# 常用 ARM 指令集及汇编

2003 年 11 月 24 日

# 前言

ARM（Advanced RISC Machines）是微处理器行业的一家知名企业，该企业设计了大量高性能、廉价、耗能低的 RISC 处理器、相关技术及软件。技术具有性能高、成本低和能耗省的特点，适用于多种领域，比如嵌入控制、消费/教育类多媒体、DSP 和移动式应用等。

ARM 将其技术授权给世界上许多著名的半导体、软件和 OEM 厂商，每个厂商得到的都是独一无二的ARM相关技术及服务,利用这种合作关系,ARM很快成为许多全球性RISC标准的缔造者。

目前，ARM内核的微处理器正在我国迅速普及和发展，许多朋友已经开始着手学习、研究和利用这类芯片进行商品化设计,为了帮助广大朋友更快的学习和使用这种先进的技术，**广州周立功单片机发展有限公司**(http://www.zlgmcu.com)曾面向大众推出了一款EasyARM试验板,本文的大部分内容来自此款试验板的配套教材--《ARM微控制器基础》，并且得到周立功正式授权。本文主要是针对ARM7TDMI(-S)内核，经本人重新排版，制作成PDF格式，便于大家查阅和使用。

本文仅仅用于大家的学习和研究使用，由于本文的原因造成元器件的损坏、设计方案的缺陷与失败等一切损失，概不负责。

感谢周立功老师为我们提供了丰富的资源并亲自审阅了本文，感谢**单片机学习网**(http://www.mcustudy.com)为我们提供了交流和学习的空间，感谢我的女友对我这项工作的支持，她自己放弃了大量的业余时间帮我录入和排版。

由于时间仓促,本文难免会存在不当之处,欢迎大家来信(email:mcu8031@163.com)或在网上(QQ:57523799)批评指正和交流，以便在今后的版本加以更正，也希望大家能把自己在工作中的心得体会和经验拿出来，和众多的同行及爱好者交流，共同提高，一起进步。

宛城布衣

2003 年 11 月 24 日星期一

# 目录

# ARM7TDMI(-S)指令集及汇编

ARM 处理器是基于精简指令集计算机（RISC）原理设计的，指令集和相关译码机制较为简单，ARM7TDMI(-S)具有 32 位 ARM 指令集和 16 位 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。所有 ARM 指令都是可以有条件执行的，而 Thumb 指令仅有一条指令具备条件执行功能。ARM 程序和 Thumb 程序可相互调用，相互之间的状态切换开销几乎为零。

# ARM 处理器寻址方式

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 处理器有 9 种基本寻址方式。

## 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操作。

寄存器寻址指令举例如下：

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

## 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就是操作数本身，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（立即数）。

立即寻址指令举例如下：

```
SUBS   R0,R0,#1       ;R0 - 1 ->  R0
MOV    R0,#0xff00     ;0xff00 ->  R0
```

立即数要以“＃”为前缀，表示 16 进制数值时以“0x”表示。

## 寄存器偏移寻址

寄存器偏移寻址是ARM指令集特有的寻址方式，当第 2 操作数是寄存器偏移方式时，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。

寄存器偏移寻址方式指令举例如下：

```
MOV    R0,R2,LSL #3       ;R2 的值左移 3 位，结果放入 R0，即 R0 = R2 * 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
```

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

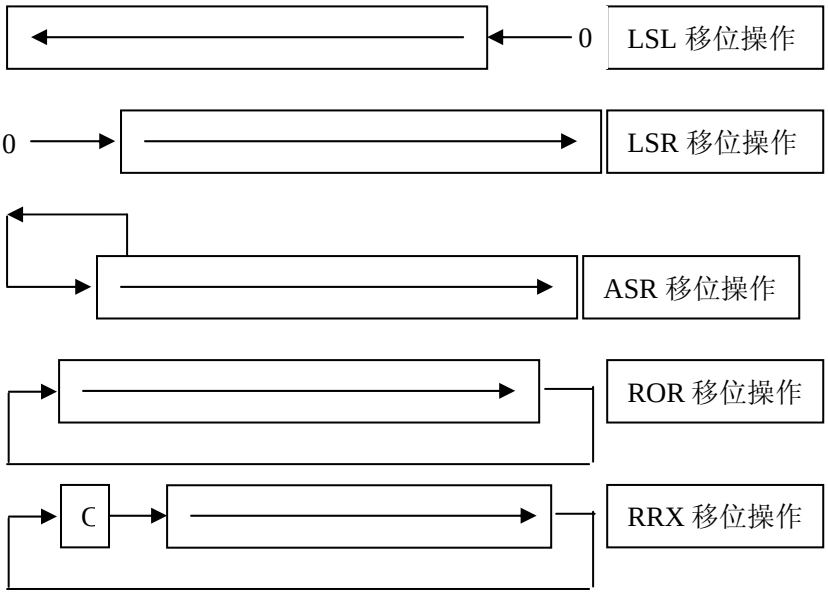
LSR：逻辑右移（Logical Shift Right），寄存器中字的高端空出的位补 0

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 1place），操作数右移一位，高端空出的位用原 C 标志值填充。

各移位操作如下图所示。




## 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。

寄存器间接寻址指令举例如下：

```
LDR   R1,[R2]    ;将 R2 中的数值作为地址，取出此地址中的数据保存在 R1 中
SWP   R1,R1,[R2];将如中的数值作为地址，取出此地址中的数值与 R1 中的值交换
```

## 基址寻址

基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，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,#0x0F] ;将 R3 中的数值加 0x0F 作为地址，取出此地址的数值保存在 R2 中
STR     R1,[R0,#-2] ;将 R0 中的数值减 2 作为地址，把 R1 中的内容保存到此地址位置
```

## 多寄存器寻址

多寄存器寻址就是一次可以传送几个寄存器值，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 单元中的数据读出到 R2－R7，R12，R1 自动加 1
STMIA   R0!，{R3-R6,R10};将 R3-R6，R10 中的数据保存到 R0 指向的地址，R0 自动加 1
```

使用多寄存器寻址指令时，寄存器子集的顺序时由小到大的顺序排列，连续的寄存器可用“－”连接，否则，用“,”分隔书写。

## 堆栈寻址

堆栈是特定顺序进行存取的存储区，操作顺序分为“后进先出”和“先进后出”，堆栈寻址时隐含的，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就是堆栈的栈顶。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

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要放入的空位置，称为空堆栈。这样就有 4 中类型的堆栈表示递增和递减的满堆栈和空堆栈的各种组合。

**满递增**：堆栈通过增大存储器的地址向上增长，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**：堆栈通过增大存储器的地址向上增长，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：堆栈通过减小存储器的地址向下增长，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**：堆栈通过减小存储器的地址向下增长，堆栈指针指向堆栈下的第一个空位置。指令如 LDMED，STMED 等。

堆栈寻址指令举例如下：

```
STMFD     SP!,{R1-R7,LR}   ; 将 R1～R7，LR 入栈。满递减堆栈。
LDMFD     SP!,{R1-R7,LR}   ;数据出栈，放入 R1～R7，LR 寄存器。满递减堆栈。
```

## 块拷贝寻址

多寄存器传送指令用于一块数据从存储器的某一位置拷贝到另一位置。

块拷贝寻址指令举例如下：

```
STMIA   R0!,{R1-R7}   ;将 R1～R7 的数据保存到存储器中，存储器指针在保存第一
                      ;个值之后增加，增长方向为向上增长。
STMIB   R0!,{R1-R7}   ;将 R1～R7 的数据保存到存储器中，存储器指针在保存第一
                      ;个值之前增加，增长方向为向上增长。
STMDA   R0!,{R1-R7}   ;将 R1～R7 的数据保存到存储器中，存储器指针在保存第一
                      ;个值之后增加，增长方向为向下增长。
STMDB   R0!,{R1-R7}   ;将 R1～R7 的数据保存到存储器中，存储器指针在保存第一
                      ;个值之前增加，增长方向为向下增长。
```

## 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      ROUTE1      ;调用到 ROUTE1 子程序
BEQ     LOOP        ;条件跳转到 LOOP 标号处
…
LOOP    MOV  R2,#2
```

```
…
ROUTE1
…
```

# 指令集介绍

# ARM 指令集

## 指令格式

基本格式

<opcode>{<cond>}{S} <Rd>,<Rn>{,<opcode2>}

其中，<>内的项是必须的，{}内的项是可选的，如<opcode>是指令助记符，是必须的，而{<cond>}为指令执行条件，是可选的，如果不写则使用默认条件 AL(无条件执行)。

| | |
|---|---|
| opcode | 指令助记符，如 LDR，STR 等 |
| cond | 执行条件，如 EQ，NE 等 |
| S | 是否影响 CPSR 寄存器的值，书写时影响 CPSR，否则不影响 |
| Rd | 目标寄存器 |
| Rn | 第一个操作数的寄存器 |
| operand2 | 第二个操作数 |

指令格式举例如下：

```
LDR     R0,[R1]     ;读取 R1 地址上的存储器单元内容，执行条件 AL
BEQ     DATAEVEN    ;跳转指令，执行条件 EQ，即相等跳转到 DATAEVEN
ADDS    R1,R1,#1    ;加法指令，R1＋1＝R1 影响 CPSR 寄存器，带有 S
SUBNES R1,R1,#0xD;条件执行减法运算(NE),R1-0xD=>R1,影响 CPSR 寄存器,带有 S
```

## 第 2 个操作数

在 ARM 指令中，灵活的使用第 2 个操作数能提高代码效率，第 2 个操作数的形式如下：

### #immed_8r

常数表达式，该常数必须对应 8 位位图，即常数是由一个 8 位的常数循环移位偶数

位得到。

合法常量：

0x3FC、0、0xF0000000、200，0xF0000001

非法常量：

0x1FE、511、0xFFFF、0x1010、0xF0000010

常数表达式应用举例如下：

```
MOV   R0,#1          ;R0=1
AND   R1,R2,#0x0F    ;R2 与 0x0F，结果保存在 R1
LDR   R0，[R1],#-4 ;读取 R1 地址上的存储器单元内容，且 R1＝R1－4
```

### Rm

寄存器方式，在寄存器方式下操作数即为寄存器的数值。

寄存器方式应用举例：

```
SUB  R1，R1，R2      ；R1-R2=>R1
MOV  PC，R0          ；PC=R0，程序跳转到指定地址
LDR  R0，[R1]，-R2   ；读取 R1 地址上的存储器单元内容并存入 R0，且 R1=R1-R2
```

### Rm,shift

寄存器移位方式。将寄存器的移位结果作为操作数，但 RM 值保存不变，移位方法如下：

ASR #n　算术右移 n 位（1≤n≤32）

LSL #n　逻辑左移 n 位（1≤n≤31）

LSR #n　逻辑左移 n 位（1≤n≤32）

ROR #n　循环右移 n 位（1≤n≤31）

RRX　带扩展的循环右移 1 位

type　Rs　其中，type 为 ASR，LSL，和 ROR 中的一种；Rs 偏移量寄存器，低 8 位有效，若其值大于或等于 32，则第 2 个操作数的结果为 0（ASR、ROR 例外）。

寄存器偏移方式应用举例：

```
ADD    R1，R1，R1，LSL #3  ；R1=R1*9
```

```
SUB     R1，R1，R2，LSR#2  ；R1=R1-R2*4
```

R15 为处理器的程序计数器 PC，一般不要对其进行操作，而且有些指令是不允许使用 R15，如 UMULL 指令。

## 条件码

使用指令条件码，可实现高效的逻辑操作，提高代码效率。

条件码表

| 条件码助记符 | 标志 | 含义 |
|---|---|---|
| EQ | Z＝1 | 相等 |
| NE | Z＝0 | 不相等 |
| CS/HS | C=1 | 无符号数大于或等于 |
| CC/LO | C=0 | 无符号数小于 |
| MI | N=1 | 负数 |
| PL | N=0 | 正数或零 |
| VS | V=1 | 溢出 |
| VC | V=0 | 没有溢出 |
| HI | C=1，Z=0 | 无符号数大于 |
| LS | C=0，Z=1 | 无符号数小于或等于 |
| GE | N=V | 带符号数大于或等于 |
| LT | N！=V | 带符号数小于 |
| GT | Z=0，N=V | 带符号数大于 |
| LE | Z=1，N！=V | 带符号数小于或等于 |
| AL | 任何 | 无条件执行（指令默认条件） |

对于 Thumb 指令集，只有 B 指令具有条件码执行功能，此指令条件码同表 2.1,但如果为无条件执行时,条件码助记符“AL”不能在指令中书写。

条件码应用举例如下：

比较两个值大小，并进行相应加 1 处理，C 代码为

```
if（a＞b）a++;
else  b++;
```

对应的 ARM 指令如下。其 R0 为 a，R1 为 b。

```
CMP     R0，R1          ；R0 与 R1 比较
ADDHI   R0，R0，#1      ；若 R0＞R1，则 R0=R0+1
ADDLS   R1，R1，#1      ；若 R0＜=R1，则 R1=R1+1
```

若两个条件均成立，则将这两个数值相加，C 代码为

If（（a!=10）&&(b!=20)） a=a+b;

对应的 ARM 指令如下.其中 R0 为 a,R1 为 b.

```
CMP     R0,#10      ;比较 R0 是否为 10
CMPNE   R1,#20      ;若 R0 不为 10,则比较 R1 是否 20
ADDNE   R0,R0,R1    ;若 R0 不为 10 且 R1 不为 20,指令执行,R0=R0+R1
```

## ARM 存储器访问指令

ARM 处理是加载/存储体系结构的典型的 RISC 处理器,对存储器的访问只能使用加载和存储指令实现。ARM 的加载/存储指令是可以实现字、半字、，无符/有符字节操作；批量加载/存储指令可实现一条指令加载/存储多个寄存器的内容，大大提高效率；SWP 指令是一条寄存器和存储器内容交换的指令，可用于信号量操作等。ARM 处理器是冯.诺依曼存储结构，程序空间、RAM 空间及 IO 映射空间统一编址，除对对 RAM 操作以外，对外围 IO、程序数据的访问均要通过加载/存储指令进行。

ARM 存储访问指令表

| 助记符 | 说明 | 操作 | 条件码位置 |
|---|---|---|---|
| LDR Rd,addressing | 加载字数据 | Rd←[addressing],addressing 索引 | LDR{cond} |
| LDRB Rd,addressing | 加载无符字节数据 | Rd←[addressing],addressing 索引 | LDR{cond}B |
| LDRT Rd，addressing | 以用户模式加载字数据 | Rd←[addressing],addressing 索引 | LDR{cond}T |
| LDRBT Rd，addressing | 以用户模式加载无符号字数据 | Rd←[addressing],addressing 索引 | LDR{cond}BT |
| LDRH Rd，addressing | 加载无符半字数据 | Rd←[addressing],addressing 索引 | LDR{cond}H |
| LDRSB Rd，addressing | 加载有符字节数据 | Rd←[addressing],addressing 索引 | LDR{cond}SB |
| LDRSH Rd，addressing | 加载有符半字数据 | Rd←[addressing],addressing 索引 | LDR{cond}SH |
| STR Rd，addressing | 存储字数据 | [addressing]←Rd,addressing 索引 | STR{cond} |
| STRB Rd，addressing | 存储字节数据 | [addressing]←Rd,addressing 索引 | STR{cond}B |
| STRT Rd，addressing | 以用户模式存储字数据 | [addressing]←Rd,addressing 索引 | STR{cond}T |
| SRTBT Rd，addressing | 以用户模式存储字节数据 | [addressing]←Rd,addressing 索引 | STR{cond}BT |
| STRH Rd，addressing | 存储半字数据 | [addressing]←Rd,addressing 索引 | STR{cond}H |
| LDM{mode} Rn{!},reglist | 批量(寄存器)加载 | reglist← [Rn…],Rn 回存等 | LDM{cond}{more} |
| STM{mode} Rn{!},rtglist | 批量(寄存器)存储 | [Rn…]← reglist,Rn 回存等 | STM{cond}{more} |
| SWP Rd,Rm,Rn | 寄存器和存储器字数据交换 | Rd←[Rd],[Rn]←[Rm](Rn≠Rd 或 Rm) | SWP{cond} |
| SWPB Rd,Rm,Rn | 寄存器和存储器字节数据交换 | Rd←[Rd],[Rn]←[Rm](Rn≠Rd 或 Rm) | SWP{cond}B |

### ◆ LDR 和 STR

加载/存储字和无符号字节指令.使用单一数据传送指令(STR 和 LDR)来装载和存储单一字节或字的数据从/到内存.LDR 指令用于从内存中读取数据放入寄存器中;STR 指令用于将寄存器中的数据保存到内存.指令格式如下:

LDR{cond}{T} Rd,<地址>;加载指定地址上的数据(字),放入 Rd 中

STR{cond}{T} Rd,<地址>;存储数据(字)到指定地址的存储单元,要存储的数据在 Rd 中

LDR{cond}B{T} Rd,<地址>;加载字节数据,放入 Rd 中,即 Rd 最低字节有效,高 24 位清零

STR{cond}B{T}　Rd,<地址>;存储字节数据,要存储的数据在 Rd,最低字节有效

其中,T 为可选后缀,若指令有 T,那么即使处理器是在特权模式下,存储系统也将访问看成是处理器是在用户模式下.T 在用户模式下无效,不能与前索引偏移一起使用 T.LDR/STR 指令寻址是非常灵活的,由两部分组成,一部分为一个基址寄存器,可以为任一个通用寄存器,另一部分为一个地址偏移量.地址偏移量有以下 3 种格式:

(1) **立即数**.立即数可以是一个无符号数值,这个数据可以加到基址寄存器,也可以从基址寄存器中减去这个数值.指令举例如下:

```
LDR    R1,[R0,#0x12] ;将 R0+0x12 地址处的数据读出,保存到 R1 中(R0 的值不变)
LDR    R1,[R0,#-0x12];将 R0-0x12 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)
```

(2)**寄存器**.寄存器中的数值可以加到基址寄存器,也可以从基址寄存器中减去这个数值.指令举例值.指令举例如下:

```
LDR    R1,[R0,R2]    ;将 R0+R2 地址的数据计读出,保存到 R1 中(R0 的值不变)
LDR    R1,[R0,-R2]   ;将 R0-R2 地址处的数据计读出,保存到 R1 中(R0 的值不变)
```

(3)**寄存器及移位常数**.寄存器移位后的值可以加到基址寄存器,也可以从基址寄存器中减去这个数值.指令举例如下:

```
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  R1,[R0,-R2,LSL #2];将 R0-R2*4 地址处的数据计读出,保存到 R1 中(R0,R2 的值不变)
```

从寻址方式的地址计算方法分,加载/存储指令有以下 4 种形式:

(1)**零偏移**.Rn 的值作为传送数据的地址,即地址偏移量为 0.指令举例如下:

```
LDR  Rd,[Rn]
```

(2)**前索引偏移**.在数据传送之前,将偏移量加到 Rn 中,其结果作为传送数据的存储地址.若使用后缀“!”,则结果写回到 Rn 中,且 Rn 值不允许为 R15.指令举例如下:

```
LDR    Rd,[Rn,#0x04]!
LDR    Rd,[Rn,#-0x04]
```

(3)**程序相对偏移**.程序相对偏移是索引形式的另一个版本.汇编器由 PC 寄存器计算偏移量,并将 PC 寄存器作为 Rn 生成前索引指令.不能使用后缀“!”.指令举例如下:

```
LDR    Rd,label      ;label 为程序标号,label 必须是在当前指令的±4KB 范围内
```

(4) 后索引偏移.Rn 的值用做传送数据的存储地址.在数据传送后,将偏移量与 Rn

相加，结果写回到 Rn 中.Rn 不允许是 R15.指令举例如下：

```
LDR   Rd,[Rn],#0x04
```

地址对准--大多数情况下，必须保证用于 32 位传送的地址是 32 位对准的.

加载/存储字和无符号字节指令举例如下：

```
LDR     R2,[R5]          ;加载 R5 指定地址上的数据(字)，放入 R2 中
STR     R1,[R0,#0x04]    ;将 R1 的数据存储到 R0+0x04 存储单元，R0 值不变
LDRB    R3,[R2],#1       ;读取 R2 地址上的一字节数据，并保存到 R3 中，R2=R3+1
STRB    R6,[R7]          ;读R6的数据保存到R7指定的地址中，只存储一字节数据
```

加载/存储半字和带符号字节.这类 LDR/STR 指令可能加载带符字节\加载带符号半字、加载/存储无符号半字.偏移量格式、寻址方式与加载/存储字和无符号字节指令相同.指令格式如下：

```
LDR{cond}SB   Rd,<地址>   ;加载指定地址上的数据(带符号字节)，放入 Rd 中
LDR{cond}SH   Rd,<地址>   ;加载指定地址上的数据(带符号字节)，放入 Rd 中
LDR{cond}H    Rd,<地址>   ;加载半字数据，放入 Rd 中，即 Rd 最低 16 位有效，高 16 位清零
STR{cond}H    Rd,<地址>   ;存储半字数据，要存储的数据在 Rd，最低 16 位有效
```

说明：带符号位半字/字节加载是指带符号位加载扩展到 32 位；无符号位半字加载是指零扩展到 32 位.

地址对准--对半字传送的地址必须为偶数.非半字对准的半字加载将使 Rd 内容不可靠，非半字对准的半字存储将使指定地址的 2 字节存储内容不可靠.

加载/存储半字和带符号字节指令举例如下：

```
LDRSB   R1[R0,R3] ;将 R0+R3 地址上的字节数据读出到 R1，高 24 位用符号位扩展
LDRSH   R1,[R9]   ;将 R9 地址上的半字数据读出到 R1，高 16 位用符号位扩展
LDRH    R6,[R2],#2  ;将 R2 地址上的半字数据读出到 R6，高 16 位用零扩展，，R2=R2+1
SHRH    R1,[R0,#2]!;将 R1 的数据保存到 R2+2 地址中，只存储低 2 字节数据，R0=R0+2
```

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。

**变量的访问**

```
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]       ;保存变量值
                …
```

GPIO 设置

```
GPIO-BASE  EQU   0Xe0028000        ;定义 GPIO 寄存器的基地址
                …
                LDR   R0,=GPIO-BASE
                LDR   R1,=0x00FFFF00  ;装载 32 位立即数,即设置值
                STR   R1,[R0,#0x0C]   ;IODIR=0x00FFFF00, IODIR 的地址为 0xE002800C
                MOV   R1,#0x00F00000
                STR   R1,[R0,#0x04]   ;IOSET=0x00F00000,IOSET 的地址为 0xE0028004
                …
```

程序散转

```
                …
                MOV       R2,R2,LSL #2   ;功能号乘上 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
                …
```

## ◆ LDM 和 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{!},reglist{^}

STM{cond}<模式> Rn{!},reglist{^}

LDM /STM 的主要用途是现场保护、数据复制、参数传送等。其模式有 8 种,如下:(前面 4 种用于数据块的传输,后面 4 种是堆栈操作)

(1) IA：每次传送后地址加 4

(2) IB：每次传送前地址加 4

(3) DA：每次传送后地址减 4

(4) DB：每次传送前地址减 4

(5) FD：满递减堆栈

(6) ED：空递增堆栈

(7) FA：满递增堆栈

(8) EA：空递增堆栈

其中,寄存器Rn为基址寄存器,装有传送数据的初始地址,Rn不允许为R15;后缀“!”表示最后的地址写回到Rn中;寄存器列表reglist可包含多于一个寄存器或寄存器范围,使用“,”分开,如{R1,R2,R6-R9},寄存器排列由小到大排列;“^”后缀不允许在用户模式呈系统模式下使用，若在 LDM 指令用寄存器列表中包含有 PC 时使用，那么除了正常的多寄存器传送外，将 SPSR 拷贝到 CPSR 中，这可用于异常处理返回；使用“^”后缀进行数据传送且寄存器列表不包含 PC 时，加载/存储的是用户模式的寄存器，而不是当前模式的寄存器。

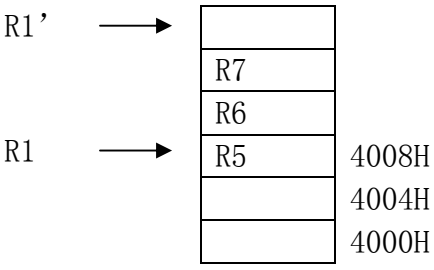
地址对准－－这些指令忽略地址的位[1：0]

批量加载/存储指令举例如下：

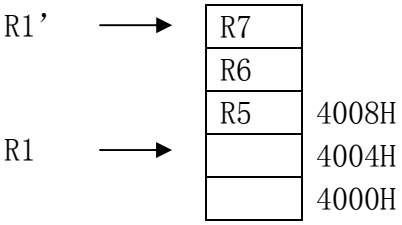
```
LDMIA   R0!,{R3-R9}     ;加载 R0 指向的地址上的多字数据,保存到 R3～R9 中,R0 值更新
STMIA   R1!,{R3-R9}     ;将 R3～R9 的数据存储到 R1 指向的地址上,R1 值更新
STMFD   SP!,{R0-R7,LR}  ;现场保存,将 R0～R7、LR 入栈
LDMFD   SP!,{R0-R7,PC}^ ;恢复现场,异常处理返回
```

在进行数据复制时,先设置好源数据指针,然后使用块拷贝寻址指令 LDMIA/STMIA、LDMIB/STMIB、LDMDA/STMDA、LDMDB/STMDB 进行读取和存储。而进行堆栈操作时，则要先设置堆栈指针，一般使用 SP 然后使用堆栈寻址指令 STMFD/LDMFD、STMED。LDMED、STMFA/LDMFA、STMEA/LDMEA 实现堆栈操作。

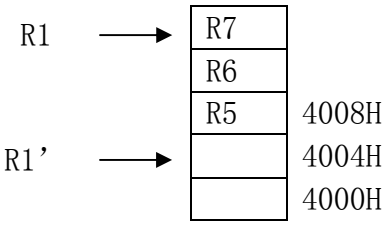
多寄存器传送指令示意图如图所示,其中R1为指令执行前的基址寄存器,R1则为指令执行完后的基址寄存器.



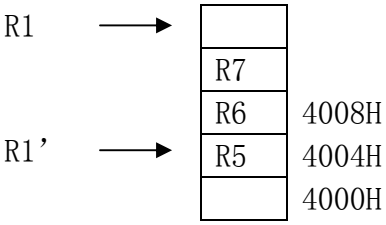

a）指令 STMIA R1!,{R5-R7}




b）指令 STMIB R1!,{R5-R7}




c）指令 STMDA R1!, {R5-R7}




d）指令 STMDB R1!,{R5-R7}

数据是存储在基址寄存器的地址之上还是之下,地址是在存储第一个值之前还是之后增加还是减少.

多寄存器传送指令映射

| | | 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 |

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
```

使用 LDM/STM 进行现场寄存器保护,常在子程序中或异常处理使用:

```
SENDBYTE
    STMFD   SP!,{R0-R7,LR}    ;寄存器入堆
     …
    BL      DELAY             ;调用 DELAY 子程序
     …
    LDMFD   SP!,{R0-R7,PC}    ;恢复寄存器,并返回
```

## ◆ SWP

寄存器和存储器交换指令.SWP 指令用于将一个内存单元(该单元地址放在寄存器 Rn 中)的内容读取到一个寄存器 Rd 中,同时将另一个寄存器 Rm 的内容写入到该内存单元中.使用 SWP 可实现信号量操作.

指令格式如下:

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

其中,B 为可选后缀,若有 B,则交换字节,否则交换 32 位字:Rd 为数据从存储器加载到的寄存器;Rm 的数据用于存储到存储器中,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.

SWP 指令举例如下:

```
SWP     R1,R1,[R0]      ;将 R1 的内容与 R0 指向的存储单元的内容进行交换
SWP     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的操作:

```
12C_SEM     EQU     0x40003000
              …
12C_SEM_WAIT
            MOV     R0,#0
            LDR     R0,=12C_SEM
            SWP     R1,R1,[R0]    ;取出信号量,并设置其为 0
            CMP     R1,#0         ;判断是否有信号
            BEQ     12C_SEM_WAIT  ;若没有信号,则等待
```

## ARM 数据处理指令

数据处理指令大致可分为 3 类;数据传送指令(如 MOV、MVN),算术逻辑运算指令(如 ADD,SUM,AND),比较指令(如 CMP,TST).数据处理指令只能对寄存器的内容进行操作.所有 ARM 数据处理指令均可选择使用 S 后缀,以影响状态标志.比较指令 CMP,CMN,TST 和 TEQ 不需要后缀 S,它们会直接影响状态标志.

ARM 数据处理指令

| 助记符号 | 说明 | 操作 | 条件码位置 |
|---|---|---|---|
| MOV Rd ,operand2 | 数据转送 | Rd←operand2 | MOV {cond}{S} |
| MVN Rd ,operand2 | 数据非转送 | Rd←(operand2) | MVN {cond}{S} |
| ADD Rd,Rn operand2 | 加法运算指令 | Rd←Rn+operand2 | ADD {cond}{S} |
| SUB Rd,Rn operand2 | 减法运算指令 | Rd←Rn-operand2 | SUB {cond}{S} |
| RSB Rd,Rn operand2 | 逆向减法指令 | Rd←operand2-Rn | RSB {cond}{S} |
| ADC Rd,Rn operand2 | 带进位加法 | Rd←Rn+operand2+carry | ADC {cond}{S} |
| SBC Rd,Rn operand2 | 带进位减法指令 | Rd←Rn-operand2-(NOT)Carry | SBC {cond}{S} |
| RSC Rd,Rn operand2 | 带进位逆向减法指令 | Rd←operand2-Rn-(NOT)Carry | RSC {cond}{S} |
| AND Rd,Rn operand2 | 逻辑与操作指令 | Rd←Rn&operand2 | AND {cond}{S} |
| ORR Rd,Rn operand2 | 逻辑或操作指令 | Rd←Rn\|operand2 | ORR {cond}{S} |
| EOR Rd,Rn operand2 | 逻辑异或操作指令 | Rd←Rn ^ operand2 | EOR {cond}{S} |
| BIC Rd,Rn operand2 | 位清除指令 | Rd←Rn&(～operand2) | BIC {cond}{S} |
| CMP Rn,operand2 | 比较指令 | 标志 N、Z、C、V←Rn-operand2 | CMP {cond} |
| CMN Rn,operand2 | 负数比较指令 | 标志 N、Z、C、V←Rn＋operand2 | CMN {cond} |
| TST Rn,operand2 | 位测试指令 | 标志 N、Z、C、V←Rn＆operand2 | TST {cond} |
| TEQ Rn,operand2 | 相等测试指令 | 标志 N、Z、C、V←Rn ^ operand2 | TEQ {cond} |

## 数据传送指令

### ◆ MOV

数据传送指令.将 8 位图立即数或寄存器(operant2)传送到目标寄存器 Rd,可用于移位运算等操作.指令格式如下:

MOV{cond}{S}　Rd,operand2

MOV 指令举例如下:

```
MOV     R1#0x10          ;R1=0x10
MOV     R0,R1            ;R0=R1
MOVS    R3,R1,LSL #2      ;R3=R1＜＜2,并影响标志位
MOV     PC,LR            ;PC=LR ,子程序返回
```

### ◆ MVN

数据非传送指令.将 8 位图立即数或寄存器(operand2)按位取反后传送到目标寄存器(Rd),因为其具有取反功能,所以可以装载范围更广的立即数.指令格式如下:

MVN{cond}{S}　Rd,operand2

MVN 指令举例如下:

```
MVN       R1,#0xFF     ;R1=0xFFFFFF00
MVN       R1,R2        ;将 R2 取反,结果存到 R1
```

## 算术逻辑运算指令

### ◆ ADD

加法运算指令.将 operand2 数据与 Rn 的值相加,结果保存到 Rd 寄存器.指令格式如下:

ADD{cond}{S}　Rd,Rn,operand2

ADD 指令举例如下:

```
ADDS      R1,R1,#1         ;R1=R1+1
ADD       R1,R1,R2         ;R1=R1+R2
ADDS      R3,R1,R2,LSL #2  ;R3=R1+R2＜＜2
```

### ◆ SUB

减法运算指令.用寄存器 Rn 减去 operand2.结果保存到 Rd 中.指令格式如下:

SUB{cond}{S} Rd,Rn,operand2

SUB 指令举例如下

```
SUBS    R0,R0,#1            ;R0=R0-1
SUBS    R2,R1,R2            ;R2=R1-R2
SUB     R6,R7,#0x10         ;R6=R7-0x10
```

### ◆ RSB

逆向减法指令.用寄存器 operand2 减法 Rn,结果保存到 Rd 中.指令格式如下:

RSB{cond}{S} Rd,Rn,operand2

SUB 指令举例如下

```
RSB   R3,R1,#0xFF00     ;R3=0xFF00-R1
RSBS  R1,R2,R2,LSL #2    ;R1=R2＜＜2-R2=R2×3
RSB   R0,R1,#0          ;R0=-R1
```

### ◆ ADC

带进位加法指令.将 operand2 的数据与 Rn 的值相加,再加上 CPSR 中的 C 条件标志位.结果保存到 Rd 寄存器.指令格式如下;

ADC{cond}{S} Rd,Rn,operand2

ADC 指令举例如下:

```
ADDS    R0,R0,R2
ADC     R1,R1,R3    ;使用 ADC 实现 64 位加法,(R1、R0)=(R1、R0)+（R3、R2）
```

### ◆ SBC

带进位减法指令。用寄存器 Rn 减去 operand2，再减去 CPSR 中的 C 条件标志位的非（即若 C 标志清零，则结果减去 1），结果保存到 Rd 中。指令格式如下：

SCB{cond}{S}Rd,Rn,operand2

SBC 指令举例如下

```
SUBS    R0，R0，R2
SBC     R1，R1，R3    ;使用 SBC 实现 64 位减法,(R1,R0)-(R3,R2)
```

### ◆ RSC

带进位逆向减法指令.用寄存器 operand2 减去 Rn,再减去 CPSR 中的 C 条件标志位,结果保存到 Rd 中.指令格式如下:

RSC{cond}{S} Rd,Rn,operand2

RSC 指令举例如下:

```
RSBS    R2,R0,#0
RSC     R3,R1,#0    ;使用 RSC 指令实现求 64 位数值的负数
```

### ◆ AND

逻辑与操作指令.将 operand2 值与寄存器 Rn 的值按位作逻辑与操作,结果保存到 Rd 中.指令格式如下:

AND{cond}{S} Rd,Rn,operand2

AND 指令举例如下:

```
ANDS    R0,R0,#x01  ;R0=R0&0x01,取出最低位数据
AND     R2,R1,R3    ;R2=R1&R3
```

### ◆ ORR

逻辑或操作指令.将operand2的值与寄存器Rn的值按位作逻辑或操作,结果保存到 Rd 中.指令格式如下:

ORR{cond}{S} Rd,Rn,operand2

ORR 指令举例如下:

```
ORR     R0,R0,#x0F      ;将 R0 的低 4 位置 1
MOV     R1,R2,LSR #4
ORR     R3,R1,R3,LSL #8 ;使用 ORR 指令将近 R2 的高 8 位数据移入到 R3 低 8 位中
```

### ◆ EOR

逻辑异或操作指令.将operand2的值与寄存器Rn的值按位作逻辑异或操作,结果保存到 Rd 中.指令格式如下:

EOR{cond}{S}Rd,Rn,operand2

EOR 指令举例如下:

```
EOR     R1,R1,#0x0F  ;将 R1 的低 4 位取反
EOR     R2,R1,R0     ;R2=R1^R0
EORS    R0,R5,#0x01 ;将 R5 和 0x01 进行逻辑异或,结果保存到 R0,并影响标志位
```

### ◆ BIC

位清除指令.将寄存器Rn的值与operand2的值的反码按位作逻辑与操作,结果保存到 Rd 中.指令格式如下:

```
BIC{cond} {S}Rd,Rn,operand2
```

BIC 指令举例如下:

```
BIC     R1,R1,#0x0F    ;将 R1 的低 4 位清零,其它位不变
BIC     R1,R2,R3       ;将拭的反码和 R2 相逻辑与,结果保存到 R1
```

## 比较指令

### ◆ CMP

比较指令.指令使用寄存器 Rn 的值减去 operand2 的值,根据操作的结果理新 CPSR 中的相应条件标志位,以便后面的指令根据相应的条件标志来判断是否执行.指令格式如下:

```
CMP{cond}    Rn,operand2
```

CMP 指令举例如下:

```
CMP   R1,#10        ;R1 与 10 比较,设置相关标志位
CMP   R1,R2         ;R1 与 R2 比较,设置相关标志位
```

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操作.

### ◆ CMN

负数比较指令.指令使用寄存器 Rn 与值加上 operand2 的值,根据操作的结果理新 CPSR 中的相应条件标志位,以便后面的指令根据相应的条件标志来判断是否执行,指令格式如下:

```
CMN{cond}    Rn,operand2
```

```
CMN     R0,#1       ;R0+1,判断 R0 是否为 1 的补码,若是 Z 置位
```

CMN 指令与 ADDS 指令的区别在于 CMN 指令不保存运算结果.CMN 指令可用于负数比较,比如 CMNR0,#1 指令则表示 R0 与-1 比较,若 R0 为-(即 1 的补码),则 Z 置位,否则 Z 复位.

### ◆ TST

位测试指令.指令将寄存器Rn的值与operand2的值按位作逻辑与操作,根据操作的结果理新 CPSR 中相应的条件标志位,以便后面指令根据相应的条件标志来判断是否执行.指令格式如下:

```
TST{cond}    Rn,operand2
```

TST 指令举例如下:

```
TST    R0,#0x01    ;判断 R0 的最低位是否为 0
TST    R1,#0x0F    ;判断 R1 的低 4 位是否为 0
```

TST 指令与 ANDS 指令的区别在于 TST4 指令不保存运算结果.TST 指令通常于 EQ,NE 条件码配合使用,当所有测试位均为 0 时,EQ 有效,而只要有一个测试为不为 0,则 NE 有效.

### ◆ TEQ

相等测试指令.指令寄存器Rn的值与operand2的值按位作逻辑异或操作,根据操作的结果理新 CPSR 中相应条件标志位,以便后面的指令根据相应的条件标志来判断是否执行.指令格式如下:

```
TEQ{cond}    Rn,operand2
```

TEQ 指令举例如下:

```
TEQ      R0,R1      ;比较 R0 与 R1 是否相等(不影响 V 位和 C 位)
```

TST 指令与EORS 指令的区别在于TST 指令不保存运算结果.使用 TEQ 进行相等测试,常与 EQNE 条件码配合使用,当两个数据相等时,EQ 有效,否则 NE 有效.

## 乘法指令

ARM7TDMI(-S)具有 32×32 乘法指令,32×32 乘加指令,32×32 结果为 64 位的乘/乘法指令.

ARM 乘法指令

| 助记符 | 说明 | 操作 | 条件码位置 |
|---|---|---|---|
| MUL Rd,Rm,Rs | 32 位乘法指令 | Rd←Rm*Rs (Rd≠Rm) | MUL{cond}{S} |
| MLA Rd,Rm,Rs,Rn | 32 位乘加指令 | Rd←Rm*Rs+Rn (Rd≠Rm) | MLA{cond}{S} |
| UMULL RdLo,RdHi,Rm,Rs | 64 位无符号乘法指令 | (RdLo,RdHi)←Rm*Rs | UMULL{cond}{S} |
| UMLAL RdLo,RdHi,Rm,Rs | 64 位无符号乘加指令 | (RdLo,RdHi)←Rm*Rs+(RdLo,RdHi) | UMLAL{cond}{S} |
| SMULL RdLo,RdHi,Rm,Rs | 64 位有符号乘法指令 | (RdLo,RdHi)←Rm*Rs | SMULL{cond}{S} |
| SMLAL RdLo,RdHi,Rm,Rs | 64 位有符号乘加指令 | (RdLo,RdHi)←Rm*Rs+(RdLo,RdHi) | SMLAL{cond}{S} |

### ◆ MUL

32 位乘法指令.指令将 Rm 和 Rs 中的值相乘,结果的低 32 位保存到 Rd 中.指令格式如下:

**MUL{cond}{S} Rd,Rm,Rs**

MUL 指令举例如下:

```
MUL   R1,R2,R3      ;R1=R2×R3
MULS  R0,R3,R7      ;R0=R3×R7,同时设置 CPSR 中的 N 位和 Z 位
```

### ◆ MLA

32 位乘加指令.指令将 Rm 和 Rs 中的值相乘,再将乘积加上第 3 个操作数,结果的低 32 位保存到 Rd 中.指令格式如下:

**MLA{cond}{S} Rd,Rm,Rs,Rn**

MLA 指令举例如下:

```
MLA   R1,R2,R3,R0   ;R1=R2×R3+10
```

### ◆ UMULL

64 位无符号乘法指令.指令将 Rm 和 Rs 中的值作无符号数相乘,结果的低 32 位保存到 RsLo 中,而高 32 位保存到 RdHi 中,.指令格式如下:

**UMULL{cond}{S} RdLo,RdHi,Rm,Rs**

UMULL 指令举例如下:

```
UMULL   R0,R1,R5,R8      ;(R1、R0)=R5×R8
```

### ◆ UMLAL

64 位无符号乘加指令.指令将 Rm 和 Rs 中的值作无符号数相乘,64 位乘积与 RdHi,RdLo 相加,结果的低 32 位保存到 RdLo 中,而高 32 位保存到 RdHi 中.指令格式如下:

```
UMLAL{cond}{S}   RdLo,RdHi,Rm,Rs
```

UMLAL 指令举例如下:

```
UMLAL     R0,R1,R5,R8      ;(R1,R0)=R5×R8+(R1,R0)
```

### ◆ SMULL

64 位有符号乘法指令.指令将 Rm 和 Rs 中的值作有符号数相乘,结果的低 32 位保存到 RdLo 中,而高 32 位保存到 RdHi 中.指令格式如下:

```
SMULL{cond}{S}   RdLo,RdHi,Rm,Rs
```

SMULL 指令举例如下:

```
SMULL    R2,R3,R7,R6      ;(R3,R2)=R7×R6
```

### ◆ SMLAL

64 位有符号乘加指令.指令将 Rm 和 Rs 中的值作有符号数相乘,64 位乘积与 RdHi,RdLo,相加,结果的低 32 位保存到 RdLo 中,而高 32 位保存到 RdHi 中.指令格式如下:

```
SMLAL{cond}{S}   RdLo,RdHi,Rm,Rs
```

SMLAL 指令举例如下:

```
SMLAL   R2,R3,R7,R6     ;(R3,R2)=R7×R6+(R3,R2)
```

## ARM 跳转指令

在 ARM 中有两种方式可以实现程序的跳转,一种是使用跳转指令直接跳转,另一种则是直接向 PC 寄存器赋值实现跳转.跳转指令有跳转指令 B,带链接的跳转指令 BL 带状态切换的跳转指令 BX.

跳转指令

| 助记符 | 说明 | 操作 | 条件码位置 |
|---|---|---|---|
| B label | 跳转指令 | Pc←label | B{cond} |
| BL label | 带链接的跳转指令 | LR←PC-4, PC←label | BL{cond} |
| BX Rm | 带状态切换的跳转指令 | PC←label,切换处理状态 | BX{cond} |

### ◆ B

跳转指令.跳转到指定的地址执行程序.指令格式如下;

B{cond} label

跳转指令 B 举例如下:

```
B    WAITA      ;跳转到 WAITA 标号处
B    0x1234     ;跳转到绝对地址 0x1234 处
```

跳转到指令 B 限制在当前指令的±32Mb 的范围内.

### ◆ BL

带链接的跳转指令.指令将下一条指令的地址拷贝到 R14(即 LR)链接寄存器中,然后跳转到指定地址运行程序.指令格式如下:

BL{cond} label

带链接的跳转指令 BL 举例如下:

```
BL     DELAY
```

跳转指令 B 限制在当前指令的±32MB 的范围内.BL 指令用于子程序调用

### ◆ BX

带状态切换的跳转指令.跳转到 Rm 指定的地址执行程序,若 Rm 的位[0]为 1,则跳转

时自动将 CPSR 中的标志 T 置位,即把目标地址的代码解释为 Thumb 代码;若 Rm 的位[0]为 0,则跳转时自动将 CPSR 中的标志 T 复位,即把目标地址的代码解释为 ARM 代码.指令格式如下:

BX{cond}　Rm

带状态切换的跳转指令 BX 举例如下

```
ADRL  R0,ThumbFun+1
BX    R0              ;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
```

## ARM 协处理器指令

ARM 支持协处理器操作,协处理器的控制要通过协处理器命令实现.

ARM 协处理器指令

| 助记符 | 说明 | 操作 | 条件码位置 |
|---|---|---|---|
| CDP coproc,opcode1,CRd,CRn,CRm{,opcode2} | 协处理器数据操作指令 | 取决于协处理器 | CDP{cond} |
| LDC{L} coproc,CRd〈地址〉 | 协处理器数据读取指令 | 取决于协处理器 | LDC{cond}{L} |
| STC{L} coproc,CRd,〈地址〉 | 协处理器数据写入指令 | 取决于协处理器 | STC{cond}{L} |
| MCR coproc, opcode1,Rd,CRn,{,opcode2} | ARM 寄存器到协处理器寄存器的数据传送指令 | 取决于协处理器 | MCR{cond} |
| MRC coproc, opcode1,Rd,CRn,{,opcode2} | 协处理器寄存器到 ARM 寄存器到数据传送指令 | 取决于协处理器 | MCR{cond} |

### ◆ CDP

协处理器数据操作指令.ARM 处理器通过 CDP 指令通知 ARM 协处理器执行特定的操作.该操作由协处理器完成,即对命令的参数的解释与协处理器有关,指令的使用取决于协处理器.若协处理器不能成功地执行该操作,将产生未定义指令异常中断.指令格式如下:

CDP{cond}　coproc,opcode1,CRd,CRn,CRm{,opcode2}

其中: coproc　指令操作的协处理器名.标准名为 pn,n 为 0~15.

opcode1 协处理器的特定操作码

CRd　作为目标寄存器的协处理器寄存器.

CRN　存放第 1 个操作数的协处理器寄存器.

CRm 存放第 2 个操作数的协处理器寄存器.

Opcode2 可选的协处理器特定操作码.

CDP 指令举例如下:

```
CDP   p7,0,c0,c2,c3,0      ;协处理器 7 操作,操作码为 0,可选操作码为 0
CDP   p6,1,c3,c4,c5        ;协处理器操作,操作码为 1
```

## ◆ LDC

协处理器数据读取指令.LDC 指令从某一连续的内存单元将数据读取到协处理器的寄存器中.协处理器数据的数据的传送,由协处理器来控传送的字数.若协处理器不能成功地执行该操作,将产生未定义指令异常中断.指令格式如下;

**LDC{cond}{L} coproc,CRd,<地址>**

其中: L 可选后缀,指明是长整数传送.

coproc 指令操作的协处理器名.标准名为 pn,n 为 0~15

CRd 作为目标寄存的协处理器寄存器.

<地址> 指定的内存地址

LDC 指令举例如下:

```
LDC   p5,c2,[R2,#4];读取 R2+4 指向的内存单元的数据,传送到协处理器 p5 的 c2 寄存器中
LDC   p6,c2,[R1]   ;读取是指向的内存单元的数据,传送到协处理器 p6 的 c2 寄存器中
```

## ◆ STC

协处理器数据写入指令.STC 指令将协处理器的寄存器数据写入到某一连续的内存单元中.进行协处理器数据的数据传送,由协处理器来控制传送的字数.若协处理器不能成功地执行该操作,将产生未定义指令异常中断.指令格式如下;

**STC{cond}{L} coproc,CRd,<地址>**

其中:

L 可选后缀,指明是长整数传送.

coproc 指令操作的协处理器名.标准名为 pn,n 为 0~15

CRd 作为目标寄存的协处理器寄存器.

〈地址〉 指定的内存地址

STC 指令举例如下：

```
STC  p5,c1,[R0]
STC  p5,c1,[Ro,#-0x04]
```

## ◆ MCR

ARM寄存器到协处理器寄存器的数据传送指令.MCR指令将ARM处理器的寄存器中的数据传送到协处理器的寄存器中.若协处理器不能成功地执行该操作,将产生未定义指令异常中断.指令格式如下;

MCR{cond} coproc,opcode1,Rd,CRn,CRm{,opcode2}

其中 coproc 指令操作的协处理器名.标准名为 pn,n 为 0~15.

cpcode1 协处理器的特定操作码.

CRD 作为目标寄存器的协处理器寄存器.

CRn 存放第 1 个操作数的协处理器寄存器

CRm 存放第 2 个操作数的协处理器寄存器.

Opcode2 可选的协处理器特定操作码.

MCR 指令举例如下：

```
MCR   p6,2,R7,c1,c2,
MCR   P7,0,R1,c3,c2,1,
```

## ◆ MRC

协处理器寄存器到 ARM 寄存器到的数据传送指令.MRC 指令将协处理器寄存器中的数据传送到 ARM 处理器的寄存器中.若协处理器不能成功地执行该操作.将产生未定义异常中断.指令格式如下.

MRC {cond} coproc,opcode1,Rd,CRn,CRm{,opcode2}

其中 coproc 指令操作的协处理器名.标准名为 pn,n,为 0~15

opcode1 协处理器的特定操作码.

CRd 作为目标寄存器的协处理器寄存器.

CRn 存放第 1 个操作数的协处理器寄存器.

CRm 存放第 2 个操作数的协处理器寄存器.

opcode2 可选的协处理器特定操作码.

MRC 指令举例如下

```
MRC  p5,2,R2,c3,c2
MRC  p7,0,R0,c1,c2,1
```

## ARM 杂项指令

| 助记符 | 说明 | 操作 | 条件码位置 |
|---|---|---|---|
| SWI immed_24 | 软中断指令 | 产生软中断，处理器进入管理模式 | SWI{cond} |
| MRS Rd,psr | 读状态寄存器指令 | Rd←psr,psr 为 CPSR 或 SPSR | MRS{cond} |
| MSR psr_fields,Rd/#immed_8r | 写状态寄存器指令 | psr_fields←Rd/#immed_8r,psr 为 CPSR 或 SPSR | MSR{cond} |

### ◆ SWI

软中断指令.SWI 指令用于产生软中断,从而实现在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,在其它模式下也可使用 SWI 指令,处理同样地切换到管理模式.指令格式如下;

SWI{cond} immed_24

其中 immed_24 24 位立即数,值为 0~16777215 之间的整数

SWI 指令举例如下:

```
SWI  0          ;软中断,中断立即数为 0
SWI  0x123456   ;软中断,中断立即数为 0x123456
```

使用 SWI 指令时,通常使用以下两种方法进行传递参数,SWI 异常中断处理程序就可以提供相关的服务,这两种方法均是用户软件协定.SWI 异常中断处理程序要通过读取引起软中断的 SWI 指令,以取得 24 位立即数.

1.指令 24 位的立即数指定了用户请求的服务类型,参数通过用寄存器传递

```
MOV   R0,#34    ;设置了功能号为 34
SWI   12        ;调用 12 号软中断
```

2.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

过其它的通用寄存器传递.

```
MOV     R0,#12      ;调用 12 号软中断
MOV     R1,#34      ;设置子功能号为 34
SWI     0
```

在 SWI 异常中断处理程序中,取出 SWI 立即数的步骤为:首先确定引起软中断的 SWI 指令是 ARM 指令还时 Thumb 指令,这可通过对 SPSR 访问得到:然后要取得该 SWI 指令的地址,这可通过访问 LR 寄存器得到:接着读出指令,分解出立即数.

读出 SWI 立即数

```
T_bit   EQU     0x20
SWI_Hander
        STMFD   SP!,{R0_R3,R12,LR}   ;现场保护
        MRS     R0,SPSR              ;读取 SPSR
        STMFD   SP!,{R0}             ;保存 SPSR
        TST     R0,#T_bit            ;测试 T 标志位
        LDRNEH  R0,[LR,#-2]          ;若是 Thumb 指令,读取指令码(16 位)
        BICNE   R0,R0,#0xFF00        ;取得 Thumb 指令的 8 位立即数
        LDREQ   R0,[LR,#-4]          ;若是 ARM 指令,读取指令码(32 位)
        BICNQ   R0,R0,#0xFF00000     ;取得 ARM 指令的 24 位立即数
        …
        LDMFD   SP!,{R0-R3,R12,PC}^  ;SWI 异常中断返回
```

## ◆ MRS

读状态寄存器指令.在 ARM 处理器中,只有 MRS 指令可以状态寄存器 CPSR 或 SPSR 读出到通用寄存器中.指令格式如下;

```
MRS{cond}  Rd ,psr
```

其中;  Rd  目标寄存器.Rd 不允许为 R15.

psr  CPSR 或 SPSR

SWI 指令举例如下

```
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 R2 中
```

MRS 指令读取 CPSR,可用来判断 ALU 的状态标志,或 IRQ,FIQ 中断是否允许等;在异常处理程序中,读 SPSR 可知道进行异常前的处理器状态等.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.保存起来.

**使能 IRQ 中断**

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

**禁能 IRQ 中断**

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

## ◆ MSR

写状态寄存器指令.在 ARM 处理器中.只有 MSR 指令可以直接设置状态寄存器 CPSR 或 SPSR.指令格式如下

**MSR{cond}　　psr_fields,#immed_8r**

**MSR{cond}　　psr_fields,Rm**

其中:　　psr　　CPSR 或 SPSR

fields　指定传送的区域.Fields 可以是以下的一种或多种(字母必须为小写):

c 控制域屏蔽字节(psr[7…0])

x 扩展域屏蔽字节(psr[15…8])

s 状态域屏蔽字节(psr[23.…16])

f 标志域屏蔽字节(psr[31…24])

immed_8r 要传送到状态寄存器指定域的立即数,8 位.

Rm 要传送到状态寄存器指定域的数据的源寄存器.

MSR 指令举例如下

```
MSR  CPSR_c,#0xD3     ;CPSR[7…0]=0xD3,即切换到管理模式.
MSR  CPSR_cxsf,R3     ;CPSR=R3
```

只有在特权模式下才能修改状态寄存器.

程序中不能通过 MSR 指令直接修改 CPSR 中的 T 控制位来实现 ARM 状态/Thumb 状态的切换,必须使用 BX 指令完成处理器状态的切换(因为 BX 指令属转移指令,它会打断流水线状态,实现处理器状态切换).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,.

**堆栈指令实始化**

```
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,StackIrq
        …
```

## ARM 伪指令

ARM 伪指令不是 ARM 指令集中的指令,只是为了编程方便编译器定义了伪指令,使用时可以像其它 ARM 指令一样使用,但在编译时这些指令将被等效的 ARM 指令代替.ARM 伪指令有四条,分别为 ADR 伪指令,ADRL 伪指令,LDR 伪指令,NOP 伪指令.

## ◆ ADR

小范围的地址读取伪指令.ADR 指令将基于 PC 相对偏移的地址值读取到寄存器中.在汇编编译源程序时,ADR 伪指令被编译器替换成一条合适的指令.通常,编译器用一条 ADD 指令或 SUB 指令来实现该 ADR 伪指令的功能,若不能用一条指令实现,则产生错误,编译失败.

ADR 伪指令格式如下

**ADR{cond}　register,exper**

其中　　register　加载的目标寄存器

　　exper　　地址表达式.当地址值是非字地齐时,取值范围-255~255 字节之间;当地址是字对齐时,取值范围-1020~1020 字节之间.对于基于 PC 相对偏移的地址值时,给定范围是相对当前指令地址后两个字处(因为 ARM7TDMI 为三级流水线).

ADR 伪指令举例如下;

```
LOOP    MOV     R1,#0xF0
        …
        ADR     R2,LOOP     ;将 LOOP 的地址放入 R2
        ADR     R3,LOOP+4
```

可以用 ADR 加载地址,实现查表:

```
        …
      ADR   R0,DISP_TAB         ;加载转换表地址
      LDRB  R1,[R0,R2]          ;使用 R2 作为参数,进行查表
        …
DISP_TAB
        DCB   0Xc0,0xF9,0xA4,0xB0,0x99,0x92,0x82,0xF8,0x80,0x90
```

## ◆ ADRL

中等范围的地址读取伪指令.ADRL 指令将基于 PC 相对偏移的地址值或基于寄存器相对偏移的地址值读取到寄存器中，比 ADR 伪指令可以读取更大范围的地址。在汇编编

译源程序时，ADRL 伪指令被编译器替换成两个条合适的指令。若不能用两条指令实现 ADRL 伪指令功能，则产生错误，编译失败。ADRL 伪指令格式如下：

ADR{cond} register,exper

其中：register 加载的目标寄存器。

expr 地址表达式。当地址值是非字对齐时，取范围-64K~64K 字节之间；当地址值是字对齐时，取值范围-256K~256K 字节之间。

ADRL 伪指令举例如下；

```
ADRL    R0,DATA_BUF
…
ADRL   R1 DATA_BUF+80
…
DATA_BUF
SPACE     100              ;定义 100 字节缓冲区
```

可以且用 ADRL 加载地址,实现程序跳转,中等范围地址的加载

```
        …
      ADR      LR,RETURNI      ;设置返回地址
      ADRL     R1 Thumb_Sub+1;取得了 Thumb 子程序入口地址,且 R1 的 0 位置 1
      BX       R1              ;调用 Thumb 子程序,并切换处理器状态
RETURNI
        …
        CODE16
Thumb_Sub
      MOV      R1,#10
      …
```

## ◆ LDR

大范围的地址读取伪指令.LDR 伪指令用于加载 32 位的立即数或一个地址值到指定寄存器.在汇编编译源程序时,LDR 伪指令被编译器替换成一条合适的指令.若加载的常数未超出 MOV 或 MVN 的范围,则使用 MOV 或 MVN 指令代替该 LDR 伪指令,否则汇编器将常

量放入字池,并使用一条程序相对偏移的LDR指令从文字池读出常量.LDR伪指令格式如下;

LDR{cond}　register,=expr/label_expr

其中　register　加载的目标寄存器

expr　32 位立即数.

label_expr　基于 PC 的地址表达式或外部表达式.

LADR　伪指令举例如下.

```
LDR     R0,=0x123456        ;加载 32 位立即数 0x12345678
LDR     R0,=DATA_BUF+60     ;加载 DATA_BUF 地址+60
…
LTORG                       ;声明文字池
```

伪指令LDR常用于加载芯片外围功能部件的寄存器地址(32 位立即数),以实现各种控制操作

**加载 32 位立即数**

```
…
LDR    R0,=IOPIN            ;加载 GPIO 寄存器 IOPIN 的地址
LDR    R1,[R0]              ;读取 IOPIN 寄存器的值
…
LDR    R0,=IOSET
LDR    R1,=0x00500500
STR    R1,[R0]              ;IOSET=0x00500500
…
```

从 PC 到文字池的偏移量必须小于 4KB

与 ARM 指令的 LDR 相比,伪指令的 LDR 的参数有“=”号

## ◆ NOP

空操作伪指令.NOP 伪指令在汇编时将会被代替成 ARM 中的空操作,比如可能为MOV,R0,R0 指令等,NOP 伪指令格式如下

NOP

NOP 可用于延时操作.

**软件延时**

```
    …
DELAY1
        NOP
        NOP
        NOP
        SUBS   R1,R1,#1
        BNE    DELAY1
        …
```

# Thumb 指令集

Thumb 指令可以看作是 ARM 指令压缩形式的子集,是针对代码密度的问题而提出的,它具有 16 位的代码密度.Thumb 不是一个完整的体系结构,不能指望处理只执行 Thumb 指令而不支持 ARM 指令集.因此,Thumb 指令只需要支持通用功能,必要时可以借助于完善的 ARM 指令集,比如,所有异常自动进入 ARM 状态.

在编写 Thumb 指令时,先要使用伪指令 CODE16 声明,而且在 ARM 指令中要使用 BX 指令跳转到 Thumb 指令,以切换处理器状态.编写 ARM 指令时,则可使用伪指令 CODE32 声明.

## Thumb 指令集与 ARM 指令集的区别

Thumb 指令集没有协处理器指令,信号量指令以及访问 CPSR 或 SPSR 的指令,没有乘加指令及 64 位乘法指令等,且指令的第二操作数受到限制;除了跳转指令 B 有条件执行功能外,其它指令均为无条件执行;大多数 Thumb 数据处理指令采用 2 地址格式.Thumb 指令集与 ARM 指令的区别一般有如下几点:

- 跳转指令

程序相对转移,特别是条件跳转与 ARM 代码下的跳转相比,在范围上有更多的限制,转向子程序是无条件的转移.

- 数据处理指令

数据处理指令是对通用寄存器进行操作,在大多数情况下,操作的结果须放入其中一个操作数寄存器中,而不是第 3 个寄存器中.

数据处理操作比 ARM 状态的更少,访问寄存器 R8~R15 受到一定限制.

除 MOV 和 ADD 指令访问器 R8~R15 外,其它数据处理指令总是更新 CPSR 中的 ALU 状态标志.

访问寄存器 R8~R15 的 Thumb 数据处理指令不能更新 CPSR 中的 ALU 状态标志.

- 单寄存器加载和存储指令

在 Thumb 状态下,单寄存器加载和存储指令只能访问寄存器 R0~R7

- 批量寄存器加载和存储指令

LDM 和 STM 指令可以将任何范围为 R0~R7 的寄存器子集加载或存储.

PUSH 和 POP 指令使用堆栈指令 R13 作为基址实现满递减堆栈.除 R0~R7 外,PUSH 指令还可以存储链接寄存器 R14,并且 POP 指令可以加载程序指令 PC

## Thumb 存储器访问指令

Thumb 指令集的 LDM 和 SRM 指令可以将任何范围为 R0~R7 的寄存器子集加载或存储.批量寄存器加载和存储指令只有 LDMIA,STMIA 指令,即每次传送先加载/存储数据,然后地址加 4.对堆栈处理只能使用 PUSH 指令及 POP 指令.

Thumb 存储器访问指令

| 助记符 | 说明 | 操作 | 影响标志 |
|---|---|---|---|
| LDR Rd,[Rn,#immed_5×4] | 加载字数据 | Rd←[Rm,#immed_5×4],Rd,Rn 为 R0～R7 | 无 |
| LDRH Rd,[Rn,#immed_5×2] | 加载无符半字数据 | Rd←[Rm,#immed_5×2],Rd,Rn 为 R0～R7 | 无 |
| LDRB Rd,[Rn,#immed_5×1] | 加载无符字节数据 | Rd←[Rm,#immed_5×1],Rd,Rn 为 R0～R7 | 无 |
| STR Rd,[Rn,#immed_5×4] | 存储字数据 | Rn,#immed_5×4Rd←Rd,Rn 为 R0～R7 | 无 |
| STRH Rd,[Rn,#immed_5×2] | 存储无符半辽数据 | Rn,#immed_5×2]Rd←Rd,Rn 为 R0～R7 | 无 |
| STRB Rd,[Rn#immed_5×1] | 存储无符字节数据 | Rn,#immed_5×1]Rd←Rd,Rn 为 R0～R7 | 无 |
| LDR Rd,[Rn,Rm] | 加载字数据 | Rd←[Rn,Rm],Rd,Rn,Rm 为 R0～R7 | 无 |
| LDRH Rd,[Rn,Rm] | 加载无符半字数据 | Rd←[Rn,Rm],Rd,Rn,Rm 为 R0～R7 | 无 |
| LDRB Rd,[Rn,Rm] | 加载无符字节数据 | Rd←[Rn,Rm],Rd,Rn,Rm 为 R0～R7 | 无 |
| LDRSH Rd[Rn,Rm] | 加载有符半字数据 | Rd←[Rn,Rm],Rd,Rn,Rm 为 R0～R7 | 无 |
| LDRSB Rd[Rn,Rm] | 加载有符字节数据 | Rd←[Rn,Rm],Rd,Rn,Rm 为 R0～R7 | 无 |
| STR Rd,[Rn,Rm] | 存储字数据 | [Rn,Rm]←Rd,Rd,Rn,Rm 为 R0～R7 | 无 |
| STRH Rd,[Rn,Rm] | 存储无符半字数据 | [Rn,Rm]←Rd,Rd,Rn,Rm 为 R0～R7 | 无 |
| STRB Rd,[Rn,Rm] | 存储无符字节数据 | [Rn,Rm]←Rd,Rd,Rn,Rm 为 R0～R7 | 无 |
| LDR Rd,[PC,#immed_8×4] | 基于 PC 加载字数据 | Rd←{PC,#immed_8×4]Rd 为 R0～R7 | 无 |
| LDR Rd,label | 基于 PC 加载字数据 | Rd←[label],Rd 为 R0～R7 | 无 |
| LDR Rd,[SP,#immed_8×4] | 基于 SP 加载字数据 | Rd←{SP,#immed_8×4]Rd 为 R0～R7 | 无 |
| STR Rd,[SP,#immed_8×4] | 基于 SP 存储字数据 | {SP,#immed_8×4]←Rd,Rd 为 R0～R7 | 无 |
| LDMIA Rn{!}reglist | 批量(寄存器)加载 | regist←[Rn…],Rn 回存等(R0～R7) | 无 |
| STMIA Rn{!}reglist | 批量(寄存器)加载 | [Rn…]←reglist,Rn 回存等(R0～R7) | 无 |
| PUSH {reglist[,LR]} | 寄存器入栈指令 | [SP…]←reglist[,LR],SP 回存等(R0～R7,LR) | 无 |
| POP {reglist[,PC]} | 寄存器入栈指令 | reglist[,PC]←[SP…],SP 回存等(R0～R7,PC) | 无 |

## ◆ LDR 和 STR

立即数偏移的LDR和STR指令.存储器的地址以一个寄存器的立即数偏移指明.指令格式如下:

```
LDR   Rd,[Rn,#immed_5×4];加载指定地址上的数据(字),放入 Rd 中
STR   Rd,[Rn,#immed_5×4];存储数据(字)到指定地址的存储单元,要存储数据在 Rd 中
LDRH  Rd,[Rn,#immed_5×4];加载半字数据,放入 Rd 中,即 Rd 低 16 位有效,高 16 位清零
STRH  Rd,[Rn,#immed_5×4];存储半字数据,要存储的数据在 Rd,最低 16 位有效
LDRB  Rd,[Rn,#immed_5×4];加载字节数据,放入 Rd 中,即 Rd 最低字节有效,高 24 位清零
STRB  Rd,[Rn,#immed_5×4];存储字节数据,要存储的数据在 Rd,最低字节有效
```

其中 Rd 加载或存储的寄存器.必须为 R0~R7.

Rn 基址寄存器.必须为 R0~R7.

immed_5×N 偏移量.它是一个无符立即数表达式,其取值为(0~3)×N

立即数偏移的半字和字节加载是无符号的.数据加载到 Rd 的最低有效半字或字节,Rd 的其余位补 0.

地址对准一一字传送时,必须保证传送地址为 32 位对准.半字传送时,必须保证传送地址为 16 位对准

立即数偏移的 LDR 和 STR 指令举例如下;

```
LDR   R0,[R1,#0x4]
STR   R3,[R4]
LDRH  R5,[R0,#0x02]
STRH  R1,[R0,#0x08]
LDRB  R3,[R6,#20]
STRB  R1,[R0,#31]
```

寄存器偏移的 LDR 和 STR 指令.存储器的地址用一个寄存器的基于寄存器偏移来指明.指令格式如下,

```
LDR    Rd,[Rn,Rm]   ;加载一个字数据
STR    Rd,[Rn,Rm]   ;存储一个字数据
LDRH   Rd,[Rn,Rm]   ;加载一个无符半字数据
```

```
STRH    Rd,[Rn,Rm]      ;存储一个无符半字数据
LDRB    Rd,[Rn,Rm]      ;加载一个无符字节数据
STRB    Rd,[Rn,Rm]      ;存储一个无符字节数据
LDRSH   Rd,[Rn,Rm]      ;加载一个有符半字数据
LDRSB   Rd,[Rn,Rm]      ;存储一个有符半字数据
```

其中： Rd 加载或存储的寄存器.必须为 R0~R7

Rn 基址寄存器.必须为 R0~R7

Rm 内含偏移量的寄存器.必须为 R0~R7.

寄存器偏移的半字和字节加载可以是有符号或无符号的,数据加载到 Rd 的其余位拷贝符号位.

地址对准—字传送时,必须保证传送地址为 32 位对准.半字传送时,必须保证传送地址为 16 位对准.

寄存器偏移的 LDR 和 STR 指令举例如下;

```
LDR     R3,[R1,R0]
STR     R1,[R0,R2]
LDRH    R6,[R0,R1]
STRH    R0,[R4,R5]
LDRB    R2,[R5,R1]
STRB    R1,[R3,R2]
LDRSH   R7,[R6,R3]
LDRSB   R5,[R7,R2]
```

PC 或 SP 相对偏移的 LDR 和 STR 指令.用 PC 或 SP 寄存器中的值的立即数偏移来指明存储器的地址.指令格式如下.

```
LDR     Rd,[PC,#immed_8×4]
LDR     Rd,label
LDR     Rd,[SP,#immed_8×4]
STR     Rd,[SP,#immed_8×4]
```

其中： Rd 加载或存储的寄存器.必须为 R0~R7

immed_8×4] 偏移量.它是一个无符立即数表达式,其取值为(0~255)×4

label 程序相对偏移表达式.label 必须在当前指令之后 IK 字节范围内.

地址对准—地址必须是 4 的整数倍.

PC 或 SP 相对偏移的 LDR 和 STR 指令举例如下;

```
LDR   R0,[PC,#0x08]   ;读取 PC+0x08 地址上的字数据,保存到 R0 中
LDR   R7,LOCALDAT     ;读取 LOCALDAT 地址上的字数据,保存到 R7 中
LDR   R3,[SP,#1020]   ;读取 SP+1020 地址上的字数据,保存到 R3 中
STR   R2,[SP]         ;存储 R2 寄存器的数据到 SP 指向的存储单元(偏移量为 0)
```

## ◆ PUSH 和 POP

寄存器入栈及出栈指令.实现低寄存器和可选的 LR 寄存器入栈寄存器和可选的 PC 寄存器出栈操作,堆栈地址由 SP 寄存设置,堆栈是满递减堆栈.指令格式如下;

```
PUSH   {reglist[,LR]}
POP    {reglist[,PC]}
```

其中 reglist 入栈/出栈低寄存器列表,即 R0~R7

LR 入栈时的可选寄存器

PC 出栈时的可选寄存器

寄存器入栈及出栈指令举例如下;

```
PUSH   {R0-R7,LR}     ;将低寄存器 R0~R7 全部入栈,LR 也入栈
POP    {R0-R7,PC}     ;将堆栈中的数据弹出到低寄存器 R0~R7 及 PC 中
```

## ◆ LDMIA 和 STMIA

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.Thumb 指令集批量加载/存储指令为 LDMIA 和 STMIA,LDMIA 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 8 个低寄存器的任何子集.指令格式如下;

```
LDMIA   Rn!,reglist
STMIA   Rn!,reglist
```

其中 Rn 加载/存储的起始地址寄存器.Rn 必须为 R0~R7

Reglist 加载/存储的寄存器列表.寄存器必须为 R0~R7

LDMIA/STMIA 的主要用途是数据复制,参数传送等,进行数据传送时,每次传送后地址加 4.若 Rn 在寄存器列表中,对于 LDMIA 指令,Rn 的最终值是加载的值,而不是增加后的地址;对于 STMIA 指令,在 Rn 是寄存器列表中的最低数字的寄存器,则 Rn 存储的值为 Rn 在初值,其它情况不可预知.

批量加载/存储指令举例如下;

```
LDMIA    R0,{R2-R7}  ;加载 R0 指向的地址上的多字数据,保存到 R2~R7 中,R0 的值更新.
STMIA    R1!,{R2-R7};将 R2~R7 的数据存储到 R1 指向的地址上,R1 值更新.
```

## Thumb 数据处理指令

大多数 Thumb 处理指令采用 2 地址格式,数据处理操作比 ARM 状态的更少,访问寄存器 R8~R15 受到一定限制.

Thumb 数据处理指令

| 助记符 | 说明 | 操作 | 影响标志 |
|---|---|---|---|
| MOV Rd,#expr | 数据转送 | Rd←expr,Rd 为 R0～R7 | 影响 N,Z |
| MOV Rd,Rm | 数据转送 | Rd←Rm,Rd、Rm 均可为 R0～R15 | RdT 和 Rm 均为 R0～R7 时,影响 N,Z,清零 C,V |
| MVN Rd,Rm | 数据非传送指令 | Rd←（～Rm),Rd,Rm 均为 R0～R7 | 影响 N,Z |
| NEG Rd,Rm | 数据取负指令 | Rd←(-Rm),Rd,Rm 均为 R0～R7 | 影响 N,Z,C,V |
| ADD Rd.Rn,Rm | 加法运算指令 | Rd←Rn+Rm,Rd,Rn,Rm 均为 R0～R7 | 影响 N,Z,C,V |
| ADD Rd.Rn,#expr3 | 加法运算指令 | Rd←Rn+expr#,Rd,Rn 均为 R0～R7 | 影响 N,Z,C,V |
| ADD Rd,#expr8 | 加法运算指令 | Rd←Rd+expr8,Rd 为 R0～R7 | 影响 N,Z,C,V |
| ADD Rd,Rm | 加法运算指令 | Rd←Rd+Rm,Rd,Rm 均可为 R0～R15 | Rd 和 Rm 均为 R0～R7 时，影响 N,Z,C,V |
| ADD Rd,Rp#expr | SP/PC 加法运算指令 | Rd←SP+expr 或 PC+expr,Rd 为 R0～R7 | 无 |
| ADD SP,#expr | SP 加法运算指令 | SP←SP+expr | 无 |
| SUB Rd,Rn,Rm | 减法运算指令 | Rd←Rn-Rm,Rd、Rn、Rm 均为 R0～R7 | 影响 N,Z,C,V |
| SUB Rd,Rn#expr3 | 减法运算指令 | Rd←Rn-expr3，RdRn 均为 R0～R7 | 影响 N,Z,C,V |
| SUB Rd,#expr8 | 减法运算指令 | RD←Rd-expr8，Rd 为 R0～R7 | 影响 N,Z,C,V |
| SUB SP,#expr | SP 减法运算指令 | SP←SP-expr | 无 |
| ADC Rd,Rm | 带进位加法指令 | Rd←Rd+Rm+Carry，Rd、Rm 为 R0～R7 | 影响 N,Z,C,V |
| SBC Rd,Rm | 带位减法指令 | Rd←Rd-Rm-(NOT)Carry，Rd、Rm 为 R0～R7 | 影响 N,Z,C,V |
| MUL Rd,Rm | 乘法运算指令 | Rd←Rd*Rm,Rd、Rm 为 R0～R7 | 影响 N,Z, |
| AND Rd,Rm | 逻辑与操作指令 | Rd←Rd&Rm,Rd、Rm 为 R0～R7 | 影响 N,Z, |
| ORR Rd,Rm | 逻辑或操作指令 | Rd←Rd\|Rm,Rd、Rm 为 R0～R7 | 影响 N,Z, |
| EOR Rd,Rm | 逻辑异或操作指令 | Rd←Rd^Rm,Rd、Rm 为 R0～R7 | 影响 N,Z, |
| BIC Rd,Rm | 位清除指令 | Rd←Rd&(～Rm),Rd、Rm 为 R0～R7 | 影响 N,Z, |
| ASR Rd,Rs | 算术右移指令 | Rd←Rd 算术右移 Rs 位,Rd,Rs 为 R0～R7 | 影响 N,Z,C, |
| ASR Rd,Rm#expr | 算术右移指令 | Rd←Rm 算术右移 expr 位,Rd、Rm 为 R0～R7 | 影响 N,Z,C, |
| LSL Rd,Rs | 逻辑左移指令 | Rd←Rd<<Rs,Rd、Rs 为 R0～R7 | 影响 N,Z,C, |
| LSL Rd,Rm,#expr | 逻辑左移指令 | Rd←Rm<<expr,Rd、Rm 为 R0～R7 | 影响 N,Z,C, |
| LSR Rd,Rs | 逻辑右移指令 | Rd←Rd>>Rs,Rd、Rs 为 R0～R7 | 影响 N,Z,C, |
| LSR Rd,Rm,#expr | 逻辑右移指令 | Rd←Rm>>mexpr,Rd、Rm 为 R0～R7 | 影响 N,Z,C, |
| ROR Rd,Rs | 循环右移指令 | Rd←Rm 循环右移 Rs 位,Rd、Rs 为 R0～R7 | 影响 N,Z,C, |
| CMP Rn,Rm | 比较指令 | 状态标←Rn-Rm,Rn、Rm 为 R0～R15 | 影响 N,Z,C,V |
| CMP Rn,#expr | 比较指令 | 状态标←Rn-expr,Rn 为 R0～R7 | 影响 N,Z,C,V |
| CMN Rn,Rm | 负数比较指令] | 状态标←Rn+Rm,Rn、Rm 为 R0～R7 | 影响 N,Z,C,V |

| | | | |
|---|---|---|---|
| TST Rn,Rm | 位测试指令 | 状态标←Rn&Rm,Rn、Rm 为 R0～R7 | 影响 N,Z,C,V |

## 数据传送指令

### ◆ MOV

数据传送指令.将 8 位立即数或寄存器(operand2)传送到目标寄存器(Rd).指令格式如下;

```
MOV     Rd,#expr
MOV     Rd,Rm
```

其中 Rd 目标寄存器.MOV Rd#expr 时,必须在 R0～R7 之间

exper 8 位立即数,即 0~255

Rm 源寄存器.为 R0~R15

条件码标志:

MOV Rd,#expr 指令会更新 N 和 Z 标志,对标志 C 和 V 无影响.而 MOV,Rd,Rm 指令,若 Rd 或 Rm 是高寄存器(R8~R15),则标志不受影响,若 Rd 或 Rm 都是低寄存器(R0~R7),则会更新 N 和 Z,且清除标志 C 和 V.

MOV 指令举例如下

```
MOV     R1,#0x10      ;R1=0x10
MOV     R0,R8         ;R0=R8
MOV     PC,LR         ;PC=LR,子程序返回
```

### ◆ MVN

数据非传送指令.将寄存器 Rm 按位取反后传送到目标寄存器(Rd).指令格式如下:

```
MVN   Rd,Rm
```

其中: Rd 目标寄存器.必须在 R0~R7 之间

Rm 源寄存器.必须在 R0~R7 之间

条件码标志:

指令会更新 N 和 Z 标志,对标志 C 和 V 无影响.

MVN 指令举例如下

```
MVN     R1,R2         ;将 R2 取反结果存到 R1
```

◆ **NEG**

数据取负指令.将寄存器 Rm 乘以-1 后传送到目标寄存器(Rd).指令格式如下:

**NEG Rd,Rm**

其中: Rd 目标寄存器.必须在 R0~R7 之间

Rm 源寄存器.必须在 R0~R7 之间

条件码标志

指令会更新 N,Z,C,V 和标志.

NEG 指令举例如下

```
NEG   R1,R0,      ;R1=-R0
```

## 算术逻辑运算指令

◆ **ADD**

加法运算指令.将两个数据相加,结果保存到 Rd 寄存器.

**低寄存器的 ADD 指令**的指令格式如下

**ADD Rd,Rn,Rm**

**ADD Rd,Rn,#expr3**

**ADD Rd,#expr8**

其中 Rd 目标寄存器.必须在 R0~R7 之间.

Rn 第一个操作数寄存器.必须在 R0~R7 之间.

Rm 第二个操作数寄存器.必须在 R0~R7 之间.

expr3 3 位立即数,即 0~7

expr8 8 位立即数,即 0~255.

条件码标志:指令会更新 N、Z、C 和 V 标志.

**高或低寄存器的 ADD 指令**的指令格式如下:

**ADD Rd,Rm**

其中 Rd 目标寄存器,也是第一个操数寄存器.

Rm 第二个操作数寄存器

条件码标志:若 Rd 或 Rm 都是低寄存器(R0~R7),指令会更新 N、Z、C 和 V 标志.其它情况不影响条件码标志.

**PC 或 SP 相对偏移的 ADD 指令**指令格式如下:

**ADD Rd,Rp,#expr**

其中 Rd 目标寄存器.必须在 R0~R7 之间

Rp PC 或 SP,第一个操作数寄存器.

expr 立即数,在 0~1020 范围内.

条件码标志:不影响条件码标志.

**SP 操作的 ADD 指令**的指令格式如下

**ADD SP,#expr**

**ADD SP,SP,#expr**

其中 SP 目标寄存器,也是第一个操作数寄存器.

expr 立即数,在-508~+508 之间的 4 的整数倍的数

条件码标志:不影响条件码标志.

ADD 指令举例如下

```
ADD     R1,R1,R0        ;R1=R1+R0
ADD     R1,R1,#7        ;R1=R1+7
ADD     R3,#200         ;R3=R3+200
ADD     R3,R8           ;R3=R3+R8
ADD     R1,SP,#1000     ;R1=SP+1000
ADD     SP,SP,#-500  ;SP=SP-500
```

## ◆ SUB

减法运算指令.将两个数相减,结果保存到 Rd 中.

**低寄存器的 SUB 指令**的指令格式如下;

**SUB Rd,Rn,Rm**

**SUB Rd,Rn,#expr3**

**SUB Rd,#expr8**

其中 Rd 目标寄存器.必须在 R0~R7 之间

Rn 第一个操作数寄存器.必须在 R0~R7 之间

Rm 第一个操作数寄存器.必须在 R0~R7 之间

expr3 3 位立即数,即 0~7

expr8　8 位立即数. 即 0~255

条件码标志:指令会更新 N、Z、C 和 V 标志.

**SP 操作的 SUB 指令**的指令格式如下

**SUB　SP,#expr**

**SUB　SP,SP,#expr**

其中　SP　　目标寄存器,也是第一个操作数据寄存器.

expr　立即数,在-508~+508 之间的 4 的整数倍的数

条件码标志:不影响条件码标志

SUB 指令举例如下

```
SUB   R0,R2,R1,      ;R0=R2-R1
SUB   R2,R1,#1       ;R2=R1-1
SUB   R6,#250        ;R6=R6-250
SUB   SP,#380        ;SP=SP-380
```

### ◆ ADC

带进位加法指令. 将 Rm 的值与 Rd 的值相加,再加上 CPSR 中的 C 条件标志位,结果保存到 Rd 寄存器. 指令格式如下

**ADC　Rd,Rm**

其中　Rd　　目标寄存器,也是第一个操作数寄存器. 必须在 R0~R7 之间

Rm　　第二个操作数寄存器. 必须在 R0~R7 之间

条件码标志:指令会更新 N、Z、C 和 V 标志.

ADC 指令举例如下;

```
ADD   R0,R0,R2,
ADC   R1,R1,R3       ;使用 ADC 实现 64 位加法,(R1,R0)+(R3,R2)
```

### ◆ SBC

带进位减法指令. 用寄存器 Rd 减去 Rm,再减去 CPSR 中的 C 条件标志的非(即若 C 标志清零,则结果减去 1),结果保存到 Rd 中. 指令格式如下

**SBC　Rd,Rm**

其中　Rd　　目标寄存器,也是第一个操作数寄存器. 必须在 R0~R7 之间

Rm 第二个操作数寄存器.必须在 R0~R7 之间]

条件码标志:指令会更新 N、Z、C 和 V 标志

SBC 指令举例如下

```
SUB  R0,R0,R2
SUB  R1,R1,R3  ;使用 SBC 实现 64 位减法,(R1,R0)=(R1,R0)-(R3,R2)
```

### ◆ MUL

乘法运算指令.用寄存器 Rd 乘以 Rm,结果保存到 Rd 中.指令格式如下;

```
MUL  Rd,Rm
```

其中 Rd 目标寄存器,也是第一个操作数寄存器.必须在 R0~R7 之间

Rm 第二操作数寄存器.必须在 R0~R7 之间

条件码标志:指令会更新 N 和 Z 标志

MUL 指令举例如下

```
MUL R2,R0,R1    ;R2=R0*R1
```

### ◆ AND

逻辑与操作指令.将寄存器 Rd 的值与寄存器 Rm 值按位作逻辑与操作,结果保存到 Rd 中.指令格式如下

```
AND  Rd,Rm
```

其中 Rd 目标寄存器,也是第一个操作数寄存器.必须在 R0~R7 之间

Rm 第二个操作数寄存器.必须在 R0~R7 之间

条件码标志:指令会更新 N 和 Z 标志

AND 指令举例如下;

```
MOV  R1,#0x0F
AND  R0,R1      ;R0=R0 & R1
```

### ◆ ORR

逻辑或操作指令.将寄存器 Rd 与寄存器 Rn 的值按位作逻辑或操作,结果保存到 Rd 中.指令格式如下:

```
ORR  Rd,Rm
```

其中 Rd 目标寄存器,也是第一个操作数寄存器.必须在 R0~R7 之间

Rm 第二个操作数寄存器.必须在 R0~R7 之间

条件码标志:指令会更新 N 和 Z 标志

ORR 指令举例如下

```
MOV   R1,#0x03
ORR   R0,R1          ;R0=R0|R1
```

### ◆ EOR

逻辑异或操作指令.寄存器Rd的值与寄存器Rn的值按位作逻辑异或操作,结果保存到 Rd 中,指令格式如下;

**EOR Rd,Rm**

其中 Rd 目标寄存器,也是第一个操作数寄存器.必须在 R0~R7 之间

Rm 第二个操作数寄存器.必须在 R0~R7 之间

条件码标志:指令会更新 N 和 Z 标志

EOR 指令举例如下

```
MOV   R2,#0Xf0
EOR   R3,R2,          ;R3=R3^R2
```

### ◆ BIC

位清除指令.将寄存器Rd的值与寄存器Rm的值反码按位作逻辑与操作.结果保存到 Rd 中,指令格式如下

**BIC Rd,Rm**

其中 Rd 目标寄存器,也是第一个操作数寄存器.必须在 R0~R7 之间.

Rm 第二个操作数寄存器.必须在 R0~R7 之间

条件码标志:指令会更新 N 和 Z 标志

BIC 指令举例如下:

```
MOV   R1,#0x80
BIC   R3,R1          ;将 R1 的最高位清零,其它位不变
```

### ◆ ASR

算术右移指令.数据算术右移,将符号位拷贝到空位,移位结果保存到 Rd 中,指令格式如下;

ASR Rd,Rs

ASR Rd,Rm,#expr

其中 Rd 目标寄存器,也是第一个操作数寄存器.必须在 R0~R7 之间

Rs 寄存器控制移位中包含移位量的寄存器.必须在 R0~R7 之间

Rm 立即数移位的源寄存器.必须在 R0~R7 之间

expr 立即数移位量,值为 1~32

条件码标志:指令会更新 N、Z 和 C 标志(若移位量为零,则不影响 C 标志)

ASR 指令举例如下

```
ASR   R1,R2,
ASR   R3,R1,#2
```

若移位量为 32,则 Rd 清零,最后移出的位保留在标志 C 中,若移位量大于 32,则 Rd 和标志 C 均被清零;若移位量为 0,则不影响 C 标志

### ◆ LSL

逻辑左移指令.数据逻辑左移,空位清零,移位结果保存到 Rd 中,指令格式如下

LSL Rd,Rs

LSL Rd,Rm,#expr

其中 Rd 目标寄存器,也是第一个操作数寄存器.必须在 R0~R7 之间

Rs 寄存器控制移位中包含位量的寄存器.必须在 R0~R7 之间

Rm 立即数移位的源寄存器.必须在 R0~R7 之间

expr 立即数移位量,值为 1~31

条件码标志:指令会更新 N、Z 和 C 标志(若移位量为零,则不影响 C 标志)

LSL 指令举例如下

```
LSL   R6,R7
LSL   R1,R6,#2
```

若移位量为 32,则 Rd 清零,最后移出的位保留在标志 C 中;若移位量大于 32,则 Rd 和标志 C 均被清零;若移位量为 0,则不影响 C 标志

### ◆ LSR

逻辑左移指令.数据逻辑左移,空位清零,移位结果保存到 Rd 中.指令格式如下

LSR Rd,Rs

LSR Rd,Rm,#expr

其中 Rd 目标寄存器,也是第一个操作数寄存器.必须在 R0~R7 之间

Rs 寄存器控制移位中包含移位量的寄存器.必须在 R0~R7 之间

Rm 立即数移位的源寄存器.必须在 R0~R7 之间

expr 立即数移位量,值为 1~32

条件码标志:指令会更新 N、Z 和 C 标志(若移位量为零,则不影响 C 标志)

LSR 指令举例如下

```
LSR  R3,R0
LSR  R5,R2,#2
```

若移位量为 32,则 Rd 清零,最后移出的位保留在标志 C 中;若移位量大于 32,则 Rd 和标志 C 均被清零,若移位量为 0,则不影响 C 标志.

### ◆ ROR

循环右移指令.数据循环右移,寄存器右边移出的位循环移回到左边,移位结果保存到 Rd 中,指令格式如下

ROR Rd,Rs

其中 Rd 目标寄存器.也是第一个操作数寄存器.必须在 R0~R7 之间

Rs 寄存器控制移位中包含移位量的寄存器.必须在 R0~R7 之间

条件标志:指令会更新 N,Z,C 的标志(若移位量为零,则不影响 C 标志).

ROR 指令举例如下

```
ROR  R2,R3
```

## 比较指令

### ◆ CMP

比较指令.指令使用寄存器 Rn 的值减去第二个操作数的值,根据操作的结果理新 CPSR 中的相应条件标志位.指令格式如下

CMP Rn,Rm

CMP Rn,#expr

其中 Rn 第一个操作数寄存器.对 CMP,Rn#expr 指令,Rn 在 R0~R7 之间;对于 CMP,Rn,Rm 指令在 Rn 在 R0~R15 之间

Rm 第二个操作数寄存器.Rm 在 R0~R15 之间

expr 立即数,值为 0~255

条件码标志:指令会更新 N、Z、C 和 V 标志

CMP 指令举例如下

```
CMP  R1,#10      ;R1 与 10 比较,设置相关标志位
CMP  R1,R2       ;R1 与 R2 比较,设置相关标志位
```

### ◆ CMN

负数比较指令.指令使用寄存器 Rn 的值加上寄存器 Rm 的值,根据操作的结果理新 CPSR 中的相应条件标志.位指令格式如下

**CMN Rn,Rm**

其中 Rn 第一个操作数寄存器,必须在 R0~R7 之间

Rm 第二个操作数寄存器.必须在 R0~R7 之间

条件码标志:指令会更新 N、Z、C 和 V 标志.

CMN 指令举例如下

```
CMM   R0,R2       ;R0 与 R2 进行比较
```

### ◆ TST

位测试指令.指令将寄存器Rn的值与寄存器Rm的值按位作逻辑与操作.根据操作的结果理新 CPSR 相应条件标志位.指令格式如下.

**TST Rn,Rm**

其中 Rn 第一个操作数寄存器.必须在 R0~R7 之间

Rm 第二个操作数寄存器.必须在 R0~R7 之间

条件码标志:指令会更新 N、Z、C 和 V 标志

TST 指令举例如下

```
MOV  R0,#x01
TST  R1,R0,       ;判断 R1 的最低位是否为 0
```

## Thumb 跳转指令

| 助记符 | 说明 | 操作 | 条件码位置 |
|---|---|---|---|
| B label | 跳转指令 | PC←label | B{cond} |
| BL label | 带链接的跳转指令 | LR←PC←4,PC←label | 无 |
| BX Rm | 带状态切换的跳转指令 | PC←label 切换处理器状态 | 无 |

### ◆ B

跳转指令.跳转到指定的地址执行程序.这是 Thumb 指令集中的惟一的有条件执行指令.指令格式如下

B{cond} label

跳转指令 B 举例如下

```
B       WAITB
BEQ     LOOP1
```

若使用 cond 则 label 必须在当前指令的-252~+256 字节范围内;若指令是无条件的,则跳转指令 label 必须在当前指令的±2K 字节范围内

### ◆ BL

带链接的跳转指令.指令先将下一条指令的地址拷贝到 R14(即 LR)链接寄存器中,然后跳转到指定地址运行程序.指令格式如下:

BL label

带链接的跳转指令 BL 举例如下

```
BL    DELAYI
```

机器级转指令 BL 限制在当前指令的±4Mb 的范围内.(必要时,ARM 链接器插入代码以允许更长的转移.)

### ◆ BX

带状态切换的跳转指令.跳转到 Rm 指定的地址执行程序.若 Rm 的位[0]为 0,则 Rm 的位于也必须为 0,跳转时自动将 CPSR 中的标志 T 复位,即把目标地址的代码解释为 ARM 代码.指令格式

BX Rm

带状态切换的跳转指令 BX 举例如下.

```
ADR     R0,ArmFun
BX      R0          ;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.
```

## Thumb 杂项指令

### ◆ SWI

软中断指令.SWI 指令用于产生软中断,从而实现在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.在其它模式下也可使用 SWI 指令,处理器同样地切换到管理模式.

指令格式如下.

SWI immed_8

其中 immed_8 8 位立即数,值为 0~255 之间的整数.

SWI 指令举例如下

```
SWI   1       ;软中断,中断立即数为 0
SWI   0x55    ;软中断,中断立即数为 0x55
```

使用 SWI 指令时,通常使用以下两种方法进行传递参数,SWI 异常中断处理程序可以提供相关的服务,这两种方法均是用户软件协定.SWI 异常中断处理程序要通过读取引起软中断的 SWI 指令.以取得 8 位立即数.

1.指令中 8 位的立即数指定了用户请求的服务类型,参数通过用寄存器传递.

```
MOV   R0,#34    ;设置子功能号为虎作 34
SWI   18        ;调用 18 号软中断
```

2.指令中的 8 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,参数通过其它的通用寄存器传递.

```
MOV   R0,#18        ;调用 18 号软中断
MOV   R1,#34        ;设置子功能号为了 4
SWI   0
```

## Thumb 伪指令

### ◆ ADR

小范围的地址读取伪指令.ADR 指令将基于 PC 相对偏移的地址值读取到寄存器中.ADR 伪指令格式如下.

ADR register,expr

其中 register 加载的目标寄存器.

expr 地址表达式.偏移量必须是正数并小于 1KB.Expr 必须局部定义,不能被导入.

ADR 伪指令举例如下

```
ADR   register, expr
```

其中 register 加载的目标寄存器。

expr 地址表达式。偏移量必须是正数并小说于 1KB。Expr 必须局部定义，不能被导入。

ADR 伪指令举例如下：

```
ADR   R0, TxtTab
...
TxtTab
DCB   ”ARM7TDMI”, 0
```

### ◆ LDR

大范围的地址读取伪指令.LDR 伪指令用于加载 32 位的立即数或一个地址值到指定寄存器.在汇编编译源程序时,LDR 伪指令被编译器替换成一条合适的指令.若加载的常数未超出 MOV 范围,则使用 MOV 或 MVN 指令代替 LDR 伪指令,否则汇编器将常量放入文字池,并使用一条程序相对偏移的 LDR 指令从文字池读出常量.LDR 伪指令格式如下.

LDR register,=expr/label_expr

其中, register 加载的目标寄存器

expr 32 位立即数.

label_expr 基于 PC 的地址表达式或外部表达式.

LADR 伪指令举例如下;

```
LDR   R0,=0x12345678      ;加载 32 位立即数 0x12345678
LDR   R0,=DATA_BUF+60     ;加载 DATA_BUF 地址+60
…
LTORG                     ;声明文字池
…
```

从 PC 到文字池的偏移量必须是正数小于是 1KB.

与 Thumb 指令的 LDR 相比,伪指令的 LDR 的参数有“=”号.

### ◆ NOP

空操作伪指令.NOP 伪指令在汇编时将会将会被代替成 ARM 中的空操作,比如可能为 MOV,R8,R8 指令等.NOP 伪指令格式如下.

```
NOP
```

NOP 可用于延进操作

# 伪指令

ARM 汇编程序的由机器指令,伪指令和宏指令组成.伪指令不像机器指令那样在处理器运行期间由机器执行,而是汇编程序对源程序汇编期间由汇编程序处理.在前面的指令集章节中,我们已经接触了几条常用到的伪指令,如 ADR ,ADRL,LDR,NOP 等,把它们和指令集一起介绍是因为它们在汇编时会被合适的机器指令代替,实现真正机器指令操作.宏是一段独立的程序代码,它是通过伪指令定义的,在程序中使用宏指令即可调用宏.当程序被汇编时,汇编程序将对每个调用进行展开,用宏定义取代源程序中的宏指令.

# 符号定义伪指令

符号定义伪指令用于定义 ARM 汇编程序的变量,对变量进行赋值以及定义寄存器名称,该类伪指令如下;

全局变量声明:GBLA,GBLL 和 GBLS

局部变量声明:LCLA,LCLL 和 LCLS

变量赋值: SETA,SETL 和 SETS

为一个通用寄存器列表定义名称:RLIST

为一个协处理器的寄存器定义名称:CN

为一个协处理定义名称: CP

为一个 VFP 寄存器定义名称:DN 和 SN

为一个 FPA 浮点寄存器定义名称:FN

## ◆ GBLA、GBLL、GBLS

全局变量声明伪指令.

GBLA 伪指令用于声明一个全局的算术变量,并将其初始化为 0;

GBLL 伪指令用于声明一个全局的逻辑变量,并将其初始化为{FALSE};

GBLS 伪指令用于声明一个全局的字符串变量,并将其初始化为空字符串“”。

伪指令格式;

```
GBLA    variable
```

```
GBLL    variable
GBLS    variable
```

其中 variable 定义的全局变量名,在其作用范围内必须惟一.全局变量的作用范围为包含该变量的源程序.

伪指令应用举例如下;

```
        GBLL    codedbg         ;声明一个全局逻辑变量
codebg  SETL    {TRUE}          ;设置变量为{TRUE}
        …
```

## ◆ LCLA、LCLL、LCLS

局部变量声明伪指令.用于宏定义的体中.

LCLA 伪指令用于声明一个局部的算术变量,并将其初始化为 0

LCLL 伪指令用于声明一个局部的逻辑变量,并将其初始化为{FALSE}

LCLS 伪指令用于声明一个局部的字符串变量,并将其初始化为空字符串“”

伪指令格式;

```
LCLA    variable
LCLL    variable
LCLS    variable
```

其中 variable 定义的局部变量名。在其作用范围内必须惟一。局部变量的作用范围为包含该局部变量只能在宏中进行声明及使用。

伪指令应用举例如下;

```
MACRO                   ;声明一个宏
SENDDAT  $dat           ;宏的原型
LCLA     bitno          ;声明一个局部算术变量
…
bitno    SETA   8       ;设置变量值为 8
…
MEND
```

## ◆ SETA、SETL、SETS

变量赋值伪指令.用于对已定义的全局变量,局部变量赋值.

SETA 伪指令用于给一个全局/局部的算术变量赋值.

SETL 伪指令用于给一个全局/局部的逻辑变量赋值.

SETS 伪指令用于给一个全局/局部的字符串变量赋值.

伪指令格式;

```
variable_a    SETA    expr_a
variable_l    SETL    expr_l
variable_s    SETS    expr_s
```

其中 variable_a 算术变量.用 GBLA,LCLA 伪指令定义的变量.

expr_a 赋值的常数.

variable_l 逻辑变量.用 GBLL,LCLL 伪指令定义的变量.

expr_l 逻辑值,即{TRUE}或{FALSE}.

variable_s 字符串变量.用 GBLS,LCLS 伪指令定义的变量.

expr_s 赋值的字符串.

伪指令应用举例如下.

```
        GBLS    ErrStr
        …
ErrStr  SETS    “No,semaphone”
        …
```

## ◆ RLIST

RLIST 为一个通用寄存器列表定义名称.伪指令格式如下:

```
name    RLIST    {reglist}
```

其中 name 要定义的寄存器列表的名称.

reglist 通用寄存器列表.

伪指令应用举例如下;

```
LoReg    RLIST    {R0-R7}    ;定义寄存器列表 LoReg
```

```
...
STMFD    SP!,LoReg   ;保存寄存器列表 LoReg
...
```

## ◆ CN

CN 为一个协处理器的寄存器定义名称.

伪指令格式;

**name CN expr**

其中 name 要定义的协处理器的寄存器名称.

expr 协处理器的寄存器编号,数值范围为 0~15.

伪指令应用举例如下;

```
MemSet   CN   1      ;将协处理的寄存器 1 名称定义为 MemSet
```

## ◆ CP

CP 为一个协处理器定义的名称.

伪指令格式;

**name CP expr**

其中 name 要定义的协处理器名称.

expr 协处理器的编号,数值范围为 0~15.

伪指令应用举例如下;

```
DivRun    CN   5   ;将协处理器 5 名称定义为 DivRun
```

## ◆ DN、SN

DN 和 SN 为 VFP 的寄存器的名称定义的伪指令.

DN 为一个双精度原 VFP 寄存器定义名称.

SN 为一个单精度的 VFP 寄存器定义名称.

伪指令格式;

**name DN expr**

name SN expr

其中 name 要定义的 VFP 寄存器名称.

expr 双精度的 VFP 寄存器编号为 0~15,单精度的 VFP 寄存器编号为 0~31.

伪指令应用举例如下;

```
cdn  DN  1    ;将 VFP 双精度寄存器 1 名称定义为 cdn
rex  SN  3    ;将 VFP 单精度寄存器 3 名称定义为 rex
```

### ◆ FN

FN 为一个 FPA 浮点寄存器定义名称

伪指令格式;

name FN expr

其中 name 要定义的浮点寄存器名称.

expr 浮点寄存器的编号,值为 0~7

伪指令应用举例如下;

```
ibq    FN  1    ;将浮点寄存器 1 名称定义为 ibq
```

## 数据定义伪指令

数据定义伪指令用于数据表定义,文字池定义,数据空间分配等.该类伪指令如下;

声明一个文字池：LTORG;

定义一个结构化的内存表的首地址：MAP

定义结构化内存表中的一个数据域：FIELD

分配一块内存空间,并用 0 初始化:SPACE

分配一段字节的内存单元,并用指定的数据初始化：DCB;

分配一段字的内存单元,并用指令的数据初始化：DCD 和 DCDU;

分配一段字的内存单元,将每个单元的内容初始化为该单元相对于静态基址寄存器的偏移量：DCDO;

分配一段双字的内存单元,并用双精度的浮点数据初始化：DCFD 和 DCFDU;

分配一段字的内存单元,并用单精度的浮点数据初始化：DCFS 和 DCFSU;

分配一段字的内存单元,并用单精度的浮点数据初始化,指定内存单元存放的是代码,而不是数据：DCI

分配一段双字的内存单元,并用 64 位整数数据初始化：DCQ 和 DCQU

分配一段半字的内存单元,并用指定的数据初始化：DCW 和 DCWU;

## ◆ LTORG

LTORG 用于声明一个文字池,在使用 LDR 伪指令时,要在适当的地址加入 LTORG 声明文字池,这样就会把要加载的数据保存在文字池内,再用 ARM 的加载指令读出数据.(若没有使用 LTORG 声明文字池,则汇编器会在程序末尾自动声明.)

伪指令格式：

**LTORG**

伪指令应用举例如下;

```
…
LDR    R0,=0x12345678
ADD    R1,R1,R0
MOV    PC,LR
LTORG                    ;声明文字池,此地址存储 0x12345678
…                        ;其它代码
```

LTORG 伪指令常放在无条件跳转指令之后,或者子程序返回指令之后,这样处理器就不会错误地将文字池中的数据当作指令来执行.

## ◆ MAP

MAP 用于定义一个结构化的内存表的首地址.此时,内存表的位置计数器{VAR}设置为该地址值{VAR}为汇编器的内置变量.^与 MAP 同义.

伪指令格式：

**MAP expr,{base_register}**

其中 expr 数字表达式或程序中的标号.当指令中没有 base_register 时,expr 即为结构化内存表的首地址.

base_register 一个寄存器.当指令中包含这一项时,结构化内存表的首地址为 expr 与 base_register 寄存器值的和.

伪指令应用举例如下;

```
        MAP     0x00,R9   ;定义内存表的首地址为 R9
Timer   FIELD   4         ;定义数据域 Timer,长度为 4 字节
Attrib  FIELD   4         ;定义数据域 Attrib,长度为 4 字节
String  FIELD   100       ;定义数据域 String,长度为 100 字节
        …
        ADR   R9,DataStart   ;设置 R9 的值,即设置结构化的内存表地址
        LDR   R0,Atrrib      ;相当于 LDR,R0,[R9,#4]
        …
```

MAP 伪指令和 FIELD 伪指令配合使用,用于定义结构化的内存表结构.MAP 伪指令中的base-register 寄存器的值对于其后所有的FIELD伪指令定义的数据域是默认使用的,直到遇到新的包含 base-register 项的 MAP 伪指令.

## ◆ FIELD

FIELD 用于定义一个结构化内存表中的数据域.#与 FIELD 同义.

伪指令格式:

{tabel} **FIELD** **expr**

其中 label 当指令中包含这一项时,label 的值为当前内存表的位置计数器{VAR}的值,汇编编译器处理了这条 FIELD 伪指令后,内存表计数器的值将加上 expr.

expr 表示本数据域在内存表中所占用的字节数.

伪指令应用举例如下;

```
        MAP     0x40003000   ;内存表的首地址为 0x40003000
count1  FIELD     4          ;定义数据域 count1,长度为 4 字节
count2  FIELD     4          ;定义数据域 count2,长度为 4 字节
count3  FIELD     4          ;定义数据域 count3,长度为 4 字节
        …
```

```
        LDR     R1,count1       ;R1=[0x40003000+0x00]
        STR     R1,count2       ;[0x40003000+0x00]=R1
```

MAP,FIELD 伪指令仅仅是定义数据结构,它们并不实际分配内存单元.

## ◆ SPACE

SPACE 用于分配一块内存单元,并用 0 初始化.%与 SPACE 同义.

伪指令格式：

{label}　SPACE　expr

其中　label　内存块起始地址标号.

expr　所要分配的内存字节数.

伪指令应用举例如下;

```
          AREA    DataRA,DATA, READWROTE      ;声明一数据段,名为 DataRAM
DataBuf   SPACE   1000                        ;分配 1000 字节空间
```

## ◆ DCB

DCB 用于分配一段字节内存单元,并用伪指令中的 expr 初始化.一般可用来定义数据表格,或文字符串.=与 DCB 同义.

伪指令格式：

{label}　DCB　expr{,expr}{,expr}…

其中　label　内存块起始地址标号.

expr　可以为-128~255 的数值或字符串.内存分配的字节数由 expr 个数决定.

伪指令应用举例如下

```
DISPTAB     DCB     0x33,0x43,0x76,0x12
            DCB     -120,20,36,55
ERRSTR      DCB     “Send,data is error!”,0
```

## ◆ DCD 和 DCDU

DCD用于分配一段字内存单元,并用伪指令中的expr初始化.DCD伪指令分配的内存需要字对齐,一般可用来定义数据表格或其它常数.&与 DCD 同义.

DCDU 用于分配一段字内存单元,并用伪指令中的 expr 初始化.DCD 伪指令分配的内存不需要字对齐,一般可用来定义数据表格或其它常数.

伪指令格式：

{label} DCD expr{,expr}{,expr}…

{label} DCDU expr{,expr}{,expr}…

其中 label 内存块起始地址标号.

expr 常数表达式或程序中的标号.内存分配字节数由 expr 个数决定.

伪指令应用举例如下：

```
Vectors
        LDR     PC,ReserAddr
        LDR     PC,UndefinedAddr
        …
ResetAddr       DCD   Reset
UndefinedAddr   DCD   Undefined
        …
Reset
       …
Undefined
       …
```

## ◆ DCDO

DCDO 用于分配一段字内存单元.并将每个单元的内容初始化为该单元相对于静态基址寄存器的偏移量.DCDO 伪指令作为基于静态基址寄存器 R9 的偏移量分配内存单元.DCDO 伪指令分配的内存需要字对齐.

伪指令格式;

{label} DCDO expr{,expr}{,expr}…

其中 label 内存块起始地址标号.

expr 地址偏移表达式或程序中的标号.内存分配的字数由 expr 个数决定.

伪指令应用举例如下;

```
IMPORT  externsym
DCDO    externsym    ;分配32位的字单元,其值为标号externsym基于R9的偏移
```

## ◆ DCFD 和 DCFDU

DCFD 用于分配一段双字的内存单元,并用双精度的浮点数据 fpliteral 初始化。每个双精度的浮点数占据两个字单元。DCFD 伪指令分配的内存需要字对齐。

DCFDU 具有 DCFD 同样的功能，但分配的内存不需要字对齐。

伪指令格式：

{label} DCFD fpliteral{,fpliteral}{,fpliteral}…

{label} DCFDU fpliteral{,fpliteral}{,fpliteral}…

其中 label 内存块起始地址标号.

fpliteral 双精度的浮点数.

伪指令应用举例如下;

```
DCFD   2E30,-3E-20
DCFDU  -.1,1000,2.1E18
```

## ◆ DCFS 和 DCFSU

DCFS 用于分配一段字的内存单元,并用单精度的浮点数据 fpliteral 初始化.每个单精度的浮点数占据一个字单元.DCFD 伪指令分配的内存需要字对齐.

DCFSU 具有 DCFS 同样的功能,但分配的内存不需要字对齐.

伪指令格式：

{label} DCFS fpliteral{,fpliteral}{,fpliteral}…

{label} DCFSU fpliteral{,fpliteral}{,fpliteral}…

其中 label 内存块起始地址标号

fpliteral 单精度的浮点数.

伪指令应用举例如下;

```
DCFS    1.1E2,-1.3E10,0.0999
```

## ◆ DCI

在 ARM 代码中,DCI 用于分配一段字节的内存单元,用指定的数据 expr 初始化.指定内存单元存放的是代码,而不是数据.

在 Thumb 代码中,DCI 用于分配一段半字节的内存单元,用指定的数据 expr 初始化.指定内存单元存放的是代码,而不是数据.

伪指令格式：

```
{label}  DCI  expr
```

其中 label 内存块起始地址标号.

expr 可为数字表达式.

DCI 伪指令和 DCD 伪指令非常类似,不同之处在于 DCI 分配的内存中的数据被标识为指令.可用于通过宏指令业定义处理器不支持的指令.

伪指令应用举例如下;

```
MACRO                   ;宏定义(定义 NEWCMN Rd,Rn 指令)
NEWCMN  $Rd,$Rm         ;宏名为 NEWCMN,参数为 Rd 和 Rm
DCI     0xe16a0e20:OR:($Rd:SHL:12):OR:$Rm
MEND
```

## ◆ DCQ 和 DCQU

DCQ 用于分配一段双字的内存单元,并用 64 位的整数数据 literal 初始化.DCQ 伪指令分配的内存需要字对齐.

DCQU 具有 DCQ 同样的功能,但分配的内存不需要字对齐.

伪指令格式：

```
{label}  DCQ   {-}literal{,{-}{literal}}…
{label}  DCQU  {-}literal{,{-}{literal}}…
```

其中 label 内存块起始地址标号.

literal 64 位的数字表达式.取值范围为 0~$2^{64}$-1 当 literal 前有“.”号时,取值范围为-$2^{63}$～-1 之间

伪指令应用举例如下;

```
DCQU    1234,-76568798776
```

### ◆ DCW 和 DCWU

DCW用于分配一段字的内存单元,并用指定的数据expr初始化.DCW伪指令分配的内存需要字对齐.

DCWU 具有 DCW 同样的功能,但分配的内存不需要字对齐.

伪指令格式：

```
{label}    DCW    expr{,expr}{,expr}…
{label}    DCWU   expr{,expr}{,expr}…
```

其中 label 内存块起始地址标号.

expr 数字表达式,取值范围为-32768~65535.

伪指令应用举例如下;

```
DCW    -592,123,6756
```

## 报告伪指令

报告伪指令用于汇编报告指示.该类伪指令如下：

断言错误:ASSERT;

汇编诊断信息显示:INFO;

设置列表选项:OPT;

插入标题:TTL 和 SUBT.

### ◆ ASSERT

ASSERT 为断言错误伪指令.在汇编编译器对汇编程序的第二遍扫描中,如果其中ASSERT 条件不成立,ASSERT 伪指令将报告该错误信息.

伪指令格式：

ASSERT　　Logical_expr

其中　　Logical_expr　　用于断言的逻辑表达式

伪指令应用举例如下

```
ASSERT    Top<>Temp    ;断言 Top 不等于 Temp
```

## ◆ INFO

汇编诊断信息显示伪指令,在汇编器处理过程中的第一遍扫描或第一遍扫描时报告诊断信息.

伪指令格式:

INFO　　numeric_expr,string_expr

其中　　numeric_expr　　数据表达式.若值为 0,则在第一遍扫描时报告诊断信息.否则在第一遍扫描时报告诊断信息.

strint_expr　　要显示的字串

伪指令应用举例如下:

```
INFO   0,”Version 0.1”       ;在第二遍扫描时，报告版本信息
if  cont1 > cont2            ;如果 cont1 > cont2
INFO   1,”cont1 > cont2”     ;则在第一遍扫描时报告”cont1 > cont2”
```

## ◆ OPT

设置列表选项伪指令.通过 OPT 伪指令可以在源程序中设置列表选项.

伪指令格式:

OPI　　n

其中 n 所设置的选项的编码如下:

1　　设置常规列表选项

2　　关闭常规列表选项

4　　设置分页符,在新的一页开始显示

8　　将行号重新设置为 0

16 设置选项,显示 SET,GBL,LCL 伪指令

32 设置选项,不显示 SET,GBL,LCL 伪指令

64 设置选项,显示宏展开

128 设置选项,不显示宏展开

256 设置选项,显示宏调用

512 设置先项,不显示宏调用

1024 设置选项,显示第一遍扫描列表

2048 设置选项,不显示第一遍扫描列表

4096 设置选项目,显示条件汇编伪指令

8192 设置选项,不显示条件汇编伪指令

16384 设置选项,显示 MEND 伪指令

32768 设置选项,不显示 MEND 伪

默认情况下,-list 选项生成常规的列表文件,包括变量声明,宏展开,条件汇编伪指令及MEND伪指令,而且列表文件只是在第二遍扫描时给出,通过OPT伪指令,可以在源程序中改变默认的选项.

伪指令应用举例如下

```
…           ;代码
OPT   512   ;不显示宏调用
…           ;代码
```

## ◆ TTL 和 SUBT

TTL 和 SUBT 为插入标题伪指令.

TTL 伪指令在列表文件的每一页的开头插入一个标题.该 TTL 伪指令的作用在其后的每一页,直到遇到新的 TTL 伪指令.

SUBT 伪指令在列表文件的每页的开头第一个子标题.该 SUBT 伪指令的作用在其后的每一页,直到遇到新的 SUBT 伪指令.

伪指令格式:

```
TTL    title
SUBT   subtitle
```

其中 title 标题名.

subtitle 子标题名.

伪指令应用举例如下;

```
…
TTL    mainc
…
SUBT   subc con
…
```

# 汇编控制伪指令

汇编控制伪指令用于条件汇编,宏定义,重复汇编控制等.该类伪指令如下:

条件汇编控制: IF,ELSE 和 ENDIF

宏定义: MACRO 和 MEND

重复汇编: WHILE 及 WEND

## ◆ IF、ELSE 和 ENDIF

IF ,ELSE 和 ENDIF 伪指令能够根据条件把一段代码包括在汇编程序内或将其排除在程序之外.

[与 IF 同义,|与 ELSE 同义,]与 ENDIF 同义

伪指令格式:

```
IF     logical_expr
       ;指令或伪指令代码段 1
{
ELSE
       ;指令或伪指令代码段 2
}
```

ENDIF

其中 logical_expr 用于控制的逻辑表达式.若条件成立,则代码段落在汇编源程序中有效.若条件不成立,代码段 1 无效,同时若使用 ELSE 伪指令,代码段有效.

伪指令应用举例如下;

```
…
IF {CONFIG}=16
   BNE     __rt_udiv_1
   LDR     R0,=__rt_div0
   BX      R0
ELSE
   BEQ     __rt_div0
ENDIF
```

IF,ELSE 和 ENDIF 伪指令是可以嵌套使用的.

## ◆ MACRO 和 MEND

MACRO 和 MEND 伪指令用于宏定义.MACRO 标识宏定义的开始,MEND 标识宏定义久的结束.用MACRO及MEND定义的一段代码,称为宏定义体.这样在程序中就可以通过宏指令多次调用该代码段.

伪指令格式:

```
MACRO
{$label}    macroname     {$parameter} {$parameter}…
            ;宏定义体.
MEND
```

其中 $label 宏指令被展开时,label 可被替换成相应的符号,通常为一个标号在一个符号前使用$表示被汇编时将使用相应的值替代$后的符号.

macroname 所定义的宏的名称.

$parameter 宏指令的参数.当宏指令被展开时将被替换成相应的值,类

似于函数中的形式参数

对于子程序代码比较短,而需要传递的参数比较多的情况下可以使用汇编技术.首先要用MACR和MEND伪指令定义宏,包括宏定义体代码.在MACRO伪指令之后的第一行声明宏的原型,其中包含该宏定义的名称,及需要的参数.在汇编程序中可以通过该宏定义的名称来调用它.当源程序被汇编时,汇编编译器将展开每个宏调用,用宏定义体代替源程序中的宏定义的名称,并用实际的参数值代替宏定义时的形式参数.

伪指令应用举例如下

```
MACRO
CSI_SETB                ;宏名为 CSI_SETB,无参数
LDR     R0,=rPDATG      ;读取 GPG0 口的值
LDR     R1,[R0]
ORR     R1,R1#0x01      ;CSI 置位操作
STR     R1,[R0]         ;输出控制
MEND
```

带参数的宏定义如程序清单:

```
    MACRO
$IRQ_Label      HANDLER   $IRQ_Exception
                EXPORT    $IRQ_Label
                IMPORT    $IRQ_Exception
$IRQ_Label
          SUB     LR,LR,#4
          STMFD   SP!,{R0-R3,R12,LR}
          MRS     R3,STSR
          STMFD   SP!,{R3}
          …
    MEND
```

## ◆ WHIL 和 WEND

WHILE 和 WEND 伪指令用于根据条件重复汇编相同的或几乎相同的一段源程序.

伪指令格式:

```
WHILE    logical_expr
        ;指令或伪指令代码段
WEND
```

其中　　logical_expr　　用于控制的逻辑表达式.若条件成立,则代码段在汇编源程序中有效,并不断重复这段代码直到条件不成立.

伪指令应用举例如下;

```
WHILE   no<5
no      SETA        no+1
        …
        WEND
```

WHILE 和 WEND 伪指令是可以嵌套使用的.

## 杂项伪指令

杂项伪指令在汇编编程设计较为常用,如段定义伪指令,入口点设置伪指令,包含文件伪指令,标号导出或引入声明等,该类伪指令如下;

边界对齐:　ALIGN

段定义:　　AREA

指令集定义: CODE16 和 CODE32

汇编结束:　END

程序入口:　ENTRY

常量定义:　EQU

声明一个符号可以被其它文件引用:EXPORT 和 GLORBAL

声明一个外部符号:IMPORT 和 EXTERN

包含文件:GET 和 INCLUDE

包含不被汇编的文件:INCBIN

保留符号表中的局部符号:KEEP

禁止浮点指令:NOFP

指示两段之间的依赖关系:REQUIRE

堆栈 8 字节对准:PEQUIRE8 和 PRESERVE8

给特定的寄存器命名:RN

标记局部标号使用范围的界限:ROUT.

## ◆ ALIGN

ALIGN 伪指令通过添加补丁字节使当前位置满足一定的对齐方式.

伪指令格式:

ALIGN {expr{,offset}}

其中 expr 数字表达式,用于指定对齐的方式.取值为 2 的 n 次幂,如 1,2,4,8,等,不能为 0 其没有 expr.,则默认为字对齐方式.

offset 数字表达式,当前位置对齐到下面形式的地址处:offset+n*expr

在下面的情况中,需要特定的地址对齐方式;

1.Thumb 伪指令 ADR 要求地址是字对齐的.而 Thumb 代码中地址标号可能不是字对齐的.这时就要使用伪指令 ALIGN4 使 Thumb 代码中地址标号为字对齐.

2.由于有些 ARM 处理器的 Cache 采用了其他对齐方式.如 16 字节对齐方式,这时使用 ALIGN 伪指令指定合适的对齐方式可以充分发挥 Cache 的性能优势.

3.LDRD 和 STRD 指令要求存储单元为 8 字节对齐.这样在为 LDRD/STRD 指令分配的存储单元前要使用伪指令 ALIGN8 实现 8 字节对齐方式.

4.地址标号通常自身没有对齐要求,而在 ARM 代码中要求地起标号对齐是字对齐的,Thumb 代码中要求半字对齐.这样可以使用 ALIGN4 和 ALIGN2 伪指令来调整对齐方式.

伪指令应用举例如下:

通过 ALIGN 伪指令使程序中的地址标号字对齐:

```
        AREA  Example ,CODE,READONLY   ;声明代码段 Example
START   LDR   R0,=Sdfjk
        …
        MOV   PC,LR
Sdfjk   DCB   0x58          ;定义一字节存储空间,字对齐方式被破坏
        ALIGN               ;声明字对齐
```

```
SUBI    MOV   R1,R3     ;其它代码
        …
        MOV   PC ,LR
```

在段定义 AREA 中,也可使用 ALIGN 伪指令对齐,但表达式的数字含义是同的

```
                AREA   MyStack,DATA,NOINIT,ALIGN=2    ;声明数据段
                                                      ;MyStack,并重新字对齐
IrqStackSpace   SPACE  IRQ_STACK_LEGTH*4   ;中断模式堆栈空间
FiqStackSpace   SPACE  FIQ_STACK_LEGTH*4   ;快速中断模式堆栈空间
AbtStackSpace   SPACE  ABT_STACK_LEGTH*4   ;中止义模式堆栈空间
UndtStackSpace SPACE   UND_STACK_LEGTH*4   ;未定义模式堆栈
      …
```

将两个字节的数据放在同一个字的第一个字节和第四个字节中,带 offset 的 ALIGN 对齐:

```
AREA     offsetFxample, CODE
DCB      0x31                  ;第一个字节保存 0x31
ALIGN    4,3                   ;字对齐
DCB      0x32                  ;第四个字节保存 0x32
…
```

## ◆ AREA

AREA 伪指令用于定义一个代码段或数据段.ARM 汇编程序设计采用分段式设计,一个 ARM 源程序至少需要一个代码段,大的程序可以包含多少个代码段及数据段.

伪指令格式:

**AREA sectionname {,attr} {,attr}…**

其中 sectionname 所定义的代码段或数据段的名称.如果该名称是以数据开头的,则该名称必须用“|”括起来,如|1_datasec|.还有一些代码段具有的约定的名称.如|text|表示 C 语言编译器产生的代码段或者与 C 语言库相关的代码段.

attr 该代码段或数据段的属性.

在 AREA 伪指令中,各属性之间用逗号隔开.以下为段属性及相关说明：

**ALIGN = expr**.默认的情况下,ELF 的代码段和数据段是 4 字节对齐的，expr 可以取 0~31 的数值,相应的对齐方为 2expr 字节对齐.如 expr=3 时为字节对齐.对于代码段,expr 不能为 0 或 1；

**ASSOC = section**.指定与本段相关的 ELF 段.任何时候连接 section 段也必须包括 sectionname 段；

**DODE** 为定义代码段.默认属性为 READONLY；

**COMDEF** 定义一个通用的段.该段可以包含代码或者数据.在其它源文件中,同名的 COMDEF 段必须相同；

**COMMON** 定义一个通用的段.该段不包含任何用户代码和数据,连接器将其初始化为此.各源文件中同名的 COMMON 段共用同样的内存单元,连接器为其分配合适的尺寸；

**DATA** 为定义段.默认属性为 READWRITE；

**NOINIT** 指定本数据段仅仅保留了内存单元,而没有将各初始写入内存单元,或者将内存单元值初始化为 0；

**READONLY** 指定本段为只读,代码段的默认属性为 READONLY；

**READWRITE** 指定本段为可读可写.数据段的默认属性为 READWRITE；

使用 AREA 伪指令将程序分为多个 ELF 格式的段,段名称可以相同，这时同名的段被放在同一个 ELF 段中.

伪指令应用举例如下；

```
AREA    Example ,CODE,READNOLY     ;声明一个代码,名为 Example
```

## ◆ CODE16 和 CODE32

CODE16 伪指令指示汇编编译器后面的指令为 16 位的 Thumb 指令.

CODE32 伪指令指示汇编编译器后面的指令为 32 位的 ARM 指令.

伪指令格式：

```
CODE16
CODE32
```

CODE16 和 CODE32 伪指令只是指示汇编编译器后面的指令的类型,伪指令本身并不进行程序状态的切换.要进行状态切换,可以使用 BX 指令操作.

伪指令应用举例如下；

```
AREA  Example CODE,READONLY
CODE32
…
```

使用 CODE16 和 CODE32 定义 Thumb 指令及 ARM 指令并用 BX 指令进行切换。

CODE16 和 CODE32 的使用：

```
    AREA        ArmThumC,CODE,READONLY
    CODE32
    ADR         R0,ThumbStart+1
    BX          R0
    CODE16
ThumbStart
    MOV         R0,#10
    …
END
```

## ◆ END

END 伪指令用于指示汇编编译器源文件已结束.每一个汇编源文件均要使用一个 END 伪指令,指示本源程序结束

伪指令格式：

END

## ◆ ENTRY

ENTRY 伪指令用于指定程序的入口点.

伪指令格式：

ENTRY

一个程序(可以包含多个源文件)中至少要有一个 ENTRY,可以有多个 ENTRY.但一个源文件中最多只有一个 ENTRY.

伪指令应用举例如下.

```
AREA,     Example,  CODE,READNOLY
ENTRY
CODE32
START    MOV   R1，#0x5F
…
```

## ◆ EQU

EQU 伪指令为数字常量，基于寄存器的值和程序中的标号定义一个名称。*与 EQU 同义。

指令格式：

**name　　EQU　　expr{,type}**

其中　name　要定义的常量的名称.

　　expr　基于寄存器的地址值,程序中的标号,32 位地址常量或 32 位常量.

　　type　当 expr 为 32 位常量时,可用 type 指示 expr 表示的数据类型.如下

　　CODE16

　　CODE32

　　DATA

EQU 伪指令的作用类似于 C 语言中的#define.用于为一个常量定义名称.

伪指令应用举例如下;

```
T_bit    EQU    0x20          ;定义常量 T_bit,其值为 0x20
PLLCON   EQU    0xE01FC080    ;定义寄存器 PLLCON,地址为 0Xe01F080
ABCD     EQU    label+8       ;定义 ABCD 为 label+8
```

## ◆ EXPORT 和 GLOBAL

EXPORT 声明一个符号可以被其它文件引用.相当于声明了一个全局变量.

GLOBAL 与 EXPORT 相同

指令格式：

EXPORT symbol{[WEAK]}

GLOBAL symbol{[WEAK]}

其中 symbol 要声明的符号名称

[WEAK] 声明其它的同名符优先于本符号被引用.

伪指令应用举例如下：

```
EXPORT  InitStack
GLOBAL  Vectors
```

## ◆ IMPORT 和 EXTERN

IMJPORT 伪指令指示编译器当前的符号不是在本源文件中定义的,而是在其他源文件中定义的,在本源文件中可能引用该符号.

EXTERN 与 IMPORT 相同

指令格式：

IMPORT symbol{[WEAK]}

EXTERN symbol{[WEAK]}

其中 symbol 要声明的符号名称.

[WEAK] 指定该选项后,如果 symbol 在所有的源程序中都没有被定义,编译器也不会生任何错误信息,同时编译器也不会到当前没有被 INCLUDE 进来库中去查找该标号.

使用 IMPORT 或 EXTERN 声明外部标号时,若连接器在连接处理时不能解释该符号,而伪指令中没有[WEAK]选项,则连接器会报告错误,若伪指令中有[WEAK]选项,则连接器不会报告错误,而是进行下面的操作：

1.如果该符号被 B 或者 BL 指令引用,则该符号被设置成下一条指令的地址,该 B 或者 BL 指令相当于一条 NOP 指令.

2.其它情况下该符号被设置 0

伪指令应用举例如下

```
IMPORT  InitStack
EXTERN  Vectors
```

## ◆ GET 和 INCLUDE

GET 伪指令将一个源文件包含到当前源文件中,并将被包含的文件在具当前位置进行汇编处理 INCLUDE 与 GFT 同义

指令格式

```
GET       filename
INCLUDE   filename
```

其中　　filename　要包含的源文件名,可以使用路径信息.

GET 伪指令通常用于包含一些宏定义或常量定义的源文件.如用 EQU 定义的常量,用 MAP 和 FIELD 定义的结构化的数据类型,这样的源文件类似于 C 语言中的头文件,GET,INCLUDE 伪指令不能用来包含目标文件,而 INCBIN 伪指令可以包含目标文件.

伪指令应用举例如下

```
INCLUDE     LPC2106.inc
```

## ◆ INCBIN

INCBIN 伪指令将一个文件包含到当前源文件中,而被包含的文件不进行汇编处理.

指令格式:

```
INCBIN    filename
```

其中　　filename　要包含的源文件名,可以使用路径信息.

通常可以使用 INCBIN 将一个执行文件或者任意数据包含到当前文件中,被包含的执行文件或数据将被原封不动地放下当前文件中,编译器从 INCBIN 伪指令后面开始继续处理.

伪指令应用举例如下

```
INCBIN    charlib.bin
```

## ◆ KEEP

KEEP 伪指令指示编译器保留符号表中的局部符号.

伪指令格式

KEEP {symbol}

其中 symbol 要保留的局部标号.若没有此项,则除了基于寄存器处的所有符号将包含在目标文件的符号表中.

## ◆ NOFP

NOFP 伪指令用于禁止源程序中包含浮点运算指令.

伪指令格式.

NOFP

## ◆ REQUIRE

REQUIRE 伪指令指定段之间的依赖关系.

伪指令格式

REQUIRE label

其中 label 所需要的标号的名称.

当进行链接处理时,包含了 REQUIRE label 伪指令的源文件,则定义 label 的源文件也被包含.

## ◆ PEQUIRE8 和 PRESERVE8

PEQUIRE8 伪指令指示当前文件请求堆栈为 8 字节对齐,

PRESERVE8 伪指令指示当前文件保持堆栈为 8 字节对齐.

伪指令格式.

PEQUIRE8

PRESERVE8

链接器保证要求 8 字节对齐的堆栈只能被堆栈为 8 字的对齐的代码调用.

## ◆ RN

RN 伪指令用于给一个特殊的寄存器命名.

伪指令格式

name RN expr

其中 name 给寄存器定义的名称.

expr 寄存器编号

伪指令应用举例如下

```
COUNT   RN 6     ;定义寄存器 R6 为 COUT
Count1  RN R7    ;定义寄存器 R7 为 Cout1
```

## ◆ ROUT

ROUT 伪指令用于定义局部标号的有效范围.

伪指令格式如下.

{name} ROUT

其中 name 所定义的作用范围的名称.

当没有使用 ROUT 伪指令时,局部标号的作用范围为其所在段.ROUT 伪指令的作用范围在本 ROUT 伪指令和下一个 ROUT 伪指令之间(指同一段中的 ROUT 伪指令.)

伪指令应用举例如下.

```
routineA    ROUT              ;定义局部标号的有效范围,名称为 routineA
              …
3routineA                     ;routineA 范围内的局部标号 3
              …
            BEQ   %4routineA ;若条件成立,跳转到 routineA 范围内的局部标号 4
            …
            BEG   %3          ;若条件成立,跳转到 routineA 范围内的局部标号 3
            …
4routineA   …                 ;routineA 范围内的局部标号 4
            …
otherstuff ROUT               ;定义新的局部标号的有效范围
```

# ARM 伪指令

ARM 伪指令有 ADR,ADRL,LDR,NOP,LDFD,LDFS.

## ◆ ADR

为小范围的地址读取伪指令.ADR 指令将基于 PC 相对偏移的地址值或基于寄存器相对偏移的地址值读取到寄存器中,当地址值是非字对齐时,取值范围-255~255 字节之间,当地址值是字对齐时,取值范围-1020~1020 字节之间

## ◆ ADRL

为中等范围的地址读取伪指令.ADRL 指令将基于 PC 相对偏移的地址值或基于寄存器相对偏移的地址值读取到寄存器中,当地址值是非字对齐时,取值范围-64K~64K 字节之间,当地址值是字对齐时,取值范围-256K~256K 字节之间

## ◆ LDR

为大范围的地址读取伪指令.LDR 伪指令用于加载 32 位的立即数或一个地址值到指定寄存器.若汇编器将常量放入文字池,并使用一条程序相对偏移的 LDR 指令从文字池读出常量,则从 PC 到文字池的偏移量必须小于 4KB.

## ◆ NOP

为空操作伪指令.NOP 伪指令在汇编时将会被代替成 ARM 中的空操作,比如可能为 MOV R0,R0 指令等.

## ◆ LDFD

伪指令将一个双精度浮点数常数放进一个浮点数寄存器.

伪指令格式:

```
LDFD  fx,=expr
```

其中 fx 浮点数寄存器

expr 双精度浮点数值.

伪指令应用举例如下

```
LDFD    f1,=0.12
```

### ◆ LDFS

伪指令将一个单精度浮点数常数放进一个浮点寄存器.

伪指令格式:

```
LDFS    fx,=expr
```

其中 fx 浮点数寄存器

expr 单精度浮点数值.

伪指令应用举例如下

```
LDFS    f1,=0.12
```

## Thumb 伪指令

Thumb 伪指令有 ADR,LDR,NOP.

### ◆ ADR

为小范围的地址读取伪指令.ADR 指令将基于 PC 相对偏移的地址值读取到寄存器中,偏移量必须是正数并小于 1KB.

### ◆ LDR

为大范围的地址读取伪指令.LDR 伪指令用于加载 32 位的立即数或一个地址值到指定寄存器.

若汇编器将常量放入文字池,并使用一条程序相对偏移的 LDR 指令从文字池读出常量,则从 PC 到文字池的偏移量必须是正数并小于 1KB

### ◆ NOP

为空操作伪指令.NOP 伪指令在汇编时将会被代替成 ARM 中的空操作,比如可能为 MOV R8,R8 指令等.

# ARM 汇编程序设计

# 文件格式

ARM 源程序文件(即源文件)为文件格式,可以使用任一文本编辑器编写程序代码.一般地,ARM 源程序文件名的后缀名如下表:

| 程序 | 文件名 |
|---|---|
| 汇编 | *.S |
| 引入文件 | *.INC |
| C 程序 | *.C |
| 头文件 | *.H |

在一个项目中,至少要有一个汇编源文件或 C 程序文件,可以有多个汇编源文件或多个 C 程序文件,或者 C 程序文件和汇编文件两者的组合.

# ARM 汇编的一些规范

## 汇编语句格式

ARM 汇编中,所有标号必须在一行的顶格书写,其后面不要添加“:”,而所有指令均不能顶格书写。ARM 汇编器对标识符大小写敏感,书写标号及指令时字母大小写要一致,在 ARM 汇编程序中,一个 ARM 指令、伪指令、寄存器名可以全部为大写字母,也可以全部为小写字母,但不要大小写混合使用。注释使用“;”,注释内容由“;”开始到此行结束,注释可以在一行的顶格书写。

格式:[标号] <指令|条件|S> <操作数>[;注释]

源程序中允许有空行,适当地插入空行可以提高源代码的可读性。如果单行太长,

可以使用字符“\”将其分行，“\”后不能有任何字符，包括空格和制表符等。对于变量的设置，常量的定义，其标识符必须在一行的顶格书写。

汇编指令正确的例子和错误的例子如下：

**正确的例子：**

```
…
Str1        SETS    My string1.”          ;设置字符串变量 Str1
Count       RN      R0                    ;定义寄存器名 Count
USR_STACK   EQU     64                    ;定义常量

START       LDR     R0,=0x1123456         ;R0=0x123456H
            MOV     R1,#0
LOOP
            MOV     R2,#3
…
```

**错误的例子：**

```
  START  MOV    R0,#1          ;标号 START 没有顶格写
ABC:     MOV    R1,#2          ;标号后不能带：
MOV    R2,#3                  ;命令不允许顶格书写
loop     Mov    R2,#3          ;指令中大小写混合
         B      Loop           ;无法跳转到 Loop 标号
```

## 标号

在 ARM 汇编中，标号代表一个地址，段内标号的地址在汇编时确定，而段外标号的地址值在连接时确定。根据标号的生成方式，可以有以下 3 钟：

### ◆ 基于 PC 的标号

基于 PC 的标号时位于目标指令前的标号或程序中的数据定义伪指令前的标号，这种标号在汇编时将被处理成 PC 值加上或减去一个数字常量。它常用于表示跳转指令的

目标地址，或者代码段中所嵌入的少量数据。

### ◆ 基于寄存器的标号

基于寄存器的标号通常用 MAP 和 FILED 伪指令定义，也可以用于 EQU 伪指令定义，这种标号在汇编时被处理成寄存器的值加上或减去一个数字常量。它常用于访问位于数据段中的数据。

### ◆ 绝对地址

绝对地址是一个 32 为的数字量，它可以寻址的范围为 0～$2^{32}$－1，可以直接寻址整个内存空间。

## 局部标号

局部标号主要用于局部范围代码中，在宏定义也是很有用的。局部标号是一个 0～99 之间的十进制数字，可重复定义，局部标号后面可以紧接一个通常表示该局部变量作用范围的符号。局部变量的作用范围为当前段，也可以用伪指令 ROUT 来定义局部标号的作用范围。

局部标号定义格式：N{routname}

其中：N　　局部标号，为 0～99

routname　　局部标号作用范围的名称，由 ROUT 伪指令定义

局部标号引用格式：

%{F|B}{A|T}　　N{routname}

其中：　%　　表示局部标号引用操作

F　　指示编译器只向前搜索

B　　指示编译器只向后搜索

A　　指示编译器搜索宏的所有嵌套层次

T　　指示编译器搜索宏的当前层

如果 F 和 B 都没有指定，则编译器先向前搜索，再向后搜索。如果 A 和 T 都没有指定，编译器搜索所有从宏的当前层次到宏的最高层次，比当前层次的层次不再搜索。

如果指定了 routname，编译器向前搜索最近的 ROUT 伪指令，若 routname 与该 ROUT 伪指令定义的名称不匹配，编译器报告错误，汇编失败。

示例如下：

```
routintA      ROUT
          …
3routineA
              BEQ     %4routineA
              BGE     %3
4routineA
          …
otherstuff    ROUT
          …
```

## 符号

在 ARM 汇编中，符号可以代表地址、变量、数字常量。当符号代表地址时又称为标号，符号就是变量的变量名、数字常量的名称、标号，符号的命名规则如下：

1、符号由大小写字母、数字以及下划线组成；

2、除局部标号以数字开头外，其它的符号不能以数字开头；

3、符号区分大小写，且所有字符都是有意义的；

4、符号在其作用域范围你必须是唯一的；

5、符号不能与系统内部或系统预定义的符号同名；

6、符号不要与指令助记符、伪指令同名。

## 常量

### ◆ 数字常数

数字常量有三种表示方式：

十进制数，如：12，5，876，0；

十六进制数，如 0x4387，0xFF0，0x1；

n 进制数，用 n-XXX 表示，其中 n 为 2~9，XXX 为具体的数。如 2-010111，8-4363156 等。

## ◆ 字符常量

字符常量由一对单引号及中间字符串表示，标准 C 语言中的转义符也可使用。如果需要包含双引号或“$”，必须使用“”和$$代替。如下：

```
Hello     SETS     “Hello World!”
Errorl    SETS     “The parameter ““VFH”” error$$2”
```

## ◆ 布尔常量

布尔常量的逻辑真为{TRUE}，逻辑假为{FALSE}.如下：

```
testno    SETS     {FALSE}
```

### 段定义

ARM汇编程序设计采用分段式设计，一个ARM源程序至少需要一个代码段，大的程序可以包含多个代码段及数据段.

ARM 汇编程序经过汇编处理后生成一个可执行的映象文件，该文件通常包含下面 3 部分内容：

- 一个或多代码段.代码段通常是只读的.
- 零个或多个包含初始化值的数据段.这些数据段通常是可读写的.

零个或多个不包含初始值的数据段.这些数据被初始化为 0，通常中可读写的.

连接器根据一定的规则将各个段安排到内存中的相应位置.源程序中段之间的相邻关系与执行的映象文件中段之间的相邻关系并不一定相同.

代码段的例子如下：

```
        AREA    Hello,CODE,READONLY       ;声明代码段 Hello
        ENTRY                             ;程序入口(调试用)
START   MOV     R7,#10
        MOV     R6,#5
```

```
        ADD     R6,R6,R7                ;R6=R6+R7
        B                               ;死循环
        END
```

每一个汇编文件都要以 END 结束,包括*INC 文件,否则编译会有警告.

数据段的例子如下：

```
            AREA    DataArea,DATA,NOINIT,ALGN=2
DISPBUF     SPACE   100
RCVBUF      SPACE   100
            …
```

**宏定义及其作用**

使用宏定义可以提高程序的可读性,简化程序代码和同步修改.ARM 宏定义与标准 C 的#define 相似,只在源程序中进行字符代换.宏定义从 MACRO 伪指令开始,到 MEND 结束,并可以使用参数.

宏要先定义,然后再使用.使用时直接书写宏名,并根据对应的宏定义格式设置输入参数或书写标号等.当源程序被汇编时,汇编编译器将展开每一个宏调用,用宏定义体代替程序中的宏调用,并使用实际的参数值代替宏定义时的形式参数.

程序程序清单见后,程序中定义了一个宏 CALL,用于调用子程序,调用时设置所要调用的子程序名$Function 及两个入口参数$dat1 和$dat2.由于宏定义体中使用的是 MOV 指令,所以$dat1 参数只能为 8 位图的立即数或通用寄存器.

宏应用的例子：

```
    …
MACRO                               ;宏定义
CALL    $Function,$dat1,$dat2       ;宏名称为 CALL,带 3 个参数
IMPORT  $Function                   ;声明外部子程序
MOV     R0,$dat1                    ;设置子程序参数,R0=$dat1
MOV     R1,$dat2
BL      Function                    ;调用子程序
MEND                               ;宏定义结束
```

```
…
CALL    FADD1,#3,#2          ;宏调用
…
```

汇编预处理后,宏调用将被展开,程序清单如下：

```
        …
        IMPORT      FADD1
        MOV         R0,#3
        MOV         R1,#3
        BL          FADD1
        …
```

## 子程序的调用

使用 BL 指令进行调用,该指令会把返回的 PC 值保存在 LR,示例如下：

```
            …
            BL    DLEAY
            …
DELAY       …
            MOV   PC,LR
```

当子程序执行完毕后,使用 MOV,B/BX,STMFD 等指令返回,当然 STMFD 要与 LDMFD 配套使用,子程序返回的方法：

```
        MOV     PC,LR
或      B       LR
或      BX      LR
或      STMFD   SP!{R0-R7,PC }
```

ARM7TDMI(-S)是没有BLX指令的,但是可以通过几条程序实现其功能,模拟BLX指令如下：

```
            ADR     R1,DELAY+1
            MOV     LR,PC               ;保存返回地址到 LR
            BX      R1                  ;跳转并切换指令集
```

```
…
```

## 数据比较跳转

汇编程序可以使用CMP指令进行两个数据比较,然后调高相应的ARM条件码,实现跳转.代码例子如下：

```
CMP     R5,#10
BEQ     DOEQUAL         ;若 R5 为 10,则跳转到 DOEQUAL
…
CMP     R1,R2
ADDHI   R1,R1,#10       ;若 R1>R2,则 R1=R1+10
ADDLS   R1,R1,#5        ;若 R1<=R2,则 R1=R1+5
…
ANDS    R1,R1,#0x80     ;R1=R1&0x80,并设置相应标志位
BNE     WAIT            ;若 R1 的 d7 位为则跳转到 WAIT
```

## 循环

下面的代码为循环程序的例子.例子指定循环次数,每循环一次进行减 1 操作,并判断结果是否为 0,若为 0 则退出循环.

```
        MOV     R1,#10
LOOP    …                       ;循环体
        SUBS    R1,R1,#1
        BNE     LOOP
        …
```

## 数据块复制

程序可以使用存储器访问指令 LDM/STM 指令进行读取和存储,数据块复制示例如下：

```
        LDR     R0,=DATA_DST    ;指向数据目标地址
```

```
        LDR     R1,=DATA_SRC      ;指向数据源地址
        MOV     R10,#10           ;复制数据个数为 10*N 个字
LOOP    LDMIA   R1!,{R2-R9}       ;N 为 LDM/STM 指令操作数据个数
        STMIA   R0!,{R2-R9}
        SUBS    R10,R10,#1
        BNE     LOOP
         …
```

## 栈操作

ARM 使用存储器访问指令 LDM/STM 实现栈操作,用于子程序寄存器保存.注意,使用堆栈时,要先分配好堆栈空间,设置好寄存器 R13(即堆栈指针 SP),否则操作失败.

```
OUTDAT
            STMFD       SP!{R0-R7,LR}
            …
            BL          DELAY
            …
            LDMFD       SP!{R0-R7,PC}
```

## 特殊寄存器定义及应用

基于 ARM 核的芯片一般有片内外设,它们通过其特殊寄存器访问.片内外设的使用示例如下：

```
WDTC    EQU     0xE000000     ;寄存器定义
        …
        LDR     R0,=WDTC
        MOV     R1,#0x12
        STR     R1,[R0]       ;WDTC=0x12
```

## 散转功能

散转是汇编程序常用的一种算法,其示例如下：

```
CMP     R0,#MAXINDEX        ;判断索引号是否超出最大索引值
ADDLO   PC,PC,R0,LSL #2     ;若没有超出,则跳转到相应位置
B       ERROR               ;若已经超出,则进行出错处理
;散转表,对应索引号为 0～N
B       FUN1
B       FUN2
B       FUN3
…
```

## 查表操作

查表操作是汇编程序常用的一种操作,其示例如下：

```
        …
        LDR     R3,=DISP_TAB        ;取得表头
        LDR     R2,[R3,R5,LSL #2]   ;根据 R5 的值查表,取出相应的值
        …
        ;下表为 0--F 的字模
DISR_TAB    DCD     0xC0,0xF9,0xA4,0x99,0x92
            DCD     0x82,0xF8,0x80,0x90,0x88,0x83
            DCD     0xC6,0xa1,0x86,0x8E,0xFF
```

## 长跳转

ARM 的 B 和 BL 指令不能全空间跳转,但通过对 PC 进行赋值,实现 32 位地址的跳转/调用,示例如下：

```
        ADD     LR,PC,#4        ;保存返回地址,即 RET_FUN
        LDR     PC,[PC,#-4]     ;跳转到 LADR_FUN
        DCD     LADR_FUN
```

```
RET_FUN  …
```

也可使用伪指令 LDR  PC,=LADR_FUN 实现长跳转。

## 对信号量的支持

ARM 提供一条内存与寄存器交换的指令 SWP 用于支持信号量的操作,实现系统任务之间的同步或互斥,其使用的例子如下：

```
DISP_SEM   EQU   0x40002A00
           …
DISP_WAIT  MOV   R1,#0
           LDR   R0,=DISP_SEM
           SWP   R1,R1[R0]        ;取出信号量,并设置其为 0
           CMP   R1,#0            ;判断是否有信号
           BEQ   DISP_WAIT        ;若没有信号,则等待
           …
```

## 伪指令使用

LDR 伪指令和 NOP 伪指令应用例子代码如下：

```
LDR  R1,=0x12345678     ;加载 32 位立即数
LDR  R0,=LDE_TAB        ;加载标号地址
NOP                     ;空指令
B                       ;死循环
```

## 一个完整的例子

下面是汇编程序完整的例子：

```
ABC  EQU  0x12
     ;声明一个代码段 Example
     AREA  Example,CODE,READONLY
     ENIRY
```

```
        CODE32
        ADR   R0,Thumb_START+1      ;装载地址,并设置 d0 位为 1
        BX    R0                    ;切换到 Thumb 状态
        CODE16                      ;声明 16 位代码(Thumb)
Thumb_START
        MOV   R1,#ABC
        ADD   R1,R1,#0x10
        B     Thumb_START
        END
```

## 外围部件控制

在 32 位的 ARM 核芯片中,其外围部件的控制寄存器中,一般会设置“置位/复位”寄存器,这样可以方便的实现位操作,而不会影响其它位,如 IOSET=0x01 只会将 P0.1 的置位,而其它 I/O 状态不变,另外,ARM 存储/保存指令具有前偏移功能,所以对外围部件的控制寄存器进行操作时可使用此功能,避免了每次都加载寄存器地址的操作示例如下：

```
        LDR   R0,=GPIO_BASE
        MOV   R1,#0x00
        STR   R1,[R0,#0x04]       ;IOSET=0x00
        MOV   R1,#0x10
        STR   R1,[R0,#0x0C]       ;IOCLR=0x101
```

## 三级流水线介绍

ARM7TDM(-S)使用三级流水线执行指令,第一阶段从内存中取回的指令,第二阶段开始解码,而第三阶段实际执行指令.故此,程序计数器总是超出当前执行的指令两条指令.(在为跳转指令计算偏移量时必须计算在内).因为有这个流水线,在跳转时丢失 2 个指令周期(因为要重新添满流水线).所以最好利用条件执行指令来避免浪费周期.

条件跳转示例：

```
…
```

```
    CMP    R0,#0
    BEQ    LOOP1
    MOV    R1,#0x10
    MOV    R2,#0x88
LOOP1
    …
```

可以写为更有效的:

```
…
CMP     R0,#0
MOVNE   R1,#0x10
MOVNE   R2,#0x88
…
```

## C 与汇编混合编程

在需要 C 与汇编混合编程时,若汇编代码较结,则可使用直接内嵌汇编的方法混合编程;否则,可以将汇编文件以文件的形式加入项目中,通过 ATPCS 规定与 C 程序相互调用及访问.

ATPCS,即 ARM,Thumb 过程调用标准(ARM/Thumb Procedure Call Standard),它规定了一些子程序间调用的基本规则,如子程序调用过程中的寄存器的使用规则,堆栈的使用规则,参数的传递规则等.

## 内嵌汇编

在 C 程序嵌入汇编程序,可以实现一些高级语言没有的功能,提高程序执行效率.armcc 编译器的内嵌汇编器支持,ARM 指令集,tcc 编译器的内嵌汇编支持 Thumb 指令集.

内嵌汇编的语法：

```
__asm
{
```

```
    指令[;指令]    /*注释*/
    …
   [指令]
}
```

嵌入汇编程序的例子如下所示，其中 enable_IRQ 函数为使能 IRQ 中断，而 disable_IRQ 函数为关闭 IRQ 中断.

使能/禁能 IRQ 中断：

```
__inline  void  enable_IRQ(void)
{
   int  tmp
   _asm                          //嵌入汇编代码
   {
     MRS  tmp,CPSR               //读取 CPSR 的值
     BIC  tmp,tmp,#0x80          //将 IRQ 中断禁止位 I 清零,即允许 IRQ 中断
     MSR
     CPSR_c,tmp                  //设置 CPSR 的值
   }
 }
  __inline  void  disable_IRQ(void)
 {
     int  tmp;
     _asm
    {
        MRS  tmp,CPSR
        ORR  tmp,tmp,#0x80
        MSR  CPSR_c,tmp
     }
  }
```

另外一个嵌入汇编程序的例子如下所示，其中 my_strcpy 函数是字符串复制函

数,src 为源字符串指针,dst 为目标字符串指针。

制操作全部由嵌入的汇编代码实现.在主程序中,可以使用 my_strcpy(a,b)来调用函数,还可以使用嵌入汇编方法进行调用,嵌入汇编的代码先要设置入口参数 R0,R1,然后使用 BL my_strcpy ,{R0,R1}指令调用函数,其中,输入寄存器列表为{R0,R1},没有输出寄存器列表.

字符串复制：

```
#include  <stdio.h>
 void  my_strcpy(const char*src, char*dst)
{
    int   ch;
    _asm
     {
        loop:
            #ifndef_thumb
                //ARM 指令版本
                LDRB  ch,[src],#1
                STRB  ch,[dst],#1
            #else
                //Thumb 指令版本
                LDRB  ch,[src]
                ADD   src,#1
                STRB  ch,[dst]
                ADD   dst,#1
            #endif
            CMP    ch,#0
            BNE    loop
     }
 }
int  main(void)
```

```
{
  const  char*a=“Hello world!”
  char   b[20]
  //my_strcpy(a,b);
  _asm
       {
           MOV  R0,a                 //设置入口参数
           MOV  R1,b
           BL   my_strcpy,{R0,R1}  //调用 my_strcpy()函数
       }
  printf(“Original string:’ %s’ \n,” a); //显示 my_strcpy()函数字符串复制结果
  printf(“Copied  string:’ %s’ \n,” b);
  return(0);
}
```

## 内嵌汇编的指令用法

操作数. 内嵌的汇编指令中作为操作数的寄存器和常量可以是表达式. 这些表达式可以是 char, short 或 int 类型, 而且这些表达式都是作为无符号数进行操作. 若需要带符号数, 用户需要自己处理与符号有关的操作. 编译器将会计算这些表达式的值, 并为其分配寄存器.

**物理寄存器**. 内嵌汇编中使用物理寄存器有以下限制;

- 不能直接向 PC 寄存器赋值, 程序跳转只能使用 B 或 BL 指令实现
- 使用物理寄存器的指令中, 不要使用过于复杂的C表达式. 因为表达式过于复杂时, 将会需要较多的物理寄存器. 这些寄存器可能与指令中的物理寄存器使用冲突.
- 编译器可能会使用 R12 或 R13 存放编译的中间结果, 在计算表达式的值时可能会将寄存器 R0~R3, R12 和 R14 用于子程序调用. 因此在内嵌的汇编指令中, 不要将这些寄存器同时指定为指令中的物理寄存器.
- 通常内嵌的汇编指令中不要指定物理寄存器, 因为这可能会影响编译器分配寄

存器,进而影响代码的效率.

**常量**.在内嵌汇编指令中,常量前面的”#”可以省略.

**指令展开**.内嵌汇编指令中,如果包含常量操作数,该指令有可能被内嵌汇编器展开成几条指令.

**标号**.C 程序中的标号可以被内嵌的汇编指令使用,但是只有指令 B 可以使用 C 程序中的标号,而指令 BL 则不能使用.

**内存单元的分配**.所有的内存分配均由 C 编译器完成,分配的内存单元通过变量供内嵌汇编器使用.内嵌汇编器不支持内嵌汇编程序中用于内存分配的伪指令.

**SWI 和 BL 指令**.在内嵌的 SWI 和 BL 指令中,除了正常的操作数域外,还必须增加以下 3 个可选的寄存器列表:

- 第 1 个寄存器列表中的寄存器用于输入的参数.
- 第 2 个寄存器列表中的寄存器用于存储返回的结果
- 第 3 个寄存器列表中的寄存器的内容可能被被调用的子程序破坏,即这些寄存器是供被调用的子程序作为工作寄存器

## 内嵌汇编器与 armasm 汇编器的差异

内嵌汇编器不支持通过“.”指示符或 PC 获取当前指令地址;不支持 LDR Rn,=expr 伪指令,而使用 MOV Rn,expr 指令向寄存器赋值;不支持标号表达式;不支持 ADR 和 ADRL 伪指令;不支持 BX 指令;不能向 PC 赋值.

使用 0x 前缀代替”&”,表示十六进制数.使用 8 位移位常数导致 CPSR 的标志更新时,N、Z、C 和 V 标志中的 C 不具有真实意义.

## 内嵌汇编注意事项

✓ 必须小心使用物理寄存器,如 R0~R3,IP,LR 和 CPSR 中的 N,Z,C,V 标志位.因为计算汇编代码中的 C 表达式时,可能会使用这些物理寄存器,并会修改 N,Z,C,V 标志位.如:

```
__asm
  {  MOV  R0,x
```

```
        ADD   y,R0,x/y       //计算 x/y 时 R0 会被修改
    }
```

在计算 x/y 时 R0 会被修改,从而影响 R0+x/y 的结果.用一个 C 程序的变量代替 R0 就可以解决这个问题:

```
__asm
{
   MOV   var,x
   ADD   y,var,x/y
 }
```

内嵌汇编器探测到隐含的寄存器冲突就会报错.

✓ 不要使用寄存器代替变量.尽管有时寄存器明显对应某个变量,但也不能直接使用寄存器代替变量.

```
int   bad_f(int x)     //x 存放在 R0 中
{
    __asm
       {
            ADD   R0,R0,#1 //发生寄存器冲突,实际上 x 的值没有变化
        }
   return(x);
}
```

尽管根据编译器的编译规则似乎可以确定 R0 对应 x,但这样的代码会使内嵌汇编器认为发生了寄存器冲突.用其他寄存器代替 R0 存放参数 x,使得该函数将 x 原封不动地返回.

这段代码的正确写法如下:

```
int   bad_f(int x)
{    __asm
     {
        ADD    x,x,#1
     }
```

```
    return(x)
}
```

✓ 使用内嵌式汇编无需保存和恢复寄存器.事实上,除了 CPSR 和 SPSR 寄存器,对物理寄存器先读后写都会引起汇编器报错.例如.:

```
int  f(int x)
{   __asm
    {
     STMFD   SP!{R0}   //保存 R0.先读后写,汇编出错
     ADD     R0,x,1
     EOR     x,R0,x
     LDMFD   SP!,{R0}
     }
  returnt(x):
}
```

LDM 和 STM 指令的寄存器列表中只允许使用物理寄存器.内嵌汇编可以修改处理器模式,协处理器模式和FP,SL,SB等APCS寄存器.但是编译器在编译时并不了解这些变化,所以必须保证在执行 C 代码前恢复相应被修改的处理器模式.

✓ 汇编语言中的”.”号作为操作数分隔符号.如果有C表达式作为操作数,若表达式包含有“.”必须使用“(”号和“)”号将其归纳为一个汇编操作数。例如：

```
_asm
{
    ADD x,y,(f(),z)    //“f(),z”为一个带有“.”的 C 表达式
}
```

## 访问全局变量

使用 IMPORT 伪指令引入全局变量,并利用 LDR 和 STR 指令根据全局变量的地址访问它们,对于不同类型的变量,需要采用不同选项的 LDR 和 STR 指令:

```
unsigned  char     LDRB/STRB
unsigned  short    LDRH/STRH
```

| | |
|---|---|
| unsingned int | LDR/STR |
| char | LDRSB/STRSB |
| short | LDRSH/STRSH |

对于结构,如果知道各个数据项的偏移量,可以通过存储/加载指令访问.如果结构所占空间小于 8 个字,可以使用 LDM 和 STM 一次性读写.

下面例子是一个汇编代码的函数,它读取全局变量 globval,将其加 1 后写回.

访问 C 程序的全局变量：

```
        AREA    globats,CODE,READONLY]
        EXPORT  asmsubroutime
        IMPORt  glovbvar                ;声明外部变量 glovbvar
asmsubroutime
        LDR     R1,=glovbvar            ;装载变量地址
        LDR     R0,[R1]                 ;读出数据
        ADD     R0,R0,#1                ;加 1 操作
        STR     R0,[R1]                 ;保存变量值
        MOV     PC LR
        END
```

## C 与汇编相互调用

在 C 程序和 ARM 汇编程序之间相互调用必须遵守 ATPCS.使用 ADS 的 C 语言编译器编译的C语言子程序满足用户指定的ATPCS类型.而对于汇编语言来说,完全要依赖用户来保证各个子程序满足选定的 ATPCS 类型.具体来说,汇编语言子程序必须满足下面 3 个条件：

- ✓ 在子程序编写时必须遵守相应的 ATPCS 规则
- ✓ 堆栈的使用要遵守相应的 ATPCS 规则.
- ✓ 在汇编编译器中使用-apcs 选项

基本 ATPCS 规定了在子程序调用时的一些基本规则,包括:各寄存器的使用规则及其相应的名称,堆栈的使用规则,参数传送的规则.

## 寄存器的使用规则

✓ 子程序间通过寄存器 R0~R3 来传递参数.这时,寄存器 R0~R3 可记作 A0~A3.被调用的子程序在返回前无须恢复寄存器 R0~R3 的内容.

✓ 在子程序中,使用寄存器 R4~R11 来保存局部变量.这时,寄存器 R4~R11 可以记作 V1~V8.如果在子程序中使用了寄存器 V1~V8 中的某些寄存器,子程序进入时必须保存这些寄存器的值,在返回前必须恢复这些寄存器的值.在 Thumb 程序中,通常只能使用寄存器 R4~R7 来保存局部变量.

✓ 寄存器 R12 用作过程调用中间临时寄存器,记作 IP.在子程序间的连接代码段中常有这种使用规则.

✓ 寄存器 R13 用作堆栈指针,记作 SP.在子程序中寄存器 R13 不能作其他用途.寄存器 SP 在进入子程序时的值和退出子程序时的值必须相等.

✓ 寄存器 R14 称为连接寄存器,记作 LR.它用于保存子程序的返回地址.如果在子程序中保存了返回地址,寄存器 R14 则可以用作其他用途.

✓ 寄存器 R15 是程序计数器,记作 PC.它不能用作其它用途.

## 堆栈使用规则

ATPCS 规定堆栈为 FD 类型,即满递减堆栈,并且对堆栈的操作是 8 字节对齐.

使用 ADS 中的编译器产生的目标代码中包含了 DRAFT2 格式的数据帧.在调试过程中,调试器可以使用这些数据帧来查看堆栈中的相关信息.对于汇编语言来说,用户必须使用 FRAME 伪指令来描述堆栈的数据帧.ARM 汇编器根据这些伪指令在目标文件中产生相应的 DRAFT2 格式的数据帧.(堆栈中的数据帧---在堆栈中,为子程序分配的用来保存寄存器和局部变量的区域).对于汇编程序来说,如果目标文件中包含了外部调用,则必须满足下列条件:

✓ 外部接口的堆栈必须是 8 字节对齐的.

✓ 在汇编程序中使用 PRESERVE8 伪指令告诉连接器,本汇编程序数据是 8 字节对齐的.

## 参数传递规则

根据参数个数是否固定可以将子程序分为参数个数固定的子程序和参数个数可变化的子程序.这两种子的参数传递规则是不一样的.

✓ 参数个数可变的子程序参数传递规则

对于参数个数可变的子程序,当参数不超过4个时,可以使用寄存器R0~R3来传递参数;当参数超过 4 个时,还可以使用堆栈来传递参数.

在参数传递时,将所有参数看作是存放在连续的内存字单元的字数据.然后,依次将各字数据传送到寄存器R0,R1,R2,R3中,如果参数多于4个,将剩余的字数据传送堆栈中,入栈的顺序与参数顺序相反,即最后一个字数据先入栈.

按照上面的规则,一个浮点数参数可以通过寄存器传递,也可以通过堆栈传递,也可能一半通过寄存器传递,另一半通过堆栈传递.

✓ 参数个数固定的子程序参数传递规则

对于参数个数固定的子程序,参数传递与参数个数可变的子程序参数传递规则不同.

如果系统包含浮点运算的硬件部件,浮点参数将按下面的规则传递;

各个浮点参数按顺序处理;

为每个浮点参数分配 FP 寄存器;

分配的方法是,满足该浮点参数需要的且编号最小的一组连续的 FP 寄存器

第一个整数参数,通过寄存器 R0~R3 来传递.其他参数通过堆栈传递.

✓ 子程序结果返回规则

子程序中结果返回的规则如下;

结果为一个 32 位的整数时,可以通过寄存器 R0 返回;

结果为一个 64 位的整数时,可以通过寄存器 R0 和 R1 返回;

结果为一个浮点数时,可以通过浮点运算部件的寄存器 f0,d0 或 s0 来返回;

结果为复合型的浮点(如复数)时,可以通过寄存器 f0~fnA 或 d0~dn 来返回;

对于位数更多的结果,需要通过内存来传递.

## C 程序调用汇编程序

汇编程序的设置要遵循 ATPCS 规则,保证程序调用时参数的正确传递.

✓ 在汇编程序中使用 EXPORT 伪指令声明本子程序,使其它程序可以调用此子程序.

✓ 在 C 语言程序中使用 extern 关键字声明外部函数(声明要调用的汇编子程序),即可调用此汇编子程序.

如以下程序所示,汇编子程序 strcopy 使用两个参数,一个表示目标字符串地址,一个表示源字符串的地址,参数分别存放 R0,R1 寄存器中.

**调用汇编的 C 函数：**

```
#include  <stdio.h>
extern    void  strcopy(char*d,const char*s) //声明外部函数,即要调用的汇编子程序
int  mian(void)
{
   const  char *srcstr=“First string-source”;     //定义字符串常量
   char   dstsrt[] =“Second string-destination”; //定义字符串变量
   printf(“Before copying：\n”);
   printf(“’ %s’ \n  ‘%s\n,” srcstr,dststr); //显示源字符串和目标字符串的内容
   strcopy(dststr,srcstr);       //调用汇编子程序,R0=dststr,R1=srcstr
   printf(“After copying：\n”)
   printf(“’ %s’ \n  ‘%s\n,” srcstr,dststr); //显示 strcopy 复制字符串结果
   return(0);
}
```

**被调用汇编子程序：**

```
      AREA      SCopy,CODE,READONLY
      EXPORT    strcopy   ;声明 strcopy,以便外部程序引用
strcopy
      ;R0 为目标字符串的地址
      ;R1 为源字符串的地址    ;
```

```
LDRB   R2,[R1],#1       ;读取字节数据,源地址加 1
STRB   R2,[R0],#1       ;保存读取的 1 字节数据,目标地址加 1
CMP    r2,#0            ;判断字符串是否复制完毕
BNE    strcopy          ;没有复制完毕,继续循环
MOV    pc,lr            ;返回
END
```

## 汇编程序调用 C 程序

汇编程序的设置要遵循 ATPCS 规则,保证程序调用时参数的正确传递.

在汇编程序中使用 IMPORT 伪指令声明将要调用的 C 程序函数.

在调用 C 程序时,要正确设置入口参数,然后使用 BL 调用.

以下程序清单所示,程序使用了 5 个参数,分别使用寄存器 R0 存储第 1 个参数,R1 存储第 2 个数,R2 存储第 3 个参数,R3 存储第 4 个参数,第 5 个参数利用堆栈传送.由于利用了堆栈传递参数,在程序调用用结果后要调整堆栈指针

**汇编调用 C 程序的 C 函数：**

```
/*函数 sum5()返回 5 个整数的和*/
int sum5(int  a,lit  b,  int  c,int  d,int  e)
{
   return(a+b+c+d+e);     //返回 5 个变量的和
}
```

**汇编调用 C 程序的汇编程序：**

```
  EXPORT    CALLSUM5
  AREA   Example,  CODE,READONLY
  IMPORT   sum5            ;声明外部标号 sum5,即 C 函数 sum5()
CALLSUMS
  STMFD    SP!{LR}         ;LR 寄存器放栈
  ADD      R1,R0,R0        ;设置 sum5 函数入口参数,R0 为参数 a
  ADD      R2,R1,R0        ;R1 为参数 b,R2 为参数 c
  ADD      R3,R1,R2,
```

```
STR       R3,[SP,#-4]!      ;参数 e 要通过堆栈传递
ADD       R3,R1,R1          ;R3 为参数 d
BL        sum5              ;调用 sum5(),结果保存在 R0
ADD       SP,SP#4           ;修正 SP 指针
LDMFD     SP,{PC            ;子程序返回
END
```

# ARM 指令集列表

## ARM 存储器访问指令表列表

| 助记符 | 说明 | 操作 | 条件码位置 |
|---|---|---|---|
| LDR Rd,addressing | 加载字数据 | Rd←[addressing],addressing 索引 | LDR{cond} |
| LDRB Rd,addressing | 加载无符字节数据 | Rd←[addressing],addressing 索引 | LDR{cond}B |
| LDRT Rd，addressing | 以用户模式加载字数据 | Rd←[addressing],addressing 索引 | LDR{cond}T |
| LDRBT Rd，addressing | 以用户模式加载无符号字数据 | Rd←[addressing],addressing 索引 | LDR{cond}BT |
| LDRH Rd，addressing | 加载无符半字数据 | Rd←[addressing],addressing 索引 | LDR{cond}H |
| LDRSB Rd，addressing | 加载有符字节数据 | Rd←[addressing],addressing 索引 | LDR{cond}SB |
| LDRSH Rd，addressing | 加载有符半字数据 | Rd←[addressing],addressing 索引 | LDR{cond}SH |
| STR Rd，addressing | 存储字数据 | [addressing]←Rd,addressing 索引 | STR{cond} |
| STRB Rd，addressing | 存储字节数据 | [addressing]←Rd,addressing 索引 | STR{cond}B |
| STRT Rd，addressing | 以用户模式存储字数据 | [addressing]←Rd,addressing 索引 | STR{cond}T |
| STRBT Rd，addressing | 以用户模式存储字节数据 | [addressing]←Rd,addressing 索引 | STR{cond}BT |
| STRH Rd，addressing | 存储半字数据 | [addressing]←Rd,addressing 索引 | STR{cond}H |
| LDM{mode} Rn{!},reglist | 批量(寄存器)加载 | reglist← [Rn…],Rn 回存等 | LDM{cond}{more} |
| STM{mode} Rn{!},rtglist | 批量(寄存器)存储 | [Rn…]← reglist,Rn 回存等 | STM{cond}{more} |
| SWP Rd,Rm,Rn | 寄存器和存储器字数据交换 | Rd←[Rd],[Rn]←[Rm](Rn≠Rd 或 Rm) | SWP{cond} |
| SWPB Rd,Rm,Rn | 寄存器和存储器字节数据交换 | Rd←[Rd],[Rn]←[Rm](Rn≠Rd 或 Rm) | SWP{cond}B |

# ARM 数据处理指令列表

| 助记符号 | 说明 | 操作 | 条件码位置 |
|---|---|---|---|
| MOV Rd ,operand2 | 数据转送 | Rd←operand2 | MOV {cond}{S} |
| MVN Rd ,operand2 | 数据非转送 | Rd←(operand2) | MVN {cond}{S} |
| ADD Rd,Rn operand2 | 加法运算指令 | Rd←Rn+operand2 | ADD {cond}{S} |
| SUB Rd,Rn operand2 | 减法运算指令 | Rd←Rn-operand2 | SUB {cond}{S} |
| RSB Rd,Rn operand2 | 逆向减法指令 | Rd←operand2-Rn | RSB {cond}{S} |
| ADC Rd,Rn operand2 | 带进位加法 | Rd←Rn+operand2+carry | ADC {cond}{S} |
| SBC Rd,Rn operand2 | 带进位减法指令 | Rd←Rn-operand2-(NOT)Carry | SBC {cond}{S} |
| RSC Rd,Rn operand2 | 带进位逆向减法指令 | Rd←operand2-Rn-(NOT)Carry | RSC {cond}{S} |
| AND Rd,Rn operand2 | 逻辑与操作指令 | Rd←Rn&operand2 | AND {cond}{S} |
| ORR Rd,Rn operand2 | 逻辑或操作指令 | Rd←Rn\|operand2 | ORR {cond}{S} |
| EOR Rd,Rn operand2 | 逻辑异或操作指令 | Rd←Rn ^ operand2 | EOR {cond}{S} |
| BIC Rd,Rn operand2 | 位清除指令 | Rd←Rn&(～operand2) | BIC {cond}{S} |
| CMP Rn,operand2 | 比较指令 | 标志 N、Z、C、V←Rn-operand2 | CMP {cond} |
| CMN Rn,operand2 | 负数比较指令 | 标志 N、Z、C、V←Rn＋operand2 | CMN {cond} |
| TST Rn,operand2 | 位测试指令 | 标志 N、Z、C、V←Rn＆operand2 | TST {cond} |
| TEQ Rn,operand2 | 相等测试指令 | 标志 N、Z、C、V←Rn ^ operand2 | TEQ {cond} |

# ARM 乘法指令列表

| 助记符 | 说明 | 操作 | 条件码位置 |
|---|---|---|---|
| MUL Rd,Rm,Rs | 32 位乘法指令 | Rd←Rm*Rs (Rd≠Rm) | MUL{cond}{S} |
| MLA Rd,Rm,Rs,Rn | 32 位乘加指令 | Rd←Rm*Rs+Rn (Rd≠Rm) | MLA{cond}{S} |
| UMULL RdLo,RdHi,Rm,Rs | 64 位无符号乘法指令 | (RdLo,RdHi)←Rm*Rs | UMULL{cond}{S} |
| UMLAL RdLo,RdHi,Rm,Rs | 64 位无符号乘加指令 | (RdLo,RdHi)←Rm*Rs+(RdLo,RdHi) | UMLAL{cond}{S} |
| SMULL RdLo,RdHi,Rm,Rs | 64 位有符号乘法指令 | (RdLo,RdHi)←Rm*Rs | SMULL{cond}{S} |
| SMLAL RdLo,RdHi,Rm,Rs | 64 位有符号乘加指令 | (RdLo,RdHi)←Rm*Rs+(RdLo,RdHi) | SMLAL{cond}{S} |

# ARM 跳转指令列表

| 助记符 | 说明 | 操作 | 条件码位置 |
|---|---|---|---|
| B label | 跳转指令 | Pc←label | B{cond} |
| BL label | 带链接的跳转指令 | LR←PC-4, PC←label | BL{cond} |
| BX Rm | 带状态切换的跳转指令 | PC←label,切换处理状态 | BX{cond} |

# ARM 协处理器指令列表

| 助记符 | 说明 | 操作 | 条件码位置 |
|---|---|---|---|
| CDP coproc,opcode1,CRd,CRn,CRm{,opcode2} | 协处理器数据操作指令 | 取决于协处理器 | CDP{cond} |
| LDC{L} coproc,CRd〈地址〉 | 协处理器数据读取指令 | 取决于协处理器 | LDC{cond}{L} |
| STC{L} coproc,CRd,〈地址〉 | 协处理器数据写入指令 | 取决于协处理器 | STC{cond}{L} |
| MCR coproc, opcode1,Rd,CRn,{,opcode2} | ARM 寄存器到协处理器寄存器的数据传送指令 | 取决于协处理器 | MCR{cond} |
| MRC coproc, opcode1,Rd,CRn,{,opcode2} | 协处理器寄存器到 ARM 寄存器到数据传送指令 | 取决于协处理器 | MCR{cond} |

# ARM 杂项指令列表

| 助记符 | 说明 | 操作 | 条件码位置 |
|---|---|---|---|
| SWI immed_24 | 软中断指令 | 产生软中断，处理器进入管理模式 | SWI{cond} |
| MRS Rd,psr | 读状态寄存器指令 | Rd←psr,psr 为 CPSR 或 SPSR | MRS{cond} |
| MSR psr_fields,Rd/#immed_8r | 写状态寄存器指令 | psr_fields←Rd/#immed_8r,psr 为 CPSR 或 SPSR | MSR{cond} |

# ARM 伪指令列表

| 伪指令助记符 | 说明 | 操作 | 条件码位置 |
|---|---|---|---|
| ADR register,expr | 小范围的地址读取伪指令 | register<-expr 指向的地址 | ADR{cond} |
| ADRL register,expr | 中等范围的地址读取伪指令 | register<-expr 指向的地址 | ADR{cond} |
| LDR register,=expr/label-expr | 大范围的地址读取伪指令 | register<-expr/label-expr 指定的数据/地址 | LDR{cond} |
| NOP | 空操作伪指令 | 无 | 无 |

# Thumb 指令集列表

## Thumb 存储器访问指令列表

| 助记符 | 说明 | 操作 | 影响标志 |
|---|---|---|---|
| LDR Rd,[Rn,#immed_5×4] | 加载字数据 | Rd←[Rm,#immed_5×4],Rd,Rn 为 R0～R7 | 无 |
| LDRH Rd,[Rn,#immed_5×2] | 加载无符半字数据 | Rd←[Rm,#immed_5×2],Rd,Rn 为 R0～R7 | 无 |
| LDRB Rd,[Rn,#immed_5×1] | 加载无符字节数据 | Rd←[Rm,#immed_5×1],Rd,Rn 为 R0～R7 | 无 |
| STR Rd,[Rn,#immed_5×4] | 存储字数据 | Rn,#immed_5×4Rd←Rd,Rn 为 R0～R7 | 无 |
| STRH Rd,[Rn,#immed_5×2] | 存储无符半辽数据 | Rn,#immed_5×2]Rd←Rd,Rn 为 R0～R7 | 无 |
| STRB Rd,[Rn#immed_5×1] | 存储无符字节数据 | Rn,#immed_5×1]Rd←Rd,Rn 为 R0～R7 | 无 |
| LDR Rd,[Rn,Rm] | 加载字数据 | Rd←[Rn,Rm],Rd,Rn,Rm 为 R0～R7 | 无 |
| LDRH Rd,[Rn,Rm] | 加载无符半字数据 | Rd←[Rn,Rm],Rd,Rn,Rm 为 R0～R7 | 无 |
| LDRB Rd,[Rn,Rm] | 加载无符字节数据 | Rd←[Rn,Rm],Rd,Rn,Rm 为 R0～R7 | 无 |
| LDRSH Rd[Rn,Rm] | 加载有符半字数据 | Rd←[Rn,Rm],Rd,Rn,Rm 为 R0～R7 | 无 |
| LDRSB Rd[Rn,Rm] | 加载有符字节数据 | Rd←[Rn,Rm],Rd,Rn,Rm 为 R0～R7 | 无 |
| STR Rd,[Rn,Rm] | 存储字数据 | [Rn,Rm]←Rd,Rd,Rn,Rm 为 R0～R7 | 无 |
| STRH Rd,[Rn,Rm] | 存储无符半字数据 | [Rn,Rm]←Rd,Rd,Rn,Rm 为 R0～R7 | 无 |
| STRB Rd,[Rn,Rm] | 存储无符字节数据 | [Rn,Rm]←Rd,Rd,Rn,Rm 为 R0～R7 | 无 |
| LDR Rd,[PC,#immed_8×4] | 基于 PC 加载字数据 | Rd←{PC,#immed_8×4]Rd 为 R0～R7 | 无 |
| LDR Rd,label | 基于 PC 加载字数据 | Rd←[label],Rd 为 R0～R7 | 无 |
| LDR Rd,[SP,#immed_8×4] | 基于 SP 加载字数据 | Rd←{SP,#immed_8×4]Rd 为 R0～R7 | 无 |
| STR Rd,[SP,#immed_8×4] | 基于 SP 存储字数据 | {SP,#immed_8×4]←Rd,Rd 为 R0～R7 | 无 |
| LDMIA Rn{!}reglist | 批量(寄存器)加载 | regist←[Rn…],Rn 回存等(R0～R7) | 无 |
| STMIA Rn{!}reglist | 批量(寄存器)加载 | [Rn…]←reglist,Rn 回存等(R0～R7) | 无 |
| PUSH {reglist[,LR]} | 寄存器入栈指令 | [SP…]←reglist[,LR],SP 回存等(R0～R7,LR) | 无 |
| POP {reglist[,PC]} | 寄存器入栈指令 | reglist[,PC]←[SP…],SP 回存等(R0～R7,PC) | 无 |

# Thumb 数据处理指令列表

| 助记符 | 说明 | 操作 | 影响标志 |
|---|---|---|---|
| MOV Rd,#expr | 数据转送 | Rd←expr,Rd 为 R0～R7 | 影响 N,Z |
| MOV Rd,Rm | 数据转送 | Rd←Rm,Rd、Rm 均可为 R0～R15 | RdT 和 Rm 均为 R0～R7 时,影响 N,Z,清零 C,V |
| MVN Rd,Rm | 数据非传送指令 | Rd←（～Rm),Rd,Rm 均为 R0～R7 | 影响 N,Z |
| NEG Rd,Rm | 数据取负指令 | Rd←(-Rm),Rd,Rm 均为 R0～R7 | 影响 N,Z,C,V |
| ADD Rd.Rn,Rm | 加法运算指令 | Rd←Rn+Rm,Rd,Rn,Rm 均为 R0～R7 | 影响 N,Z,C,V |
| ADD Rd.Rn,#expr3 | 加法运算指令 | Rd←Rn+expr#,Rd,Rn 均为 R0～R7 | 影响 N,Z,C,V |
| ADD Rd,#expr8 | 加法运算指令 | Rd←Rd+expr8,Rd 为 R0～R7 | 影响 N,Z,C,V |
| ADD Rd,Rm | 加法运算指令 | Rd←Rd+Rm,Rd,Rm 均可为 R0～R15 | Rd 和 Rm 均为 R0～R7 时,影响 N,Z,C,V |
| ADD Rd,Rp#expr | SP/PC 加法运算指令 | Rd←SP+expr 或 PC+expr,Rd 为 R0～R7 | 无 |
| ADD SP,#expr | SP 加法运算指令 | SP←SP+expr | 无 |
| SUB Rd,Rn,Rm | 减法运算指令 | Rd←Rn-Rm,Rd、Rn、Rm 均为 R0～R7 | 影响 N,Z,C,V |
| SUB Rd,Rn，#expr3 | 减法运算指令 | Rd←Rn-expr3，RdRn 均为 R0～R7 | 影响 N,Z,C,V |
| SUB Rd,#expr8 | 减法运算指令 | RD←Rd-expr8，Rd 为 R0～R7 | 影响 N,Z,C,V |
| SUB SP,#expr | SP 减法运算指令 | SP←SP-expr | 无 |
| ADC Rd,Rm | 带进位加法指令 | Rd←Rd+Rm+Carry，Rd、Rm 为 R0～R7 | 影响 N,Z,C,V |
| SBC Rd,Rm | 带位减法指令 | Rd←Rd-Rm-(NOT)Carry，Rd、Rm 为 R0～R7 | 影响 N,Z,C,V |
| MUL Rd,Rm | 乘法运算指令 | Rd←Rd*Rm,Rd、Rm 为 R0～R7 | 影响 N,Z, |
| AND Rd,Rm | 逻辑与操作指令 | Rd←Rd&Rm,Rd、Rm 为 R0～R7 | 影响 N,Z, |
| ORR Rd,Rm | 逻辑或操作指令 | Rd←Rd\|Rm,Rd、Rm 为 R0～R7 | 影响 N,Z, |
| EOR Rd,Rm | 逻辑异或操作指令 | Rd←Rd^Rm,Rd、Rm 为 R0～R7 | 影响 N,Z, |
| BIC Rd,Rm | 位清除指令 | Rd←Rd&(～Rm),Rd、Rm 为 R0～R7 | 影响 N,Z, |
| ASR Rd,Rs | 算术右移指令 | Rd←Rd 算术右移 Rs 位,Rd,Rs 为 R0～R7 | 影响 N,Z,C, |
| ASR Rd,Rm，#expr | 算术右移指令 | Rd←Rm 算术右移 expr 位,Rd、Rm 为 R0～R7 | 影响 N,Z,C, |
| LSL Rd,Rs | 逻辑左移指令 | Rd←Rd<<Rs,Rd、Rs 为 R0～R7 | 影响 N,Z,C, |
| LSL Rd,Rm,#expr | 逻辑左移指令 | Rd←Rm<<expr,Rd、Rm 为 R0～R7 | 影响 N,Z,C, |
| LSR Rd,Rs | 逻辑右移指令 | Rd←Rd>>Rs,Rd、Rs 为 R0～R7 | 影响 N,Z,C, |
| LSR Rd,Rm,#expr | 逻辑右移指令 | Rd←Rm>>mexpr,Rd、Rm 为 R0～R7 | 影响 N,Z,C, |
| ROR Rd,Rs | 循环右移指令 | Rd←Rm 循环右移 Rs 位,Rd、Rs 为 R0～R7 | 影响 N,Z,C, |
| CMP Rn,Rm | 比较指令 | 状态标←Rn-Rm,Rn、Rm 为 R0～R15 | 影响 N,Z,C,V |
| CMP Rn,#expr | 比较指令 | 状态标←Rn-expr,Rn 为 R0～R7 | 影响 N,Z,C,V |
| CMN Rn,Rm | 负数比较指令] | 状态标←Rn+Rm,Rn、Rm 为 R0～R7 | 影响 N,Z,C,V |
| TST Rn,Rm | 位测试指令 | 状态标←Rn&Rm,Rn、Rm 为 R0～R7 | 影响 N,Z,C,V |

# Thumb 跳转指令及软中断指令列表

| 助记符 | 说明 | 操作 | 条件码位置 |
|---|---|---|---|
| B label | 跳转指令 | PC←label | B{cond} |
| BL label | 带链接的跳转指令 | LR←PC←4,PC←label | 无 |
| BX Rm | 带状态切换的跳转指令 | PC←label 切换处理器状态 | 无 |
| SWI immed_8 | 软中断指令 | 产生软中断,处理器进入管理模式 | 无 |

# Thumb 伪指令列表

| 伪指令助记符 | 说明 | 操作 | 条件码位置 |
|---|---|---|---|
| ADR register,expr | 小范围的地址读取伪指令 | register<-expr 指向的地址 | 无 |
| LDR register,=expr/label-expr | 大范围的地址读取伪指令 | register<-expr/label-expr 指定的数据/地址 | 无 |
| NOP | 空操作伪指令 | 无 | 无 |

# 汇编预定义变量及伪指令

## 预定义的寄存器和协处理器名

ARM 汇编器对 ARM 的寄存器进行了预定义(包括 APCS 对 R0~R15 寄存器的定义), 所有的寄存器和协处理器名都是大小写敏感.预定义的寄存器如下:

### 通用寄存器

R0~R15 和 r0~r15(16 个通用寄存器);

a1~a4(参数,结果或临时寄存器,同 R0~R3);

v1~v8(变量寄存器,同 R4~R11);

SB 和 sb(静态基址,同 R9);

SL 和 sl(堆栈限制,同 R10);

FP 和 fp(帧指针);

IP 和 ip(过程调用中间临时寄存器,同 R12);

SP 和 sp(堆栈指针,同 R13);

LR 和 lr(链接寄存器,同 R14);

PC 和 pc(程序计数器,同 R15).

### 程序状态寄存器

CPSR 和 cpsr;

SPSR 和 spsr;

### 浮点数寄存器

F0~F7 和 f0~f7(FPA 寄存器);

S0~S7 和 s0~s7(VFP 单精度寄存器);

D0~D7 和 d0~d7(VFP 双精度寄存器);

## 协处理器及协处理器寄存器

p0~p15(协处理器 0~15)；

c0~c15(协处理器寄存器 0~15)；

## 内置变量列表

ARM 汇编器中定义了一些内置变量,如下表所示.这些内置变量不能使用伪指令设置(如 SETA,SETL,SETS),一般用于程序的条件汇编控制等,如下；

```
IF    {CONFIG}=16
...
ELSE
...
ENDIF
B                ;跳转到当前地址,即死循环
```

内置变量表

| 变量 | 说明 |
|---|---|
| {PC}或"." | 当前指令的地址 |
| {VAR}或"@" | 存储区位置计数器的当前值 |
| {TRUE} | 逻辑真 |
| {FALSE} | 逻辑假 |
| {OPT} | 当前设置列表选项值.OPT 用来保存当前列表选项,改变选项值,或恢复原始值 |
| {CONFIG} | 如果汇编器汇编 ARM 代码,则值为 32;若是汇编 Thumb 代码,则值为 16 |
| {ENDLAN} | 如果汇编器在大端模式下,则值为 big;若在小端模式下,否则为 little |
| {CODESIZE} | 如果汇编 Thumb 代码,则值为 16,否则为 32.同{CONFIG}变量 |
| {CPU} | 选定的 CPU 名,缺省为 ARM7TDMI.如果用命令行-cpu 选项,则为 genericARM |
| {FPU} | 设定的 FPU 名,缺省为 SoftVFP |
| {ARCHITECTURE} | 选定的 ARM 体系结构的值,如 3,3M,4,4T,4TxM |
| {PCSTOREOFFSET} | STR pc,[…]或 STR Rb,{…PC}指令的地址和 PC 的存储值之间的偏移量 |
| {ARMASM_VERSION}或\|ads$ version\| | ARM ASM 的版本号,为整数 |

# 伪指令列表

| 伪指令类型 | 伪指令 | 功能 |
|---|---|---|
| 符号定义指示符 | GBLA | 声明一个全局的算术变量，并将其初始化为 0 |
| | GBLL | 声明一个全局的逻辑变量，并将其初始化为{FALSE} |
| | GBLS | 声明一个全局的字符串变量，并将其初始化为空字符串"" |
| | LCLA | 声明一个局部的算术变量，并将其初始化为 0 |
| | LCLL | 声明一个局部的逻辑变量，并将其初始化为{FALSE} |
| | LCLS | 声明一个局部的字符串变量，并将其初始化为空字符串"" |
| | SETA | 给一个全局/局部的算术变量赋值 |
| | SETL | 给一个全局/局部的逻辑变量赋值 |
| | SETS | 给一个全局/局部的字符串变量赋值 |
| | RLIST | 为一个通用寄存器列表定义名称 |
| | CN | 给一个协处理器寄存器命名，协处理器寄存器编号为 0～15 |
| | CP | 为一个协处理器定义名称，协处理器编号为 0～15 |
| | DN | 为一个双精度的 VFP 寄存器定义名称 |
| | SN | 为一个单精度的 VFP 寄存器定义名称 |
| | FN | 为一个 FPA 浮点寄存器定义名称 |
| 数据定义指示符 | LTORG | 声明一个文字池 |
| | MAP或^ | 定义一个机构化的内存表首地址 |
| | FIELD或# | 定义机构化内存表中的一个数据域 |
| | SPACE或% | 分配一块内存空间，并用 0 初始化 |
| | DCB或= | 分配一段字节的内存单元，并用指定的数据初始化 |
| | DCD或& | 分配一段字的内存单元，并用指定的数据初始化 |
| | DCDU | 分配一段在字的内存单元，并用指定的数据初始化（不需要字对齐） |
| | DCDO | 分配一段字的内存单元，将每个单元的内容初始化为该单元相对于静态基地址寄存器的偏移量 |
| | DCFD | 分配一段双字的内存单元，并用双精度的浮点数初始化 |
| | DCFDU | 分配一段双字的内存单元，并用双精度的浮点数据初始化（不需要字对齐） |
| | DCFS | 分配一段字的内存单元，并用单精度的浮点数据初始化 |
| | DCFSU | 分配一段字的内存单元，并用单精度的浮点数据初始化（不需要字对齐） |
| | DCI | 分配一段字节的内存单元，用指定的数据初始化，指定内存单元存放的是代码而不是数据 |
| | DCQ | 分配一段双字的内存单元，并用 64 位的整数数据初始化 |
| | DCQU | 分配一段双字的内存单元，并用 64 位的整数数据初始化（不需要字对齐） |
| | DCW | 分配一段半字的内存单元，并用指定的数据初始化 |
| | DCWU | 分配一段半字的内存单元，并用指定的数据初始化（不需要字对齐） |
| 报告指示符 | ASSERT | 在汇编编译器对汇编程序的第二遍扫描中，如果 ASSERT 条件不成立，ASSERT 伪指令将报告该错误信息 |
| | INFO或! | 在汇编编译器对汇编程序的第一遍或第二遍扫描时报告诊断信息 |
| | OPT | 在源程序中设置列表选项 |

| | | |
|---|---|---|
| | TTL | 在列表文件中的每一页的开头插入一个标题 |
| | SUBT | 在列表文件中的每一页的开头插入一个子标题 |
| 汇编控制指示符 | IF或[ | IF、ELSE 和 ENDIF 伪指令能够根据条件把一段源代码包括在汇编源程序内或将其排除在外 |
| | ELSE或\| | |
| | ENDIF或] | |
| | MACRO | MACRO 和 MEND 伪指令用于宏定义 |
| | MEND | |
| | MEXIT | 退出宏定义 |
| | WHILE | WHILE 和 WEND 伪指令用于根据条件重复汇编相同的或几乎相同的一段源代码 |
| | WEND | |
| 杂项指示符 | ALIGN | 通过添加补丁字节使当前位置满足一定的对齐方式 |
| | AREA | 定义一个代码段或数据段 |
| | CODE16 | 指示汇编编译器后面的指令为 16 位的 Thumb 指令 |
| | CODE32 | 指示汇编编译器后面的指令为 32 位的 ARM 指令 |
| | END | 指示汇编编译器源文件已结束 |
| | ENTRY | 指定程序入口点 |
| | EQU或* | 为数字常量、基于寄存器的值和程序中的标号定义一个名称 |
| | EXPORT或 GLOBAL | 声明一个符号可以被其它文件引用。相当于生命了一个全局变量 |
| | IMPORT或 EXTERN | 指示编译器当前的符号不是在本源文件中定义的，而是在其它源文件中定义的，在本源文件中可能引用该符号 |
| | GET或 INCLUDE | 将一个源文件包含到当前源文件中，并将被包含的文件在其当前位置进行汇编处理 |
| | INCBIN | 将一个文件包含到当前源文件中，而被包含的文件不进行和汇编处理 |
| | KEEP | 指示编译器保留符号表中的局部符号 |
| | NOFP | 禁止源程序中包含浮点运算指令 |
| | REQUIRE | 指定段之间的依赖关系 |
| | REQUIRE8 | 指示当前文件请求堆栈为 8 字节对齐 |
| | PRESERVE8 | 指示当前文件保持堆栈为 8 字节对齐 |
| | RN | 给一个特殊的寄存器命名 |
| | ROUT | 定义局部标号的有效范围 |
| ARM 伪指令 | ADR | 小范围的地址读取伪指令 |
| | ADRL | 中等范围的地址读取伪指令 |
| | LDR | 大范围的地址读取伪指令 |
| | NOP | 空操作伪指令 |
| Thumb 伪指令 | ADR | 中等范围的地址读取伪指令 |
| | LDR | 大范围的地址读取伪指令 |
| | NOP | 空操作伪指令 |

# 指令条件码列表

| 条件码助记符 | 标志 | 含义 |
|---|---|---|
| EQ | Z＝1 | 相等 |
| NE | Z＝0 | 不相等 |
| CS/HS | C=1 | 无符号数大于或等于 |
| CC/LO | C=0 | 无符号数小于 |
| MI | N=1 | 负数 |
| PL | N=0 | 正数或零 |
| VS | V=1 | 溢出 |
| VC | V=0 | 没有溢出 |
| HI | C=1，Z=0 | 无符号数大于 |
| LS | C=0，Z=1 | 无符号数小于或等于 |
| GE | N=V | 带符号数大于或等于 |
| LT | N！=V | 带符号数小于 |
| GT | Z=0，N=V | 带符号数大于 |
| LE | Z=1，N！=V | 带符号数小于或等于 |
| AL | 任何 | 无条件执行（指令默认条件） |

# CPSR 和 SPSR 分配图

CPSR 或 SPSR 模式常量定义：

| CPSR_f 或 SPSR_f | | | | | | | | CPSR_s 或 SPSR_s | | | | | | | | CPSR_x 或 SPSR_x | | | | | | | | CPSR_c 或 SPSR_c | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 09 | 08 | 07 | 06 | 05 | 04 | 03 | 02 | 01 | 00 |
| **N** | **Z** | **C** | **V** | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | **I** | **F** | **T** | **M4** | **M3** | **M2** | **M1** | **M0** |
| 负数或小于 | 零 | 进位或借位或扩展 | 溢出 | 保留位 | | | | | | | | | | | | | | | | | | | | IRQ 禁止 1-禁止 0－允许 | FIQ 禁止 1-禁止 0－允许 | 状态位 0-ARM 1-Thumb | 模式位<br>10000-0x0010-用户<br>10001-0x0011-快速中断<br>10010-0x0012-中断<br>10011-0x0013 管理<br>10111-0x0017-未定义<br>11111-0x001F-系统 | | | | |