令不影响CPSR中的V位和C位。

TEQ R0,R1

使用TEQ进行相等测试，常与EQ和NE条件码配合使用，当两个数据相等时，条件码EQ有效；否则条件码NE有效。

例：

```
TEQ      R0, R1
ADDEQ    R0, R0, #1
```

13. CMP指令

CMP (Compare) 指令使用寄存器Rn的值减去operand2的值，根据操作的结果更新CPSR中相应的条件标志位，以便后面的指令根据相应的条件标志来判断是否执行。

(1) 指令的语法格式

CMP {<cond>} <Rn>, <shifter_operand>

(2) CMP指令举例

① 比较R1和立即数10并设置相关的标志位。

CMP R1, #10

② 比较寄存器R1和R2中的值并设置相关的标志位。

CMP R1, R2

CMP指令与SUBS指令的区别在于CMP指令不保存运算结果，在进行两个数据大小判断时，常用CMP指令及相应的条件码来进行操作

14. CMN指令

CMN（Compare Negative）指令使用寄存器Rn的值减去operand2的负数值（加上operand2），根据操作的结果更新CPSR中相应的条件标志位，以便后面的指令根据相应的条件标志来判断是否执行。

（1）指令的语法格式

CMN{<cond>} <Rn>,<shifter_operand>

（2）CMN指令举例

CMP Rn,#0

CMN Rn,#0

第1条指令使标志位C值为1，第2条指令使标志位C值为0
使R0值加1，判断R0是否为1的补码，若是，则Z置位

CMN R0,#1

15. ORR指令

ORR（Logical OR）为逻辑或操作指令，它将第2个源操作数 shifter_operand的值与寄存器Rn的值按位做“逻辑或”操作，结果保存到Rd中。

（1）指令的语法格式

ORR{<cond>}{S} <Rd>,<Rn>,<shifter_operand>

（2）ORR指令举例

① 设置R0中位0和1。

ORR R0, R0, #3

② 将R0的低4位置1。

ORR R0,R0,#0x0F

③ 使用ORR指令将R2的高8位数据移入到R3的低8位中。

MOV R1,R2,LSR #4

ORR R3,R1,R3,LSL #8

16. BIC位清零指令

BIC（Bit Clear）位清零指令，将寄存器Rn的值与第2个源操作数 shifter_operand的值的反码按位做“逻辑与”操作，结果保存到Rd中。

（1）指令的语法格式

BIC{<cond>}{S} <Rd>,<Rn>,<shifter_operand>

（2）BIC指令举例

① 清除R0中的位0、1和3，保持其余的不变。

BIC R0, R0, #0x1011

② 将R3的反码和R2逻辑与，结果保存到R1中。

BIC R1,R2,R3

乘法指令

ARM乘法指令完成两个数据的乘法。两个32位二进制数相乘的结果是64位的积。 影响N和Z标志位

各种形式乘法指令

操作码[23:21]	助 记 符	意 义	操 作
000	MUL	乘（保留 32 位结果）	$Rd: = (Rm \times Rs) [31:0]$
001	MLA	乘—累加（32 位结果）	$Rd: = (Rm \times Rs + Rn) [31:0]$
100	UMULL	无符号数长乘	$RdHi: RdLo: = Rm \times Rs$
101	UMLAL	无符号长乘—累加	$RdHi: RdLo: += Rm \times Rs$
110	SMULL	有符号数长乘	$RdHi: RdLo: = Rm \times Rs$
111	SMLAL	有符号数长乘—累加	$RdHi: RdLo: += Rm \times Rs$

1. MUL指令

MUL (Multiply) 32位乘法指令将Rm和Rs中的值相乘，结果的最低32位保存到Rd中。

(1) 指令的语法格式

MUL{<cond>}{S} <Rd>,<Rm>,<Rs>

(2) 指令举例

① $R1 = R2 \times R3$ 。

MUL R1, R2, R3

② $R0 = R3 \times R7$ ，同时设置CPSR中的N位和Z位。

MULS R0, R3, R7

2. MLA乘—累加指令

MLA（Multiply Accumulate）32位乘—累加指令将Rm和Rs中的值相乘，再将乘积加上第3个操作数，结果的最低32位保存到Rd中。

（1）指令的语法格式

MLA{<cond>}{S} <Rd>,<Rm>,<Rs>,<Rn>

（2）指令举例

下面指令完成 $R1 = R2 \times R3 + 10$ 的操作。

MOV R0, #0x0A

MLA R1, R2, R3, R0

3. UMULL指令

UMULL (Unsigned Multiply Long) 为64位无符号乘法指令。它将Rm和Rs中的值做无符号数相乘，结果的低32位保存到RsLo中，高32位保存到RdHi中。

(1) 指令的语法格式

UMULL{<cond>}{S} <RdLo>,<RdHi>,<Rm>,<Rs>

(2) 指令举例

下面指令完成 $(R1, R0) = R5 \times R8$ 操作。

UMULL R0, R1, R5, R8;

4. UMLAL指令

UMLAL（Unsigned Multiply Accumulate Long）为64位无符号长乘—累加指令。指令将Rm和Rs中的值做无符号数相乘，64位乘积与RdHi、RdLo相加，结果的低32位保存到RsLo中，高32位保存到RdHi中。

（1）指令的语法格式

UMALL{<cond>}{S} <RdLo>,<RdHi>,<Rm>,<Rs>

（2）指令举例

下面的指令完成 $(R1, R0) = R5 \times R8 + (R1, R0)$ 操作。

UMLAL R0, R1, R5, R8;

5. SMULL指令

SMULL (Signed Multiply Long) 为64位有符号长乘法指令。指令将Rm和Rs中的值做有符号数相乘，结果的低32位保存到RsLo中，高32位保存到RdHi中。

(1) 指令的语法格式

SMULL {<cond>} {S} <RdLo>, <RdHi>, <Rm>, <Rs>

(2) 指令举例

下面的指令完成 $(R3, R2) = R7 \times R6$ 操作。

SMULL R2, R3, R7, R6;

6. SMLAL指令

SMLAL (Signed Multiply Accumulate Long) 为64位有符号长乘—累加指令。指令将Rm和Rs中的值做有符号数相乘，64位乘积与RdHi、RdLo相加，结果的低32位保存到RsLo中，高32位保存到RdHi中。

(1) 指令的语法格式

SMLAL {<cond>} {S} <RdLo>, <RdHi>, <Rm>, <Rs>

(2) 指令举例

下面的指令完成 $(R3, R2) = R7 \times R6 + (R3, R2)$ 操作。

SMLAL R2, R3, R7, R6;

Load/Store指令

Load/Store内存访问指令在ARM寄存器和存储器之间传送数据。ARM指令中有3种基本的数据传送指令。

1. 单寄存器Load/Store指令（Single Register）

这些指令在ARM寄存器和存储器之间提供更灵活的单数据项传送方式。数据项可以是字节、16位半字或32位字。

2. 多寄存器Load/Store内存访问指令

这些指令的灵活性比单寄存器传送指令差，但可以使大量的数据更有效地传送。它们用于进程的进入和退出、保存和恢复工作寄存器以及复制存储器中的一块数据。

3. 单寄存器交换指令（Single Register Swap）

这些指令允许寄存器和存储器中的数值进行交换，在一条指令中有效地完成Load/Store操作。它们在用户级编程中很少用到。它的主要用途是在多处理器系统中实现信号量（Semaphores）的操作，以保证不会同时访问公用的数据结构。

单寄存器的Load/Store指令

用于把单一的数据传入或者传出一个寄存器。支持的数据类型有字节（8位）、半字（16位）和字（32位）。

单寄存器Load/Store指令

指 令	作 用	操 作
LDR	把存储器中的一个字装入一个寄存器	$Rd \leftarrow mem32[address]$
STR	将寄存器中的字保存到存储器	$Rd \rightarrow mem32[address]$
LDRB	把一个字节装入一个寄存器	$Rd \leftarrow mem8[address]$
STRB	将寄存器中的低 8 位字节保存到存储器	$Rd \rightarrow mem8[address]$
LDRH	把一个半字装入一个寄存器	$Rd \leftarrow mem16[address]$
STRH	将寄存器中的低 16 位半字保存到存储器	$Rd \rightarrow mem16[address]$
LDRBT	用户模式下将一个字节装入寄存器	$Rd \leftarrow mem8[address] \text{ under user mode}$
STRBT	用户模式下将寄存器中的低 8 位字节保存到存储器	$Rd \rightarrow mem8[address] \text{ under user mode}$
LDRT	用户模式下把一个字装入一个寄存器	$Rd \leftarrow mem32[address] \text{ under user mode}$
STRT	用户模式下将存储器中的字保存到寄存器	$Rd \rightarrow mem32[address] \text{ under user mode}$
LDRSB	把一个有符号字节装入一个寄存器	$Rd \leftarrow sign(mem8[address])$
LDRSH	把一个有符号半字装入一个寄存器	$Rd \leftarrow sign(mem16[address])$

3.2.3.1 单寄存器的Load/Store指令

1. LDR指令：用于从内存中将一个32位的字读取到目标寄存器

(1) 指令的语法格式

LDR {<cond>} <Rd>, <addr_mode>

(2) 指令举例

LDR R1, [R0, #0x12]	;将R0+12地址处的数据读出，保存到R1中（R0的值不变）
LDR R1, [R0]	;将R0地址处的数据读出，保存到R1中（零偏移）
LDR R1, [R0, R2]	;将R0+R2地址的数据读出，保存到R1中（R0的值不变）
LDR R1, [R0, R2, LSL #2]	;将R0+R2×4地址处的数据读出，保存到R1中（R0、R2的值不变）
LDR Rd, label	;label为程序标号，label必须是当前指令的-4K~4KB范围内
LDR Rd, [Rn], #0x04	;Rn的值用作传输数据的存储地址在数据传送后，将偏移量0x04与Rn相加，结果写回到Rn中。Rn不允许是R15

3.2.3.1 单寄存器的Load/Store指令

2. STR指令：用于将一个32位的字写入到指定的内存单元

(1) 指令的语法格式

STR {<cond>} <Rd>, <addr_mode>

(2) 指令举例

LDR/STR指令用于对内存变量的访问、内存缓冲区数据的访问、查表、外围部件的控制操作等，若使用**LDR**指令加载数据到**PC**寄存器，则实现程序跳转功能，这样也就实现了程序散转。

① 变量访问

```
NumCount EQU 0x40003000 ;定义变量NumCount
LDR R0,=NumCount        ;使用LDR伪指令装载NumCount的地址到R0
LDR R1,[R0]              ;取出变量值
ADD R1,R1,#1              ;[NumCount]=[NumCount]+1
STR R1,[R0]              ;保存变量
```

(2) 指令举例

② GPIO设置

GPIO—BASE EQU 0xe0028000; 定义GPIO寄存器的基地址

...

LDR R0, =GPIO—BASE

LDR R1, =0x00ffff00 ;将设置值放入寄存器

STR R1, [R0, #0x0C] ;IODIR=0x00ffff00, IOSET的地址为0xe002800C

(2) 指令举例

③ 程序散转

MOV R2, R2, LSL #2

;功能号乘以4，以便查表

LDR PC, [PC, R2]

;查表取得对应功能子程序地址
并跳转

NOP

FUN—TAB DCD FUN—SUB0

DCD FUN—SUB1

DCD FUN—SUB2

3.2.3.1 单寄存器的Load/Store指令

3. LDRB指令

LDRB指令根据addr_mode所确定的地址模式将一个字节读取到指令中的目标寄存器Rd的低8位，并将高位补0。

指令的语法格式：

LDR{<cond>}B <Rd>, <addr_mode>

4. STRB指令

STRB指令从寄存器中取出指定的一个字节放入存储器的低8位地址。

指令的语法格式：

STR{<cond>}B <Rd>, <addr_mode>

3.2.3.1 单寄存器的Load/Store指令

5. LDRH指令

LDRH指令用于从内存中将一个16位的半字读取到目标寄存器，高位补0。

如果指令的内存地址不是半字节对齐的，指令的执行结果不可预知。

指令的语法格式：

LDR{<cond>}H <Rd>,<addr_mode>

6. STRH指令

STRH指令从寄存器中取出指定的16位半字放入存储器的低16位。

指令的语法格式：

STR{<cond>}H <Rd>,<addr_mode>

3.2.3.2 多寄存器的Load/Store内存访问指令

多寄存器的Load/Store内存访问指令也叫批量加载/存储指令，它可以实现在一组寄存器和一块连续的内存单元之间传送数据。**LDM**用于加载多个寄存器，**STM**用于存储多个寄存器。多寄存器的Load/Store内存访问指令允许一条指令传送**16**个寄存器的任何子集或所有寄存器。多寄存器的Load/Store内存访问指令主要用于现场保护、数据复制和参数传递等。

多寄存器的Load/Store内存访问指令：

指 令↵	作 用↵	操 作↵
LDM↵	装载多个寄存器↵	$\{Rd\} * N \leftarrow mem32[start\ address + 4 * N] \leftarrow$
STM↵	保存多个寄存器↵	$\{Rd\} * N \rightarrow mem32[start\ address + 4 * N] \leftarrow$

LDM (Load Multiple registers)

STM (Store Multiple registers)

3.2.3.2 多寄存器的Load/Store内存访问指令

1. LDM指令

LDM指令将数据从连续的内存单元中读取到指令中指定的寄存器列表中的各寄存器中。当PC包含在LDM指令的寄存器列表中时，指令从内存中读取的字数据将被作为目标地址值，指令执行后程序将从目标地址处开始执行，从而实现了指令的跳转。

指令的语法格式：

`LDM{<cond>}<addressing_mode> <Rn>{!}, <registers>`

寄存器R0~R15分别对应于指令编码中bit[0]~bit[15]位。如果Ri存在于寄存器列表中，则相应的位等于1，否则为0。

如果 **!** 存在时，将最终地址写回 Rn

条件指令仅在程序状态寄存器中的条件标志匹配时执行。

例如，BEQ（带有EQ条件的B指令）仅在设置Z标志时分支。

如果{cond}字段为空，则始终执行指令

3.2.3.2 多寄存器的Load/Store内存访问指令

2. STM指令

STM指令将指令中寄存器列表中的各寄存器数值写入到连续的内存单元中。主要用于块数据的写入、数据栈操作及进入子程序时保存相关寄存器的操作。

指令的语法格式：

STM{<cond>}<addressing_mode> <Rn>{!}, <registers>

3.2.3.2 多寄存器的Load/Store内存访问指令

3、数据传送指令应用

LDM/STM批量加载/存储指令可以实现在一组寄存器和一块连续的内存单元之间传输数据。**LDM**为加载多个寄存器，**STM**为存储多个寄存器。允许一条指令传送**16**个寄存器的任何子集或所有寄存器。

指令格式如下

LDM {cond} <模式> Rn{!}, regist{^}

STM {cond} <模式> Rn{!}, regist{^}

*{^} 为可选后缀，

当指令为**LDM**且寄存器列表中包含**R15**，选用该后缀时表示：除了正常的数据传送外，还将**SPSR**复制到**CPSR**中。

同时，该后缀还表示传入或传出的是用户模式下的寄存器，而不是当前模式下的寄存器。

3.2.3.2 多寄存器的Load/Store内存访问指令

LDM/STM的主要用途有现场保护、数据复制和参数传递等。其模式有8种，其中前面4种用于数据块的传输

- ❖ IA:每次传送后地址加4
- ❖ IB:每次传送前地址加4
- ❖ DA:每次传送后地址减4
- ❖ DB:每次传送前地址减4

后面4种是堆栈操作 根据堆栈的不同种类，将其寻址方式分为以下4种

- ❖ 满递减FD (Full Descending)
- ❖ 空递减ED (Empty Descending)
- ❖ 满递增FA (Full Ascending)
- ❖ 空递增EA (Empty Ascending)

Registers

Register	Value
Current	
R0	0x038617d0
R1	0x03fefbfff
R2	0x03fefbfff
R3	0x03fefbfff
R4	0x00000000
R5	0x03810f24
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x005448b4
R11	0x00000000
R12	0x002d583c
R13 (SP)	0x03fffbf0
R14 (LR)	0x000002ac
R15 (PC)	0x002d5840
CPSR	0x600000d3
SPSR	0x00000010
User/System	
Fast Interrupt	
Interrupt	
Supervisor	
Abort	
Undefined	
Internal	
PC \$	0x002D5840
Mode	Supervisor

Disassembly

0x002D583C E92D4070 STMDB R13!, (R4-R6,R14)

0x002D5840 E1A04000 MOV R4,R0

0x002D5844 E1A05001 MOV R5,R1

Memory 1

Address: 0x03fffbf0

0x03FFFBF0:	00 00 00 00 77 02 00 00 D0 17 86 03 FF FB FE 03
0x03FFFBF0:	00 00 00 00 24 0F 81 03 00 00 00 00 AC 02 00 00
0x03FFFC00:	EA D7 E8 F5 EF D4 E9 D6 E9 D6 CA D4 E8 D4 EC D5
0x03FFFC10:	EB D5 E8 F6 EF D6 F8 C5 EA D4 D8 D7 C7 D4 EB D5
0x03FFFC20:	EB D4 EB D6 F8 D4 F8 D4 E8 D5 D4 D4 EA C3 EB DC
0x03FFFC30:	E8 D4 E8 D4 E8 D6 EA D4 EB D6 C8 D4 E9 D6 C8 D4
0x03FFFC40:	E8 C6 E8 D6 EA C2 E8 D4 E8 D5 E9 D6 E8 D5 EA D7
0x03FFFC50:	E8 D6 EA D5 E2 D0 E8 D6 ED C6 E1 D4 EA D6 EA C4
0x03FFFC60:	E9 D4 C9 D4 EF D4 EB D6 E9 D4 E8 D5 E8 F4 EA F5
0x03FFFC70:	C8 D4 EB D2 EA D4 E8 D4 C9 D4 E8 D4 E8 D4 E8 D5
0x03FFFC80:	E9 D4 E8 D5 EC D6 E9 D4 E8 D5 EB D5 E8 C6 E8 D6
0x03FFFC90:	E9 D6 E0 D5 FB D3 FA C5 C8 D5 CA D1 EC D4 EB CC
0x03FFFCa0:	C8 D5 EB D0 E8 D6 E9 D6 EA C7 E0 C5 F8 D6 E9 D4
0x03FFFCB0:	EB D5 EB D5 E8 D4 ED C4 E9 D5 EA D5 EA D4 AB F6
0x03FFFCc0:	EA C4 EA C6 E8 D6 E8 D4 FB D6 EA D1 ED D6 E9 D7
0x03FFFCd0:	E1 D7 F8 D6 ED D4 E9 D7 EB D0 E9 D5 EB D5 E8 D1
0x03FFFCe0:	F3 D6 EC D6 E8 F5 E8 D4 E8 D7 E8 D4 FB D4 E9 D7
0x03FFFCf0:	E9 D4 F9 D4 E9 D6 E8 D4 E9 D5 C2 D4 E8 D5 E8 D4
0x03FFFD00:	EB C0 E9 D4 FA D6 EB D4 EB D6 E9 D5 EA D7 E8 D4
0x03FFFD10:	C8 DD E8 C4 EB D6 EA D4 EA DC EA F6 EA D4 E8 D4

Annotations

R4

R5

R6

R14

SP

3.2.3.2 多寄存器的Load/Store内存访问指令

批量数据寻址方式	堆栈寻址方式	L 位	P 位	U 位
LDMDA	LDMFA	1	0	0
LDMIA	LDMFD	1	0	1
LDMDB	LDMEA	1	1	0
LDMIB	LDMED	1	1	1
STMDA	STMED	0	0	0
STMIA	STMEA	0	0	1
STMDB	STMFD	0	1	0
STMIB	STMFA	0	1	1

		向 上 生 长		向 下 生 长	
		满	空	满	空
增 加	之前	STMIB			LDMIB
		STMFA			LDMED
	之后		STMIA	LDMIA	
			STMEA	LDMFD	
增 加	之前		LDMDB	STMDB	
			LDMEA	STMFD	
	之后	LDMDA			STMDA
		LDMFA			STMED

3.2.3.2 多寄存器的Load/Store内存访问指令

【举例】 使用LDM/STM进行数据复制。

```
LDR R0,=SrcData           ;设置源数据地址
LDR R1,=DstData           ;设置目标地址
LDMIA R0,{R2~R9}          ;加载8字数据到寄存器R2~R9
STMIA R1,{R2~R9}          ;存储寄存器R2~R9到目标地址
```

【举例】 使用LDM/STM进行现场寄存器保护，常在子程序或异常处理使用。

SEND BYTE

```
STMFD SP!,{R0~R7,LR} ;寄存器压栈保护
```

...

```
BL DELAY                ;调用DELAY子程序
```

...

```
LDMFD SP!,{R0~R7,PC};恢复寄存器，并返回
```

3.2.3.3 单数据交换指令

交换指令是Load/Store指令的一种特例，它把一个寄存器单元的内容与寄存器内容交换。交换指令是一个原子操作（Atomic Operation），也就是说，在连续的总线操作中读/写一个存储单元，在操作期间阻止其他任何指令对该存储单元的读/写。

交换指令SWP：

指 令↵	作 用↵	操 作↵
SWP↵	字交换↵	tmp=mem32[Rn]↵ mem32[Rn]=Rm↵ Rd=tmp↵
SWPB↵	字节交换↵	tmp=mem8[Rn]↵ mem8[Rn]=Rm↵ Rd=tmp↵

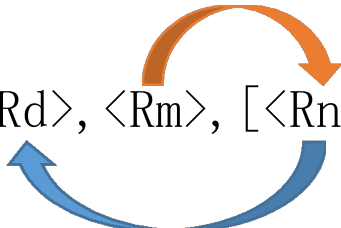
3.2.3.3 单数据交换指令

1. SWP字交换指令

SWP指令用于将内存中的一个字单元和一个指定寄存器的值相交换。操作过程如下：假设内存单元地址存放在寄存器<Rn>中，指令将<Rn>中的数据读取到目的寄存器Rd中，同时将另一个寄存器<Rm>的内容写入到该内存单元中。当<Rd>和<Rm>为同一个寄存器时，指令交换该寄存器和内存单元的内容。

指令的语法格式：

SWP {<cond>} <Rd>, <Rm>, [<Rn>]



其中Rd为目的寄存器，存储从存储器中加载的数据，同时，Rm中的数据将会被存储到存储器中。若Rm与Rn相同，则为寄存器与存储器内容进行交换。Rn为要进行数据交换的存储器地址，Rn不能与Rd和Rm相同。

3.2.3.3 单数据交换指令

2. SWPB字节交换指令

SWPB指令用于将内存中的一个字节单元和一个指定寄存器的低8位值相交换，操作过程如下：假设内存单元地址存放在寄存器<Rn>中，指令将<Rn>中的数据读取到目的寄存器Rd中，寄存器Rd的高24位设为0，同时将另一个寄存器<Rm>的低8位内容写入到该内存字节单元中。当<Rd>和<Rm>为同一个寄存器时，指令交换该寄存器低8位内容和内存字节单元的内容。

指令的语法格式：

SWP{<cond>}B <Rd>,<Rm>,[<Rn>]

SWP指令举例：

SWP R1, R1, [R0] ;将R1的内容与R0指向的存储单元内容进行交换

SWPB R1, R2, [R0] ;将R0指向的存储单元内容读取一字节数据到R1中（高24位清零），并将R2的内容写入到该内存单元中（最低字节有效）

使用SWP指令可以方便地进行信号量操作。

```
12C_SEM      EQU      0x40003000
```

...

```
12C_SEM_WAIT
```

```
        MOV        R0, #0
```

```
        LDR        R0, =12C_SEM
```

```
        SWP        R1, R1, [R0]          ;取出信号量，并将其设为0
```

```
        CMP        R1, #0                ;判断是否有信号
```

```
        BEQ        12C_SEM_WAIT ;若没有信号则等待
```

3.2.4 跳转指令



- ✎ 跳转（B）和跳转连接（BL）指令是改变指令执行顺序的标准方式。ARM一般按照字地址顺序执行指令，需要时使用条件执行跳过某段指令。只要程序必须偏离顺序执行，就要使用控制流指令来修改程序计数器。
- ✎ 跳转指令改变程序的执行流程或者调用子程序。这种指令使得一个程序可以使用子程序、if-then-else结构及循环。执行流程的改变迫使程序计数器（PC）指向一个新的地址，ARMv5架构指令集包含的跳转指令如表所示。

助 记 符↵	说 明↵	操 作↵
B↵	跳转指令↵	$pc \leftarrow label$ ↵
BL↵	带返回的连接跳转↵	$pc \leftarrow label (lr \leftarrow BL \text{ 后面的第一条指令})$ ↵
BX↵	跳转并切换状态↵	$pc \leftarrow Rm \& 0xffffffffe, T \leftarrow Rm \& 1$ ↵
BLX↵	带返回的跳转并切换状态↵	$pc \leftarrow lable, T \leftarrow 1$ ↵ $pc \leftarrow Rm \& 0xffffffffe, T \leftarrow Rm \& 1$ ↵ $lr \leftarrow BL \text{ 后面的第一条指令}$ ↵

1. 跳转指令B及带连接的跳转指令BL

跳转指令B使程序跳转到指定的地址执行程序。

带连接的跳转指令BL将下一条指令的地址拷贝到R14（即返回地址连接寄存器LR）寄存器中，然后跳转到指定地址运行程序。

（1）指令的语法格式

$B\{L\} \{ \langle \text{cond} \rangle \} \quad \langle \text{target_address} \rangle$

BL指令用于实现子程序调用。子程序的返回可以通过将LR寄存器的值复制到PC寄存器来实现。

1. 跳转指令B及带连接的跳转指令BL

下面三种指令可以实现子程序返回。

① BX R14（如果体系结构支持BX指令）。

② MOV PC, R14。

③ 当子程序在入口处使用了压栈指令：

STMFD R13!, {<registers>, R14}

可以使用指令：

LDMFD R13!, {<registers>, PC}

将子程序返回地址放入PC中。

1. 跳转指令B及带连接的跳转指令BL

- ❖ 需要注意的是，B和BL这两条指令和目标地址处的指令都要属于ARM指令集。
- ❖ 两条指令都可以根据CPSR中的条件标志位的值决定指令是否执行。

(2) 程序举例

①程序跳转到LABEL标号处。

```
B    LABEL ;  
ADD   R1, R2, #4  
ADD   R3, R2, #8  
SUB   R3, R3, R1  
LABEL  
SUB   R1, R2, #8
```

② 跳转到绝对地址0x1234处。

```
B    0x1234
```

③ 跳转到子程序func处执行，同时将当前PC值保存到LR中。

```
BL   func
```

④ 条件跳转：当CPSR寄存器中的C条件标志位为1时，程序跳转到标号LABEL处执行。

```
BCC  LABEL
```

⑤ 通过跳转指令建立一个无限循环。

LOOP

ADD R1, R2, #4

ADD R3, R2, #8

SUB R3, R3, R1

B LOOP

⑥ 通过使用跳转使程序体循环10次。

MOV R0, #10

LOOP

SUBS R0, #1

BNE LOOP

⑦ 条件子程序调用示例。

CMP R0, #5 ;如果R0<5

BLLT SUB1 ;则调用

BLGE SUB2 ;否则调用SUB2

2. BX带状态切换的跳转指令BX

BX使程序跳转到指令中指定的参数Rm指定的地址执行程序，Rm的第0位拷贝到CPSR中T位，bit[31:1]移入PC。若Rm的bit[0]为1，则跳转时自动将CPSR中的标志位T置位，即把目标地址的代码解释为Thumb代码；若Rm的位bit[0]为0，则跳转时自动将CPSR中的标志位T复位，即把目标地址代码解释为ARM代码。

(1) 指令的语法格式

BX{<cond>} <Rm>

① 当Rm[1:0]=0x10时，指令的执行结果不可预知。因为在ARM状态下，指令是4字节对齐的。

② PC可以作为Rm寄存器使用，但这种用法不推荐使用。当PC作为<Rm>使用时，指令“BX PC”将程序跳转到当前指令下面第二条指令处执行。虽然这样跳转可以实现，但最好使用下面的指令完成这种跳转。

MOV PC, PC 或 ADD PC, PC, #0

2. BX带状态切换的跳转指令BX

(2) 指令举例

- ① 转移到R0中的地址，如果R0[0]=1，则进入Thumb状态。

BX R0;

- ② 跳转到R0指定的地址，并根据R0的最低位来切换处理器状态

ADRL R0, ThumbFun+1 ;

BX R0;

3. BXL带状态切换的连接跳转指令BLX

带连接和状态切换的跳转指令（Branch with Link Exchange, BLX）使用标号，用于使程序跳转到Thumb状态或从Thumb状态返回。该指令为无条件执行指令，并用分支寄存器的最低位来更新CPSR中的T位，将返回地址写入到连接寄存器LR中。

（1）语法格式

BLX <target_add>

其中，<target_add>为指令的跳转目标地址。该地址根据以下规则计算。

- ① 将指令中指定的24位偏移量进行符号扩展，形成32位立即数。
- ② 将结果左移两位。
- ③ 位H（bit[24]）加到结果地址的第一位（bit[1]）。
- ④ 将结果累加进程序计数器（PC）中。

3. BXL带状态切换的连接跳转指令BLX

(2) 指令的使用

- ① 从Thumb状态返回到ARM状态，使用BX指令。

BX R14

- ② 可以在子程序的入口和出口增加栈操作指令。

PUSH {<registers>,R14}

...

POP {<registers>,PC}

3.2.5 状态操作指令

ARM指令集提供了两条指令，可直接控制程序状态寄存器（Program State Register, PSR）。MRS指令用于把CPSR或SPSR的值传送到一个寄存器；MSR与之相反，把一个寄存器的内容传送到CPSR或SPSR。这两条指令相结合，可用于对CPSR和SPSR进行读/写操作。

程序状态寄存器指令

指 令	作 用	操 作
MRS	把程序状态寄存器的值送到一个通用寄存器	Rd=SPR
MSR	把通用寄存器的值送到程序状态寄存器或把一个立即数送到程序状态字	PSR[field]=Rm 或 PSR[field]=immediate

在指令语法中可看到一个称为fields的项，它可以是控制（C）、扩展（X）、状态（S）及标志（F）的组合。

3.2.5 状态操作指令

1. MRS: MRS指令用于将程序状态寄存器的内容传送到通用寄存器中。

(1) 指令的语法格式: $\text{MRS}\{\text{cond}\} \text{ Rd}, \text{PSR}$

Rd为目标寄存器, Rd不允许为程序计数器(PC)。PSR为CPSR或SPSR。

(2) 指令举例

MRS	R1, CPSR	;将CPSR状态寄存器读取, 保存到R1中
MRS	R2, SPSR	;将SPSR状态寄存器读取, 保存到
		R1中

MRS指令读取CPSR, 可用来判断ALU的状态标志及IRQ/FIQ中断是否允许等; 在异常处理程序中, 读SPSR可指定进入异常前的处理器状态等。

3.2.5 状态操作指令

2. MSR：用于将通用寄存器的内容传送到程序状态寄存器中。

(1) 指令的语法格式

MSR{cond} PSR_field, #immed_8r

MSR{cond} PSR_field, Rm

其中，PSR是指CPSR或SPSR。〈fields〉设置状态寄存器中需要操作的位。状态寄存器的32位可以分为4个8位的域（field）。bits[31: 24]为条件标志位域，用f表示；bits[23: 16]为状态位域，用s表示；bits[15: 8]为扩展位域，用x表示；bits[7: 0]为控制位域，用c表示；immed_8r为要传送到状态寄存器指定域的立即数，8位；Rm为要传送到状态寄存器指定域的数据源寄存器。

(2) 指令举例

MSR CPSR_c, #0xD3 ;CPSR[7:0]=0xD3, 切换到管理模式

MSR CPSR_cxsf, R3 ;CPSR=R3

3.2.5 状态操作指令

3. 程序状态寄存器指令的应用

【举例】 使能IRQ中断。

ENABLE_IRQ

```
MRS    R0, CPSR
BIC     R0, R0, #0x80
MSR     CPSR_c, R0
MOV     PC, LR
```

【举例】 禁止IRQ中断。

DISABLE_IRQ

```
MRS     R0, CPSR
ORR     R0, R0, #0x80
MSR     CPSR_c, R0
MOV     PC, LR
```

3.2.5 状态操作指令

3. 程序状态寄存器指令的应用

【【举例】 堆栈指令初始化。

INITSTACK

MOV R0, LR ;保存返回地址

;设置管理模式堆栈

MSR CPSR_c, #0xD3

LDR SP, StackSvc

;设置中断模式堆栈

MSR CPSR_c, #0xD2

LDR SP, StackSvc

3.2.5 状态操作指令

注意：

- ✎ 特权模式下才能修改状态寄存器
- ✎ 不能通过MSR指令直接修改CPSR的T位控制位来实现ARM和Thumb状态的转换，必须用BX(BXL)完成
- ✎ MRS与MSR配合使用，实现CPSR或SPSR寄存器的读—修改—写操作，可用来进行处理器模式切换，允许/禁止IRQ/FIQ中断等设置。另外，进程切换或允许异常中断嵌套时，也需要使用MRS指令读取SPSR状态值并保存起来。

3.2.6 协处理器指令

ARM可支持多达16个协处理器，主要的作用：ARM处理器初始化，ARM与协处理器的数据处理操作，ARM的寄存器与协处理器的寄存器之间传送数据，以及ARM协处理器的寄存器和存储器之间传送数据。

ARM协处理器指令可分为以下3类。

- (1) 协处理器数据操作， CDP。
- (2) 协处理器数据传送指令，包括LDC和STC。
- (3) 协处理器寄存器传送指令，包括MCR和MRC。



助 记 符↵	操 作↵
CDP↵	协处理器数据操作↵
LDC↵	装载协处理器寄存器↵
MCR↵	从 ARM 寄存器传数据到协处理器寄存器↵
MRC↵	从协处理器寄存器传数据到 ARM 寄存器↵
STC↵	存储协处理器寄存器↵

3.2.7 异常产生指令

ARM指令集中提供了两条产生异常的指令，通过这两条指令可以用软件的方法实现异常。

ARM异常产生指令

助 记 符↵	含 义↵	操 作↵
SWI↵	软中断指令↵	产生软中断，处理器进入管理模式↵
BKPT↵	断点中断指令↵	处理器产生软件断点↵

1. 中断指令

软件中断指令（Software Interrupt, SWI）用于产生软中断，从而实现从用户模式变换到管理模式，CPSR保存到管理模式的SPSR中，执行转移到SWI向量，在其他模式下也可以使用SWI指令，处理器同样切换到管理模式。

（1）指令的语法格式

SWI {<cond>} <immed_24>

（2）指令举例

① 下面指令产生软中断，中断立即数为0。

SWI 0;

② 产生软中断，中断立即数为0x123456。

SWI 0x123456;

1. 中断指令

③ 使用**SWI**指令时，通常使用以下两种方法进行参数传递。

指令**24**位的立即数指定了用户请求的类型，中断服务程序的参数通过寄存器传递。

下面的程序产生一个中断号为**12**的软中断。

MOV R0, # 34 ;设置功能号为**34**

SWI 12 ;产生软中断，中断号为**12**

另一种情况，指令中的**24**位立即数被忽略，用户请求的服务类型由寄存器**R0**的值决定，参数通过其他寄存器传递。

下面的例子通过**R0**传递中断号，**R1**传递中断的子功能号。

MOV R0, # 12 ;设置**12**号软中断

MOV R1, # 34 ;设置功能号为**34**

SWI 0 ;

2. 断点中断指令（BreakPoint, BKPT）

断点中断指令（BreakPoint, BKPT）产生一个预取异常（Prefetch Abort），它常被用来设置软件断点，在调试程序时十分有用。当系统中存在调试硬件时，该指令被忽略。

指令格式如下：BKPT <immediate>

要正确地使用BKPT指令，必须和具体的调试系统相结合。一般来说，BKPT有两种使用方法。

（1）如果当前使用的系统调试硬件没有屏蔽BKPT指令，那么在此系统中预取指令异常和软件调试命令同时使用一个中断向量。这样当异常发生时，就要依靠系统自身来判断是真正地预取异常还是软件调试命令。根据系统的不同，判断的方法也有所不同。

（2）如果当前的系统调试硬件屏蔽了BKPT指令，那么系统会跳过BKPT指令顺序执行该指令下面的程序代码。

3.3 小结

- ❖ 本章在第2章的基础上，介绍了ARM处理器的寻址方式及ARM处理器的指令集。ARM处理器的寻址方式包括：数据处理指令寻址方式和内存访问指令寻址方式；ARM处理器的指令集包括：数据操作指令、乘法指令、load/store指令、跳转指令、状态操作指令、协处理器指令、异常产生指令。

3.4 思考与练习

- ❖ 3-1 用ARM汇编实现下面列出的操作
- ❖ a) $r0=15$ b) $r0=r1/16$ （有符号数）
- ❖ c) $r1=r2*3$ d) $r0=-r0$
- ❖ 3-2 BIC指令的作用是？
- ❖ 3-3 执行SWI指令时会发生什么？
- ❖ 3-4 B、BL、BX指令的区别？
- ❖ 3-5 下面哪个数据可以作为数据操作指令的有效立即数
- ❖ a) 0x101 b) 0x1f8 c) 0xf000000f d) 0x08000012 e) 0x104
- ❖ 3-6 ARM在哪些工作工作模式下可以修改CPSR寄存器？
- ❖ 3-7 写一个程序，判断R0的值大于0x50，则将R1的值减去0x10，并把结果送给R0。
- ❖ 3-8 编写一段ARM汇编程序，实现数据块拷贝，将R0指向的8个字的连续数据保存到R1指向的一段连续的内存单元。

谢谢！