# 第1 章 ARM 微处理器概述

## 1.1 ARM－Advanced RISC Machines

ARM（Advanced RISC Machines ），既可以认为是一个公司的名字，也可以认为是对一类微处理器的通称，还可以认为是一种技术的名字。

## 1.2 ARM 微处理器的应用领域及特点

### 1.2.1 ARM 微处理器的应用领域

到目前为止，ARM 微处理器及技术的应用几乎已经深入到各个领域：

1、工业控制领域：作为32 的RISC 架构，基于ARM 核的微控制器芯片不但占据了高端微控制器市场的大部分市场份额，同时也逐渐向低端微控制器应用领域扩展，ARM 微控制器的低功耗、高性价比，向传统的8 位/16 位微控制器提出了挑战。

2、无线通讯领域：目前已有超过85% 的无线通讯设备采用了ARM 技术， ARM 以其高性能和低成本，在该领域的地位日益巩固。

3、网络应用：随着宽带技术的推广，采用ARM 技术的ADSL 芯片正逐步获得竞争优势。此外，ARM 在语音及视频处理上行了优化，并获得广泛支持，也对DSP 的应用领域提出了挑战。

4、消费类电子产品：ARM 技术在目前流行的数字音频播放器、数字机顶盒和游戏机中得到广泛采用。

5、成像和安全产品：现在流行的数码相机和打印机中绝大部分采用ARM 技术。手机中的32 位SIM 智能卡也采用了ARM 技术。

除此以外，ARM 微处理器及技术还应用到许多不同的领域，并会在将来取得更加广泛的应用。

### 1.2.2 ARM 微处理器的特点

采用RISC 架构的ARM 微处理器一般具有如下特点：1、体积小、低功耗、低成本、高性能；2、支持Thumb（16 位）/ARM（32 位）双指令集，能很好的兼容8 位/16 位器件；3、大量使用寄存器，指令执行速度更快；4、大多数数据操作都在寄存器中完成；5、寻址方式灵活简单，执行效率高；6、指令长度固定；

## 1.3 ARM 微处理器系列

ARM 微处理器目前包括下面几个系列，以及其它厂商基于ARM 体系结构的处理器，除了具有ARM 体系结构的共同特点以外，每一个系列的ARM 微处理器都有各自的特点和应用领域。

－ARM7 系列
－ARM9 系列
－ ARM9E 系列
－ ARM10E 系列
－SecurCore 系列
－Inter 的Xscale



PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

—Inter 的StrongARM 其中，ARM7、ARM9、ARM9E 和ARM10 为4 个通用处理器系列，每一个系列提供一套相对

独特的性能来满足不同应用领域的需求。SecurCore 系列专门为安全要求较高的应用而设计。以下我们来详细了解一下各种处理器的特点及应用领域。

## 1.3.1 ARM7 微处理器系列

ARM7 系列微处理器为低功耗的32 位RISC 处理器，最适合用于对价位和功耗要求较高的消费类应用。ARM7 微处理器系列具有如下特点：—具有嵌入式ICE—RT 逻辑，调试开发方便。—极低的功耗，适合对功耗要求较高的应用，如便携式产品。—能够提供0.9MIPS/MHz 的三级流水线结构。—代码密度高并兼容16 位的Thumb 指令集。—对操作系统的支持广泛，包括Windows CE 、Linux、Palm OS 等。—指令系统与ARM9 系列、ARM9E 系列和ARM10E 系列兼容，便于用户的产品升级换代。—主频最高可达130MIPS，高速的运算处理能力能胜任绝大多数的复杂应用。ARM7 系列微处理器的主要应用领域为：工业控制、Internet 设备、网络和调制解调器设备、移动电话等多种多媒体和嵌入式应用。

ARM7 系列微处理器包括如下几种类型的核：ARM7TDMI、ARM7TDMI-S、ARM720T 、ARM7EJ 。其中，ARM7TMDI 是目前使用最广泛的32 位嵌入式RISC 处理器，属低端ARM 处理器核。TDMI 的基本含义为：

T： 支持16 为压缩指令集Thumb；
D： 支持片上Debug；
M：内嵌硬件乘法器（Multiplier）
I： 嵌入式ICE，支持片上断点和调试点；

本书所介绍的Samsung 公司的S3C4510B 即属于该系列的处理器。

## 1.3.2 ARM9 微处理器系列

ARM9 系列微处理器在高性能和低功耗特性方面提供最佳的性能。具有以下特点：

—5 级整数流水线，指令执行效率更高。—提供1.1MIPS/MHz 的哈佛结构。—支持32 位ARM 指令集和16 位Thumb 指令集。—支持32 位的高速AMBA 总线接口。—全性能的MMU，支持Windows CE 、Linux、Palm OS 等多种主流嵌入式操作系统。

—MPU 支持实时操作系统。—支持数据Cache 和指令Cache，具有更高的指令和数据处理能力。ARM9 系列微处理器主要应用于无线设备、仪器仪表、安全系统、机顶盒、高端打印机、数字

照相机和数字摄像机等。ARM9 系列微处理器包含ARM920T 、ARM922T 和ARM940T 三种类型，以适用于不同的应用场合。

# ARM 微处理器结构

## RISC 体系结构

传统的CISC（Complex Instruction Set Computer ，复杂指令集计算机）结构有其固有的缺点，即随着计算机技术的发展而不断引入新的复杂的指令集，为支持这些新增的指令，计算机的体系结构会越来越复杂，然而，在CISC 指令集的各种指令中，其使用频率却相差悬殊，大约有20%的指令会被反复使用，占整个程序代码的80%。而余下的80%的指令却不经常使用，在程序设计中只占20%，显然，这种结构是不太合理的。

基于以上的不合理性，1979 年美国加州大学伯克利分校提出了RISC（Reduced Instruction Set



PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

Computer ，精简指令集计算机）的概念，RISC 并非只是简单地去减少指令，而是把着眼点放在了如何使计算机的结构更加简单合理地提高运算速度上。RISC 结构优先选取使用频最高的简单指令，避免复杂指令；将指令长度固定，指令格式和寻地方式种类减少；以控制逻辑为主，不用或少用微码控制等措施来达到上述目的。

到目前为止，RISC 体系结构也还没有严格的定义，一般认为，RISC 体系结构应具有如下特点：－采用固定长度的指令格式，指令归整、简单、基本寻址方式有2～3 种。－使用单周期指令，便于流水线操作执行。－大量使用寄存器，数据处理指令只对寄存器进行操作，只有加载/ 存储指令可以访问存储器，

以提高指令的执行效率。除此以外，ARM 体系结构还采用了一些特别的技术，在保证高性能的前提下尽量缩小芯片的面

积，并降低功耗：－所有的指令都可根据前面的执行结果决定是否被执行，从而提高指令的执行效率。－可用加载/存储指令批量传输数据，以提高数据的传输效率。－可在一条数据处理指令中同时完成逻辑处理和移位处理。－在循环处理中使用地址的自动增减来提高运行效率。当然，和CISC 架构相比较，尽管RISC 架构有上述的优点，但决不能认为RISC 架构就可以取

代CISC 架构，事实上，RISC 和CISC 各有优势，而且界限并不那么明显。现代的CPU 往往采用CISC 的外围，内部加入了RISC 的特性，如超长指令集CPU 就是融合了RISC 和CISC 的优势，成为未来的CPU 发展方向之一。

### 1.4.3 ARM 微处理器的指令结构

ARM 微处理器的在较新的体系结构中支持两种指令集：ARM 指令集和Thumb 指令集。其中，ARM 指令为32 位的长度，Thumb 指令为16 位长度。Thumb 指令集为ARM 指令集的功能子集，但与等价的ARM 代码相比较，可节省30%～40%以上的存储空间，同时具备32 位代码的所有优点。

关于ARM 处理器的指令结构，在后面的相关章节将会详细描述。

## 1.5 ARM 微处理器的应用选型

鉴于ARM 微处理器的众多优点，随着国内外嵌入式应用领域的逐步发展，ARM 微处理器必然会获得广泛的重视和应用。但是，由于ARM 微处理器有多达十几种的内核结构，几十个芯片生产厂家，以及千变万化的内部功能配置组合，给开发人员在选择方案时带来一定的困难，所以，对ARM 芯片做一些对比研究是十分必要的。

以下从应用的角度出发，对在选择ARM 微处理器时所应考虑的主要问题做一些简要的探讨。

**ARM 微处理器内核的选择**

从前面所介绍的内容可知，ARM 微处理器包含一系列的内核结构，以适应不同的应用领域，用户如果希望使用WinCE 或标准Linux 等操作系统以减少软件开发时间，就需要选择ARM720T 以上带有MMU（Memory Management Unit ）功能的ARM 芯片，ARM720T 、ARM920T 、ARM922T 、ARM946T 、Strong-ARM 都带有MMU 功能。而ARM7TDMI 则没有MMU，不支持Windows CE 和标准Linux，但目前有uCLinux 等不需要MMU 支持的操作系统可运行于ARM7TDMI 硬件平台之上。事实上，uCLinux 已经成功移植到多种不带MMU 的微处理器平台上，并在稳定性和其他方面都有上佳表现。

本书所讨论的S3C4510B 即为一款不带MMU 的ARM 微处理器，可在其上运行uCLinux 操作系统。

**系统的工作频率**

系统的工作频率在很大程度上决定了ARM 微处理器的处理能力。ARM7 系列微处理器的典型处理速度为0.9MIPS/MHz，常见的ARM7 芯片系统主时钟为20MHz-133MHz，ARM9 系列微处理器的典型处理速度为1.1MIPS/MHz，常见的ARM9 的系统主时钟频率为100MHz-233MHz，ARM10 最高可以达到700MHz 。不同芯片对时钟的处理不同，有的芯片只需要一个主时钟频率，有的芯片内部时钟控制器可以分别为ARM 核和USB、UART 、DSP、音频等功能部件提供不同频率的时钟。**芯片内存储器的容量**



PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

# 第2 章 ARM 微处理器的编程模型

本章简介ARM微处理器编程模型的一些基本概念，包括工作状态切换、数据的存储格式、处理器异常等，通过对本章的阅读，希望读者能了解ARM微处理器的基本工作原理和一些与程序设计相关的基本技术细节，为以后的程序设计打下基础。

本章的主要内容：

－ARM 微处理器的工作状态
－ARM 体系结构的存储器格式
－ARM 微处理器的工作模式
－ARM 体系结构的寄存器组织
－ARM微处理器的异常状态 在开始本章之前，首先对字（Word）、半字（Half-Word）、字节（Byte）的概念作一个说明：字（Word）：在ARM体系结构中，字的长度为32位，而在8位/16位处理器体系结构中，字的长

度一般为16位，请读者在阅读时注意区分。 半字（Half-Word）：在ARM体系结构中，半字的长度为16位，与8位/16位处理器体系结构中字的长度一致。 字节（Byte）：在ARM体系结构和8位/16位处理器体系结构中，字节的长度均为8位。

## 2.1 ARM 微处理器的工作状态

从编程的角度看，ARM微处理器的工作状态一般有两种，并可在两种状态之间切换：
－第一种为ARM状态，此时处理器执行32位的字对齐的ARM指令；
－第二种为Thumb状态，此时处理器执行16位的、半字对齐的Thumb指令。

当ARM微处理器执行32位的ARM指令集时，工作在ARM状态；当ARM微处理器执行16位的Thumb指令集时，工作在Thumb状态。在程序的执行过程中，微处理器可以随时在两种工作状态之间切换，并且，处理器工作状态的转变并不影响处理器的工作模式和相应寄存器中的内容。 **状态切换方法：**

ARM指令集和Thumb指令集均有切换处理器状态的指令，并可在两种工作状态之间切换，但ARM微处理器在开始执行代码时，应该处于ARM状态。

进入Thumb状态：当操作数寄存器的状态位（位0）为1时，可以采用执行BX指令的方法，使微处理器从ARM状态切换到Thumb状态。此外，当处理器处于Thumb状态时发生异常（如IRQ、FIQ、Undef、Abort、SWI等），则异常处理返回时，自动切换到Thumb状态。

进入ARM状态：当操作数寄存器的状态位为0时，执行BX指令时可以使微处理器从Thumb状态切换到ARM状态。此外，在处理器进行异常处理时，把PC指针放入异常模式链接寄存器中，并从异常向量地址开始执行程序，也可以使处理器切换到ARM状态。

## 2.2 ARM 体系结构的存储器格式

ARM体系结构将存储器看作是从零地址开始的字节的线性组合。从零字节到三字节放置第一个存储的字数据，从第四个字节到第七个字节放置第二个存储的字数据，依次排列。作为32位的微处理器，ARM体系结构所支持的最大寻址空间为4GB（$2^{32}$字节）。

ARM体系结构可以用两种方法存储字数据，称之为大端格式和小端格式，具体说明如下：



PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

**大端格式：**

在这种格式中，字数据的高字节存储在低地址中，而字数据的低字节则存放在高地址中，如图2.1所示：

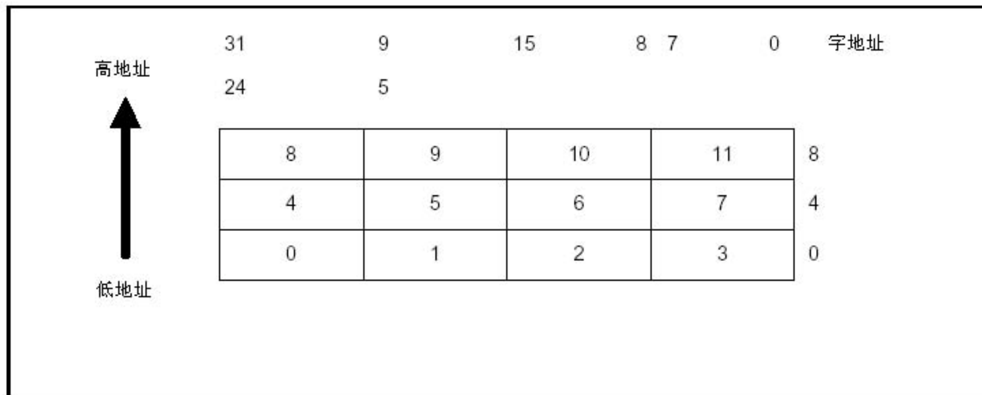


图2.1　以大端格式存储字数据

**小端格式：**

与大端存储格式相反，在小端存储格式中，低地址中存放的是字数据的低字节，高地址存放的是字数据的高字节。如图2.2所示：

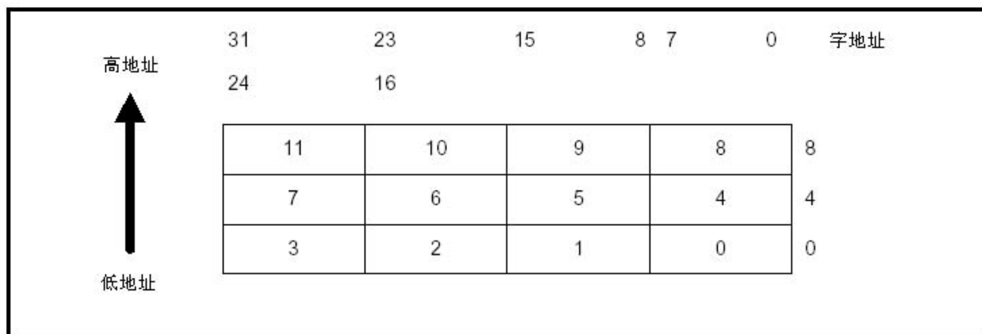


图2. 2 以小端格式存储字数据

## 2.3 指令长度及数据类型

ARM微处理器的指令长度可以是32位（在ARM状态下），也可以为16位（在Thumb状态下）。

ARM微处理器中支持字节（8位）、半字（16位）、字（32位）三种数据类型，其中，字需要4字节对齐（地址的低两位为0）、半字需要2字节对齐（地址的最低位为0）。

## 2.4 处理器模式

ARM微处理器支持7种运行模式，分别为：

— 用户模式（usr）：　ARM处理器正常的程序执行状态
— 快速中断模式（fiq）：　用于高速数据传输或通道处理
— 外部中断模式（irq）：　用于通用的中断处理
— 管理模式（svc）：　操作系统使用的保护模式
— 数据访问终止模式(abt)：　当数据或指令预取终止时进入该模式，可用于虚拟存储及存储保护。
— 系统模式（sys）：　运行具有特权的操作系统任务。
— 未定义指令中止模式（und）：当未定义的指令执行时进入该模式，可用于支持硬件协处



PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

理器的软件仿真。ARM微处理器的运行模式可以通过软件改变，也可以通过外部中断或异常处理改变。大多数的应用程序运行在用户模式下，当处理器运行在用户模式下时，某些被保护的系统资源是不能被访问的。

除用户模式以外，其余的所有6种模式称之为非用户模式，或特权模式（Privileged Modes）；其中除去用户模式和系统模式以外的5种又称为异常模式（Exception Modes），常用于处理中断或异常，以及需要访问受保护的系统资源等情况。

## 2.5 寄存器组织

ARM微处理器共有37个32位寄存器，其中31个为通用寄存器，6个为状态寄存器。但是这些寄存器不能被同时访问，具体哪些寄存器是可编程访问的，取决微处理器的工作状态及具体的运行模式。但在任何时候，通用寄存器R14～R0、程序计数器PC、一个或两个状态寄存器都是可访问的。

### 2.5.1 ARM 状态下的寄存器组织

**通用寄存器：**

通用寄存器包括R0～R15，可以分为三类： — 未分组寄存器R0～R7； — 分组寄存器R8～R14 — 程序计数器PC(R15)

**未分组寄存器R0～R7：**

在所有的运行模式下，未分组寄存器都指向同一个物理寄存器，他们未被系统用作特殊的用途，因此，在中断或异常处理进行运行模式转换时，由于不同的处理器运行模式均使用相同的物理寄存器，可能会造成寄存器中数据的破坏，这一点在进行程序设计时应引起注意。

**分组寄存器R8～R14**

对于分组寄存器，他们每一次所访问的物理寄存器与处理器当前的运行模式有关。对于R8～R12来说，每个寄存器对应两个不同的物理寄存器，当使用fiq模式时，访问寄存器R8_fiq～R12_fiq；当使用除fiq模式以外的其他模式时，访问寄存器R8_usr～R12_usr。对于R13、R14来说，每个寄存器对应6个不同的物理寄存器，其中的一个是用户模式与系统模式共用，另外5个物理寄存器对应于其他5种不同的运行模式。采用以下的记号来区分不同的物理寄存器：

```
R13_<mode>
R14_<mode>
```

其中，mode为以下几种模式之一：usr、fiq、irq、svc、abt、und。寄存器R13在ARM指令中常用作堆栈指针，但这只是一种习惯用法，用户也可使用其他的寄存器作为堆栈指针。而在Thumb指令集中，某些指令强制性的要求使用R13作为堆栈指针。

由于处理器的每种运行模式均有自己独立的物理寄存器R13，在用户应用程序的初始化部分，一般都要初始化每种模式下的R13，使其指向该运行模式的栈空间，这样，当程序的运行进入异常模式时，可以将需要保护的寄存器放入R13所指向的堆栈，而当程序从异常模式返回时，则从对应的堆栈中恢复，采用这种方式可以保证异常发生后程序的正常执行。

R14也称作子程序连接寄存器（Subroutine Link Register）或连接寄存器LR。当执行BL子程序调用指令时，R14中得到R15（程序计数器PC）的备份。其他情况下，R14用作通用寄存器。与之类似，当发生中断或异常时，对应的分组寄存器R14_svc、R14_irq、R14_fiq、R14_abt和R14_und用来保存R15的返回值。

寄存器R14常用在如下的情况：

在每一种运行模式下，都可用R14保存子程序的返回地址，当用BL或BLX指令调用子程序时，将PC的当前值拷贝给R14，执行完子程序后，又将R14的值拷贝回PC，即可完成子程序的调用返回。以上的描述可用



PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

指令完成：

1、执行以下任意一条指令： MOV PC，LR BX LR

2、在子程序入口处使用以下指令将**R14**存入堆栈： STMFD SP！,{<Regs>,LR} 对应的，使用以下指令可以完成子程序返回： LDMFD SP！,{<Regs>,PC}

**R14**也可作为通用寄存器。

**程序计数器PC(R15)**

寄存器**R15**用作程序计数器（**PC**）。在**ARM**状态下，位**[1:0]**为**0**，位**[31:2]**用于保存**PC**；在**Thumb**状态下，位**[0]**为**0**，位**[31:1]**用于保存**PC**；虽然可以用作通用寄存器，但是有一些指令在使用**R15**时有一些特殊限制，若不注意，执行的结果将是不可预料的。在**ARM**状态下，**PC**的**0**和**1**位是**0**，在**Thumb**状态下，**PC**的**0**位是**0**。

**R15**虽然也可用作通用寄存器，但一般不这么使用，因为对**R15**的使用有一些特殊的限制，当违反了这些限制时，程序的执行结果是未知的。 由于ARM 体系结构采用了多级流水线技术，对于ARM 指令集而言，PC 总是指向当前指令的下两条指令的地址，即PC 的值为当前指令的地址值加8 个字节。

图**2.3** **ARM** 状态下的寄存器组织

在**ARM**状态下，任一时刻可以访问以上所讨论的**16**个通用寄存器和一到两个状态寄存器。在非用户模式（特权模式）下，则可访问到特定模式分组寄存器，图**2.3**说明在每一种运行模式下，哪一些寄存器是可以访问的。 **寄存器R16**：

寄存器R16用作CPSR(Current Program Status Register ，当前程序状态寄存器)，CPSR可在任何运行模式下被访问，它包括条件标志位、中断禁止位、当前处理器模式标志位，以及其他一些相关的控制和状态位。

每一种运行模式下又都有一个专用的物理状态寄存器，称为**SPSR**（**Saved Program Status Register**，备份的程序状态寄存器），当异常发生时，**SPSR**用于保存**CPSR**的当前值，从异常退出时则可由**SPSR**来恢复**CPSR**。

由于用户模式和系统模式不属于异常模式，他们没有**SPSR**，当在这两种模式下访问**SPSR**，结果是未知的。



PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

## 2.5.2 Thumb 状态下的寄存器组织

**Thumb**状态下的寄存器集是**ARM**状态下寄存器集的一个子集，程序可以直接访问**8**个通用寄存器（**R7**～**R0**）、程序计数器（**PC**）、堆栈指针（**SP**）、连接寄存器（**LR**）和**CPSR**。同时，在每一种特权模式下都有一组**SP**、**LR**和**SPSR**。图**2.4**表明**Thumb**状态下的寄存器组织。

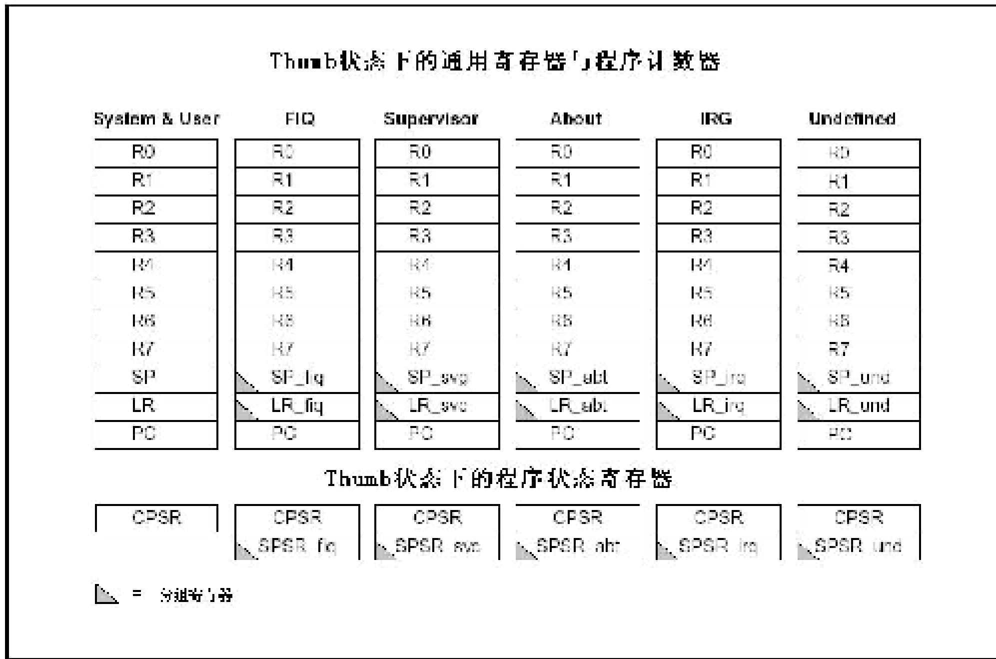


图**2.4** **Thumb** 状态下的寄存器组织

**Thumb**状态下的寄存器组织与**ARM**状态下的寄存器组织的关系：

— **Thumb**状态下和**ARM**状态下的**R0**～**R7**是相同的。 — **Thumb**状态下和**ARM**状态下的**CPSR**和所有的**SPSR**是相同的。 — **Thumb**状态下的**SP**对应于**ARM**状态下的**R13**。 — **Thumb**状态下的**LR**对应于**ARM**状态下的**R14**。 — **Thumb**状态下的程序计数器对应于**ARM**状态下**R15** 以上的对应关系如图**2.5**所示：

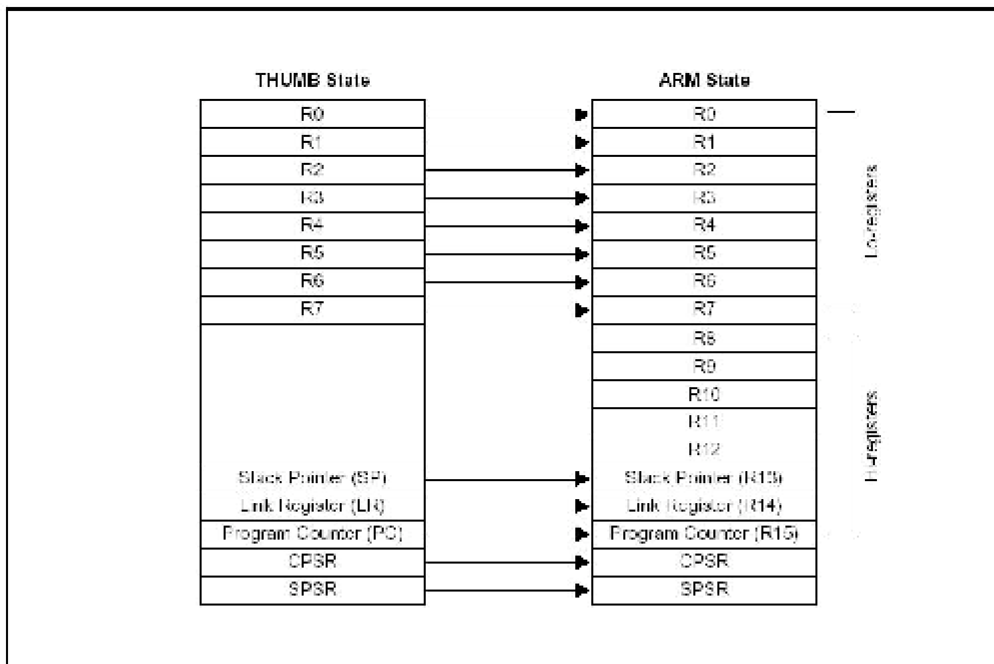


图**2.5** **Thumb** 状态下的寄存器组织



PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

访问THUMB状态下的高位寄存器（Hi-registers）：

在Thumb状态下，高位寄存器R8～R15并不是标准寄存器集的一部分，但可使用汇编语言程序受限制的访问这些寄存器，将其用作快速的暂存器。使用带特殊变量的MOV指令，数据可以在低位寄存器和高位寄存器之间进行传送；高位寄存器的值可以使用CMP和ADD指令进行比较或加上低位寄存器中的值。

## 2.5.3 程序状态寄存器

ARM体系结构包含一个当前程序状态寄存器（CPSR）和五个备份的程序状态寄存器（SPSRs）。备份的程序状态寄存器用来进行异常处理，其功能包括：— 保存ALU中的当前操作信息 — 控制允许和禁止中断 — 设置处理器的运行模式 程序状态寄存器的每一位的安排如图2.6所示：

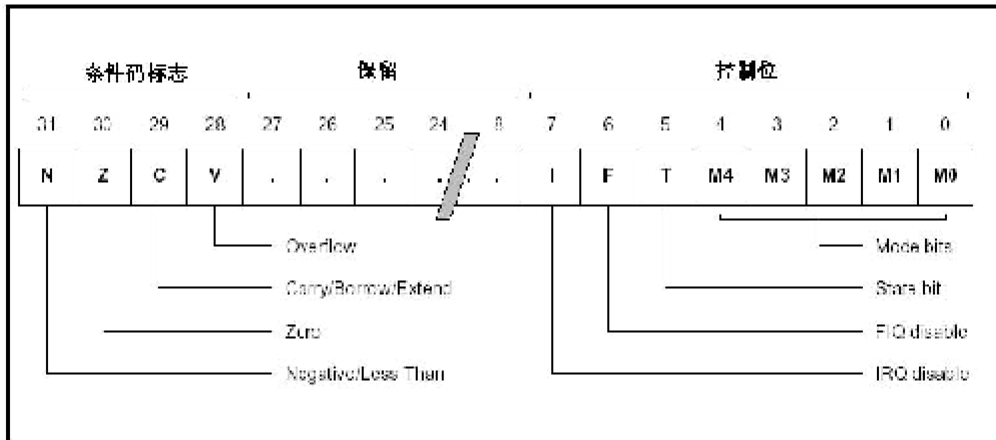


图2.6 程序状态寄存器格式

条件码标志（Condition Code Flags）

N、Z、C、V均为条件码标志位。它们的内容可被算术或逻辑运算的结果所改变，并且可以决定某条指令是否被执行。 在ARM状态下，绝大多数的指令都是有条件执行的。 在Thumb状态下，仅有分支指令是有条件执行的。 条件码标志各位的具体含义如表2-1所示：

表2-1 条件码标志的具体含义

| 标志位 | 含义 |
|---|---|
| N | 当用两个补码表示的带符号数进行运算时，N=1 表示运算的结果为负数；N=0 表示运算的结果为正数或零； |
| Z | Z=1 表示运算的结果为零；Z=0表示运算的结果为非零； |
| C | 可以有4 种方法设置C 的值： — 加法运算（包括比较指令CMN）： 当运算结果产生了进位时（无符号数溢出），C=1， 否则C=0。— 减法运算（包括比较指令CMP）：当运算时产生了借位（无符号数溢出），C=0，否则C=1。 — 对于包含移位操作的非加/减运算指令，C为移出值的最后一位。— 对于其他的非加/减运算指令，C的值通常不改变。 |
| V | 可以有2 种方法设置V 的值： — 对于加/减法运算指令，当操作数和运算结果为二进制的补码表示的带符号数时，V=1表示符号位溢出。 — 对于其他的非加/减运算指令，V的值通常不改变。 |



PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

| Q | 在ARM v5 及以上版本的E 系列处理器中，用Q 标志位指示增强的DSP 运算指令是否发生了溢出。在其他版本的处理器中，Q标志位无定义。 |
|---|---|

**控制位** PSR的低8位（包括I、F、T和M[4：0]）称为控制位，当发生异常时这些位可以被改变。如果处理器运行特权模式，这些位也可以由程序修改。— 中断禁止位I、F： I=1 禁止IRQ中断；F=1 禁止FIQ中断。— T标志位：该位反映处理器的运行状态。对于ARM体系结构v5及以上的版本的T系列处理器，当该位为1时，程序运行于Thumb状态，否则运行于ARM状态。对于ARM体系结构v5及以上的版本的非T系列处理器，当该位为1时，执行下一条指令以引起为定义的指令异常；当该位为0时，表示运行于ARM状态。 — 运行模式位M[4：0]：M0、M1、M2、M3、M4是模式位。这些位决定了处理器的运行模式。

具体含义如表2-2所示：

表2-2 运行模式位M[4：0]的具体含义

| M[4：0] | 处理器模式 | 可访问的寄存器 |
|---|---|---|
| 0b10000 | 用户模式 | PC，CPSR,R0-R14 |
| 0b10001 | FIQ 模式 | PC，CPSR, SPSR_fiq，R14_fiq-R8_fiq, R7～R0 |
| 0b10010 | IRQ 模式 | PC，CPSR, SPSR_irq，R14_irq,R13_irq,R12～R0 |
| 0b10011 | 管理模式 | PC，CPSR, SPSR_svc，R14_svc,R13_svc,,R12～R0, |
| 0b10111 | 中止模式 | PC，CPSR, SPSR_abt，R14_abt,R13_abt, R12～R0, |
| 0b11011 | 未定义模式 | PC，CPSR, SPSR_und，R14_und,R13_und, R12～R0, |
| 0b11111 | 系统模式 | PC，CPSR（ARM v4 及以上版本）, R14～R0 |

由表2-2 可知，并不是所有的运行模式位的组合都是有效地，其他的组合结果会导致处理器进入一个不可恢复的状态。**保留位**

PSR中的其余位为保留位，当改变PSR中的条件码标志位或者控制位时，保留位不要被改变，在程序中也不要使用保留位来存储数据。保留位将用于ARM版本的扩展。

## 2.6 异常（Exceptions）

当正常的程序执行流程发生暂时的停止时，称之为异常，例如处理一个外部的中断请求。在处理异常之前，当前处理器的状态必须保留，这样当异常处理完成之后，当前程序可以继续执行。处理器允许多个异常同时发生，它们将会按固定的优先级进行处理。

ARM体系结构中的异常，与8位/16位体系结构的中断有很大的相似之处，但异常与中断的概念并不完全等同。

### 2.6.1 ARM 体系结构所支持的异常类型

ARM体系结构所支持的异常及具体含义如表2-3所示。

表2-3 ARM 体系结构所支持的异常



PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

| 异常类型 | 具体含义 |
|---|---|
| 复位 | 当处理器的复位电平有效时，产生复位异常，程序跳转到复位异常处理程序处执行。 |
| 未定义指令 | 当ARM 处理器或协处理器遇到不能处理的指令时，产生未定义指令异常。可使用该异常机制进行软件仿真。 |
| 软件中断 | 该异常由执行SWI 指令产生，可用于用户模式下的程序调用特权操作指令。可使用该异常机制实现系统功能调用。 |
| 指令预取中止 | 若处理器预取指令的地址不存在，或该地址不允许当前指令访问，存储器会向处理器发出中止信号，但当预取的指令被执行时，才会产生指令预取中止异常。 |
| 数据中止 | 若处理器数据访问指令的地址不存在，或该地址不允许当前指令访问时，产生数据中止异常。 |
| IRQ（外部中断请求） | 当处理器的外部中断请求引脚有效，且CPSR 中的I 位为0 时，产生IRQ 异常。系统的外设可通过该异常请求中断服务。 |
| FIQ（快速中断请求） | 当处理器的快速中断请求引脚有效，且CPSR 中的F 位为0 时，产生FIQ 异常。 |

## 2.6.2 对异常的响应

当一个异常出现以后，ARM微处理器会执行以下几步操作：

1、将下一条指令的地址存入相应连接寄存器LR，以便程序在处理异常返回时能从正确的位置重新开始执行。若异常是从ARM状态进入，LR寄存器中保存的是下一条指令的地址（当前PC＋4或PC＋8，与异常的类型有关）；若异常是从Thumb状态进入，则在LR寄存器中保存当前PC的偏移量，这样，异常处理程序就不需要确定异常是从何种状态进入的。例如：在软件中断异常SWI，指令 MOV PC，R14_svc总是返回到下一条指令，不管SWI是在ARM状态执行，还是在Thumb状态执行。

2、将CPSR复制到相应的SPSR中。 3、根据异常类型，强制设置CPSR的运行模式位。 4、强制PC从相关的异常向量地址取下一条指令执行，从而跳转到相应的异常处理程序处。 还可以设置中断禁止位，以禁止中断发生。 如果异常发生时，处理器处于Thumb状态，则当异常向量地址加载入PC时，处理器自动切换到

ARM状态。 ARM微处理器对异常的响应过程用伪码可以描述为：

```
R14_<Exception_Mode> = Return Link
SPSR_<Exception_Mode> = CPSR
CPSR[4:0] = Exception Mode Number

CPSR[5] = 0 ；当运行于ARM工作状态时If <Exception_Mode> == Reset or FIQ then
                              ；当响应FIQ异常时，禁止新的FIQ异常CPSR[6] = 1 CPSR[7] = 1
PC = Exception Vector Address
```

## 2.6.3 从异常返回

异常处理完毕之后，ARM微处理器会执行以下几步操作从异常返回： 1、将连接寄存器LR的值减去相应的偏移量后送到PC中。 2、将SPSR复制回CPSR中。 3、若在进入异常处理时设置了中断禁止位，要在此清除。可以认为应用程序总是从复位异常处理程序开始执行的，因此复位异常处理程序不需要返回。

## 2.6.4 各类异常的具体描述

**FIQ（Fast Interrupt Request）**

FIQ异常是为了支持数据传输或者通道处理而设计的。在ARM状态下，系统有足够的私有寄存器，从而可以避免对寄存器保存的需求，并减小了系统上下文切换的开销。 若将CPSR的F位置为1，则会禁止FIQ中断，若将CPSR的F位清零，处理器会在指令执行时检查FIQ



PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

的输入。注意只有在特权模式下才能改变**F**位的状态。 可由外部通过对处理器上的**nFIQ**
引脚输入低电平产生**FIQ**。不管是在**ARM**状态还是在**Thumb**状态下进入**FIQ**模式，**FIQ**处理程
序均会执行以下指令从**FIQ**模式返回：

```
SUBS PC,R14_fiq ,#4
```

该指令将寄存器R14_fiq 的值减去4 后，复制到程序计数器PC 中，从而实现从异常处理程序中的返回，同时将SPSR_mode 寄存器的内容复制到当前程序状态寄存器CPSR 中。

**IRQ（Interrupt Request）**

**IRQ**异常属于正常的中断请求，可通过对处理器的**nIRQ**引脚输入低电平产生，**IRQ**
的优先级低于**FIQ**，当程序执行进入**FIQ**异常时，**IRQ**可能被屏蔽。 若将**CPSR**的**I**位置为
**1**，则会禁止**IRQ**中断，若将**CPSR**的**I**位清零，处理器会在指令执行完之前检查**IRQ**的输
入。注意只有在特权模式下才能改变**I**位的状态。 不管是在**ARM**状态还是在**Thumb**状态
下进入**IRQ**模式，**IRQ**处理程序均会执行以下指令从**IRQ**模式返回：

```
SUBS PC , R14_irq , #4
```

该指令将寄存器R14_irq 的值减去4 后，复制到程序计数器PC 中，从而实现从异常处理程序中的返回，同时将SPSR_mode 寄存器的内容复制到当前程序状态寄存器CPSR 中。

**ABORT（中止）**

产生中止异常意味着对存储器的访问失败。**ARM**微处理器在存储器访问周期内检查是否发生中
止异常。 中止异常包括两种类型： — 指令预取中止：发生在指令预取时。 — 数据中止：发生在数据访
问时。 当指令预取访问存储器失败时，存储器系统向**ARM**处理器发出存储器中止（**Abort**）信号，预取
的指令被记为无效，但只有当处理器试图执行无效指令时，指令预取中止异常才会发生，
如果指令未被执行，例如在指令流水线中发生了跳转，则预取指令中止不会发生。 若数据
中止发生，系统的响应与指令的类型有关。

当确定了中止的原因后，**Abort**处理程序均会执行以下指令从中止模式返回，无论是在**ARM**
状态还是**Thumb**状态： SUBS PC，R14_abt，#4 ；指令预取中止SUBS PC，R14_abt，#8 ；数据中止

以上指令恢复**PC**（从**R14_abt**）和**CPSR**（从**SPSR_abt**）的值，并重新执行中止的指令。

**Software Interruupt(软件中断)**

软件中断指令（**SWI**）用于进入管理模式，常用于请求执行特定的管理功能。软件中断处理程序执行以下指令从**SWI**模式返回，无论是在**ARM**状态还是**Thumb**状态：

```
MOV PC , R14_svc
```

以上指令恢复**PC**（从**R14_svc**）和**CPSR**（从**SPSR_svc**）的值，并返回到**SWI**的下一条指令。

**Undefined Instruction(未定义指令)**

当**ARM**处理器遇到不能处理的指令时，会产生未定义指令异常。
采用这种机制，可以通过软件仿真扩展**ARM**或**Thumb**指令集。 在仿真
未定义指令后，处理器执行以下程序返回，无论是在**ARM**状态还是
**Thumb**状态：

```
MOVS PC，R14_und
```

以上指令恢复**PC**（从**R14_und**）和**CPSR**（从**SPSR_und**）的值，并返回到未定义指令后的下一条指令。

## 2.6.5 异常进入/退出小节

表2-4 总结了进入异常处理时保存在相应R14 中的PC 值，及在退出异常处理时推荐使用的指令。

表2-4 异常进入/退出

| | 返回指令 | 以前的状态 | | 注意 |
|---|---|---|---|---|
| | | ARM R14_x | Thumb R14_x | |
| BL | MOV PC，R14 | PC+4 | PC+2 | 1 |



PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

| | | | | |
|---|---|---|---|---|
| SWI | MOVS PC，R14_svc | PC＋4 | PC＋2 | 1 |
| UDEF | MOVS PC，R14_und | PC＋4 | PC＋2 | 1 |
| FIQ | SUBS PC，R14_fiq，＃4 | PC＋4 | PC＋4 | 2 |
| IRQ | SUBS PC，R14_irq，＃4 | PC＋4 | PC＋4 | 2 |
| PABT | SUBS PC，R14_abt，＃4 | PC＋4 | PC＋4 | 1 |
| DABT | SUBS PC，R14_abt，＃8 | PC＋8 | PC＋8 | 3 |
| RESET | NA | — | — | 4 |

注意：1、在此PC 应是具有预取中止的BL/SWI/未定义指令所取的地址。2、在此PC 是从FIQ 或IRQ 取得不能执行的指令的地址。3、在此PC 是产生数据中止的加载或存储指令的地址。4、系统复位时，保存在R14_svc 中的值是不可预知的。

## 2.6.6 异常向量（Exception Vectors）

表**2-5**显示异常向量地址。

**表2-5 异常向量表**

| 地址 | 异常 | 进入模式 |
|---|---|---|
| 0x0000,0000 | 复位 | 管理模式 |
| 0x0000,0004 | 未定义指令 | 未定义模式 |
| 0x0000,0008 | 软件中断 | 管理模式 |
| 0x0000,000C<br>0x0000,0010<br>0x0000,0014<br>0x0000,0018 | 中止（预取指令）中止（数据）保留IRQ | 中止模式中止模式保留IRQ |
| 0x0000,001C | FIQ | FIQ |



## 2.6.7 异常优先级（Exception Priorities）

当多个异常同时发生时，系统根据固定的优先级决定异常的处理次序。异常优先级由高到低的排列次序如表**2-6**所示。

**表2-6 异常优先级**

| 优先级 | 异常 |
|---|---|
| 1（最高） | 复位 |
| 2 | 数据中止 |
| 3 | FIQ |
| 4 | IRQ |
| 5 | 预取指令中止 |
| 6（最低） | 未定义指令、SWI |

## 2.6.8 应用程序中的异常处理

当系统运行时，异常可能会随时发生，为保证在ARM 处理器发生异常时不至于处于未知状态，在应



PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

用程序的设计中，首先要进行异常处理，采用的方式是在异常向量表中的特定位置放置一条跳转指令，跳转到异常处理程序，当ARM 处理器发生异常时，程序计数器PC 会被强制设置为对应的异常向量，从而跳转到异常处理程序，当异常处理完成以后，返回到主程序继续执行。

## 2.7 本章小节

本章对ARM 微处理器的体系结构、寄存器的组织、处理器的工作状态、运行模式以及处理器异常等内容进行了描述，这些内容也是ARM 体系结构的基本内容，是系统软、硬件设计的基础。



PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

# 第3 章 ARM 微处理器的指令系统

本章介绍ARM指令集、Thumb指令集，以及各类指令对应的寻址方式，通过对本章的阅读，希望读者能了解ARM 微处理器所支持的指令集及具体的使用方法。

本章的主要内容有：

－ARM 指令集、Thumb指令集概述。
－ARM 指令集的分类与具体应用。
－Thumb 指令集简介及应用场合。

## 3.1 ARM 微处理器的指令集概述

### 3.1.1 ARM 微处理器的指令的分类与格式

ARM微处理器的指令集是加载/存储型的，也即指令集仅能处理寄存器中的数据，而且处理结果都要放回寄存器中，而对系统存储器的访问则需要通过专门的加载/存储指令来完成。

ARM微处理器的指令集可以分为跳转指令、数据处理指令、程序状态寄存器（PSR）处理指令、加载/存储指令、协处理器指令和异常产生指令六大类，具体的指令及功能如表3-1所示（表中指令为基本ARM指令，不包括派生的ARM指令）。

表3-1 ARM 指令及功能描述

| 助记符 | 指令功能描述 |
|---|---|
| ADC | 带进位加法指令 |
| ADD | 加法指令 |
| AND | 逻辑与指令 |
| B | 跳转指令 |
| BIC | 位清零指令 |
| BL | 带返回的跳转指令 |
| BLX | 带返回和状态切换的跳转指令 |
| BX | 带状态切换的跳转指令 |
| CDP | 协处理器数据操作指令 |
| CMN | 比较反值指令 |
| CMP | 比较指令 |
| EOR | 异或指令 |
| LDC | 存储器到协处理器的数据传输指令 |
| LDM | 加载多个寄存器指令 |
| LDR | 存储器到寄存器的数据传输指令 |
| MCR | 从ARM 寄存器到协处理器寄存器的数据传输指令 |
| MLA | 乘加运算指令 |
| MOV | 数据传送指令 |
| MRC | 从协处理器寄存器到ARM 寄存器的数据传输指令 |
| MRS | 传送CPSR 或SPSR 的内容到通用寄存器指令 |
| MSR | 传送通用寄存器到CPSR 或SPSR 的指令 |



PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

| | |
|---|---|
| MUL | 32 位乘法指令 |
| MLA | 32 位乘加指令 |
| MVN | 数据取反传送指令 |
| ORR | 逻辑或指令 |
| RSB | 逆向减法指令 |
| RSC | 带借位的逆向减法指令 |
| SBC | 带借位减法指令 |
| STC | 协处理器寄存器写入存储器指令 |
| STM | 批量内存字写入指令 |
| STR | 寄存器到存储器的数据传输指令 |
| SUB | 减法指令 |
| SWI | 软件中断指令 |
| SWP | 交换指令 |
| TEQ | 相等测试指令 |
| TST | 位测试指令 |

### 3.1.2 指令的条件域

当处理器工作在ARM状态时，几乎所有的指令均根据CPSR中条件码的状态和指令的条件域有条件的执行。当指令的执行条件满足时，指令被执行，否则指令被忽略。

每一条ARM指令包含4位的条件码，位于指令的最高4位[31:28] 。条件码共有16种，每种条件码可用两个字符表示，这两个字符可以添加在指令助记符的后面和指令同时使用。例如，跳转指令B 可以加上后缀EQ变为BEQ表示"相等则跳转"，即当CPSR中的Z标志置位时发生跳转。

在16种条件标志码中，只有15种可以使用，如表3-2所示，第16种（1111 ）为系统保留，暂时不能使用。

**表3-2 指令的条件码**

| 条件码 | 助记符后缀 | 标志 | 含义 |
|---|---|---|---|
| 0000 | EQ | Z 置位 | 相等 |
| 0001 | NE | Z 清零 | 不相等 |
| 0010 | CS | C 置位 | 无符号数大于或等于 |
| 0011 | CC | C 清零 | 无符号数小于 |
| 0100 | MI | N 置位 | 负数 |
| 0101 | PL | N 清零 | 正数或零 |
| 0110 | VS | V 置位 | 溢出 |
| 0111 | VC | V 清零 | 未溢出 |
| 1000 | HI | C 置位Z 清零 | 无符号数大于 |
| 1001 | LS | C 清零Z 置位 | 无符号数小于或等于 |
| 1010 | GE | N 等于V | 带符号数大于或等于 |
| 1011 | LT | N 不等于V | 带符号数小于 |
| 1100 | GT | Z 清零且（N 等于V） | 带符号数大于 |
| 1101 | LE | Z 置位或（N 不等于V） | 带符号数小于或等于 |
| 1110 | AL | 忽略 | 无条件执行 |

## 3.2 ARM 指令的寻址方式



PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

所谓寻址方式就是处理器根据指令中给出的地址信息来寻找物理地址的方式。目前ARM指令系统支持如下几种常见的寻址方式。

### 3.2.1 立即寻址

立即寻址也叫立即数寻址，这是一种特殊的寻址方式，操作数本身就在指令中给出，只要取出指令也就取到了操作数。这个操作数被称为立即数，对应的寻址方式也就叫做立即寻址。例如以下指令：

ADD R0，R0，#1 ；R0←**R0**＋**1** ADD R0，R0，#0x3f ；R0←**R0**＋**0x3f** 在以上两条指令中，第二个源操作数即为立即数，要求以"#"为前缀，对于以十六进制表示

的立即数，还要求在"#"后加上"0x"或"&"。

### 3.2.2 寄存器寻址

寄存器寻址就是利用寄存器中的数值作为操作数，这种寻址方式是各类微处理器经常采用的一种方式，也是一种执行效率较高的寻址方式。以下指令：

ADD R0，R1，R2 ；R0←**R1**＋**R2**

该指令的执行效果是将寄存器R1和R2的内容相加，其结果存放在寄存器R0中。

### 3.2.2 寄存器间接寻址

寄存器间接寻址就是以寄存器中的值作为操作数的地址，而操作数本身存放在存储器中。例如

以下指令：

```
    ADD R0，R1，[R2]   ；R0←R1＋[R2]
LDR R0，[R1]           ；R0←[R1]
STR R0，[R1]           ；[R1]←R0
```

在第一条指令中，以寄存器R2 的值作为操作数的地址，在存储器中取得一个操作数后与R1 相加，结果存入寄存器R0 中。第二条指令将以R1 的值为地址的存储器中的数据传送到R0 中。第三条指令将R0 的值传送到以R1 的值为地址的存储器中。

### 3.2.3 基址变址寻址

基址变址寻址就是将寄存器（该寄存器一般称作基址寄存器）的内容与指令中给出的地址偏移量相加，从而得到一个操作数的有效地址。变址寻址方式常用于访问某基地址附近的地址单元。采用变址寻址方式的指令常见有以下几种形式，如下所示：

LDR R0，[R1，#4]；R0←**[R1＋4]** LDR R0，[R1，#4]!；R0←**[R1＋4]**、**R1**←**R1**＋**4** LDR R0，[R1]，#4；R0←**[R1]**、**R1**←**R1**＋**4** LDR R0，[R1，R2]；R0←**[R1＋R2]** 在第一条指令中，将寄存器R1 的内容加上4 形成操作数的有效地址，从而取得操作数存入寄存器R0 中。在第二条指令中，将寄存器R1 的内容加上4 形成操作数的有效地址，从而取得操作数存入寄存器R0 中，然后，R1 的内容自增4 个字节。在第三条指令中，以寄存器R1 的内容作为操作数的有效地址，从而取得操作数存入寄存器R0 中，然后，R1 的内容自增4 个字节。在第四条指令中，将寄存器R1 的内容加上寄存器R2 的内容形成操作数的有效地址，从而取得操作数存入寄存器R0 中。

### 3.2.4 多寄存器寻址



PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

采用多寄存器寻址方式，一条指令可以完成多个寄存器值的传送。这种寻址方式可以用一条指令完成传送最多16 个通用寄存器的值。以下指令：LDMIAR0，{R1，R2，R3，R4}；R1←[R0]；R2←[R0+4]；R3←[R0+8]；R4←[R0+12] 该指令的后缀IA 表示在每次执行完加载/存储操作后，R0 按字长度增加，因此，指令可将连续存储单元的值传送到R1～R4。

### 3.2.5 相对寻址

与基址变址寻址方式相类似，相对寻址以程序计数器PC 的当前值为基地址，指令中的地址标号作为偏移量，将两者相加之后得到操作数的有效地址。以下程序段完成子程序的调用和返回，跳转指令BL 采用了相对寻址方式：

```
    BL NEXT ；跳转到子程序NEXT处执行
    ……
NEXT
    ……
    MOV PC，LR ；从子程序返回
```

### 3.2.6 堆栈寻址

堆栈是一种数据结构，按先进后出（First In Last Out，FILO）的方式工作，使用一个称作堆栈指针的专用寄存器指示当前的操作位置，堆栈指针总是指向栈顶。

当堆栈指针指向最后压入堆栈的数据时，称为满堆栈（Full Stack），而当堆栈指针指向下一个将要放入数据的空位置时，称为空堆栈（Empty Stack ）。

同时，根据堆栈的生成方式，又可以分为递增堆栈（Ascending Stack ）和递减堆栈（Decending Stack），当堆栈由低地址向高地址生成时，称为递增堆栈，当堆栈由高地址向低地址生成时，称为递减堆栈。这样就有四种类型的堆栈工作方式，ARM 微处理器支持这四种类型的堆栈工作方式，即：

－满递增堆栈：堆栈指针指向最后压入的数据，且由低地址向高地址生成。

－满递减堆栈：堆栈指针指向最后压入的数据，且由高地址向低地址生成。

－空递增堆栈：堆栈指针指向下一个将要放入数据的空位置，且由低地址向高地址生成。

－空递减堆栈：堆栈指针指向下一个将要放入数据的空位置，且由高地址向低地址生成。

## ARM 指令集

### 跳转指令

本节对ARM 指令集的六大类指令进行详细的描述。

跳转指令用于实现程序流程的跳转，在ARM 程序中有两种方法可以实现程序流程的跳转：

— 使用专门的跳转指令。

— 直接向程序计数器PC 写入跳转地址值。通过向程序计数器PC 写入跳转地址值，可以实现在4GB 的地址空间中的任意跳转，在跳转之

前结合使用MOV LR，PC 等类似指令，可以保存将来的返回地址值，从而实现在4GB 连续的线性地址空间的子程序调用。 ARM 指令集中的跳转指令可以完成从当前指令向前或向后的32MB 的地址空间的跳转，包括以下4



PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

条指令：

— B 跳转指令
— **BL** 带返回的跳转指令
— **BLX** 带返回和状态切换的跳转指令
— **BX** 带状态切换的跳转指令

**1、 B 指令**

**B** 指令的格式为：

**B{**条件**}** 目标地址

**B** 指令是最简单的跳转指令。一旦遇到一个 **B** 指令，**ARM** 处理器将立即跳转到给定的目标地址，从那里继续执行。注意存储在跳转指令中的实际值是相对当前**PC** 值的一个偏移量，而不是一个绝对地址，它的值由汇编器来计算（参考寻址方式中的相对寻址）。它是 **24** 位有符号数，左移两位后有符号扩展为 **32** 位，表示的有效偏移为 **26** 位**(**前后**32MB** 的地址空间**)**。以下指令：

B Label ；程序无条件跳转到标号Label处执行CMP R1，＃0 ；当CPSR寄存器中的Z条件码置位时，程序跳转到标号Label处执行BEQ Label

**2、 BL 指令**

**BL** 指令的格式为：

**BL{**条件**}** 目标地址

**BL** 是另一个跳转指令，但跳转之前，会在寄存器**R14** 中保存**PC** 的当前内容，因此，可以通过将**R14** 的内容重新加载到**PC** 中，来返回到跳转指令之后的那个指令处执行。该指令是实现子程序调用的一个基本但常用的手段。以下指令：

BL Label ；当程序无条件跳转到标号Label处执行时，同时将当前的PC值保存到R14中**3、 BLX 指令**

**BLX** 指令的格式为：

**BLX** 目标地址

**BLX** 指令从**ARM** 指令集跳转到指令中所指定的目标地址，并将处理器的工作状态有**ARM** 状态切换到**Thumb** 状态，该指令同时将**PC** 的当前内容保存到寄存器**R14** 中。因此，当子程序使用**Thumb** 指令集，而调用者使用**ARM** 指令集时，可以通过**BLX** 指令实现子程序的调用和处理器工作状态的切换。同时，子程序的返回可以通过将寄存器**R14** 值复制到**PC** 中来完成。 **4、 BX 指令**

**BX** 指令的格式为：

**BX{**条件**}** 目标地址

**BX** 指令跳转到指令中所指定的目标地址，目标地址处的指令既可以是**ARM**指令，也可以是**Thumb**指令。

## 3.3.2 数据处理指令

数据处理指令可分为数据传送指令、算术逻辑运算指令和比较指令等。

数据传送指令用于在寄存器和存储器之间进行数据的双向传输。

算术逻辑运算指令完成常用的算术与逻辑的运算，该类指令不但将运算结果保存在目的寄存器中，同时更新**CPSR** 中的相应条件标志位。

比较指令不保存运算结果，只更新**CPSR** 中相应的条件标志位。 数据处理指令包括：

— **MOV** 数据传送指令
— **MVN** 数据取反传送指令
— **CMP** 比较指令
— **CMN** 反值比较指令
— **TST** 位测试指令
— **TEQ** 相等测试指令
— **ADD** 加法指令
— **ADC** 带进位加法指令



**PDF** 文件使用 **"pdfFactory Pro"** 试用版本创建 **www.fineprint.cn**

— **SUB** 减法指令
— **SBC** 带借位减法指令
— **RSB** 逆向减法指令
— **RSC** 带借位的逆向减法指令
— **AND** 逻辑与指令
— **ORR** 逻辑或指令
— **EOR** 逻辑异或指令
— **BIC** 位清除指令

**1、 MOV 指令**

**MOV** 指令的格式为：

**MOV{**条件**}{S}** 目的寄存器，源操作数

**MOV** 指令可完成从另一个寄存器、被移位的寄存器或将一个立即数加载到目的寄存器。其中**S**选项决定指令的操作是否影响**CPSR** 中条件标志位的值，当没有**S** 时指令不更新**CPSR** 中条件标志位的值。 指令示例： MOV R1，R0 ；将寄存器R0的值传送到寄存器R1 MOV PC，R14 ；将寄存器R14的值传送到PC，常用于子程序返回 MOV R1，R0，LSL＃3 ；将寄存器R0的值左移3位后传送到R1

**2、 MVN 指令**

**MVN** 指令的格式为：

**MVN{**条件**}{S}** 目的寄存器，源操作数

**MVN** 指令可完成从另一个寄存器、被移位的寄存器、或将一个立即数加载到目的寄存器。与**MOV**指令不同之处是在传送之前按位被取反了，即把一个被取反的值传送到目的寄存器中。其中**S** 决定指令的操作是否影响**CPSR** 中条件标志位的值，当没有**S** 时指令不更新**CPSR** 中条件标志位的值。 指令示例： MVN R0，＃0 ；将立即数0取反传送到寄存器R0中，完成后R0=-1

**3、 CMP 指令**

**CMP** 指令的格式为：

**CMP{**条件**}** 操作数**1**，操作数**2**

**CMP** 指令用于把一个寄存器的内容和另一个寄存器的内容或立即数进行比较，同时更新**CPSR** 中条件标志位的值。该指令进行一次减法运算，但不存储结果，只更改条件标志位。标志位表示的是操作数**1** 与操作数**2** 的关系(大、小、相等)，例如，当操作数**1** 大于操作操作数**2**，则此后的有**GT**后缀的指令将可以执行。

指令示例：

CMP R1，R0 ；将寄存器R1的值与寄存器R0的值相减，并根据结果设置CPSR的标志位

CMP R1，＃100 ；将寄存器R1的值与立即数100相减，并根据结果设置CPSR的标志位

**4、 CMN 指令**

**CMN** 指令的格式为：

**CMN{**条件**}** 操作数**1**，操作数**2**

**CMN** 指令用于把一个寄存器的内容和另一个寄存器的内容或立即数取反后进行比较，同时更新**CPSR** 中条件标志位的值。该指令实际完成操作数**1** 和操作数**2** 相加，并根据结果更改条件标志位。指令示例： CMN R1，R0 ；将寄存器R1的值与寄存器R0的值相加，并根据结果设置CPSR的标志位CMN R1，＃100 ；将寄存器R1的值与立即数100相加，并根据结果设置CPSR的标志位

**5、 TST 指令**

**TST** 指令的格式为：

**TST{**条件**}** 操作数**1**，操作数**2**

**TST** 指令用于把一个寄存器的内容和另一个寄存器的内容或立即数



PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

进行按位的与运算，并根据运算结果更新**CPSR** 中条件标志位的值。操作数**1** 是要测试的数据，而操作数**2** 是一个位掩码，该指令一般用来检测是否设置了特定的位。 指令示例： TST R1，＃%1 ；用于测试在寄存器R1中是否设置了最低位（%表示二进制数）TST R1，＃0xffe ；将寄存器R1的值与立即数0xffe按位与，并根据结果设置CPSR的标志位

**6、 TEQ 指令**

**TEQ** 指令的格式为：

**TEQ{条件} 操作数1，操作数2**

**TEQ** 指令用于把一个寄存器的内容和另一个寄存器的内容或立即数进行按位的异或运算，并根据运算结果更新**CPSR** 中条件标志位的值。该指令通常用于比较操作数**1** 和操作数**2** 是否相等。 指令示例： TEQ R1，R2 ；将寄存器R1的值与寄存器R2的值按位异或，并根据结果设置CPSR的标志位

**7、 ADD 指令**

**ADD** 指令的格式为： **ADD{条件}{S}** 目的寄存器，操作数**1**，操作数**2** **ADD** 指令用于把两个操作数相加，并将结果存放到目的寄存器中。操作数1 应是一个寄存器，操作数2 可以是一个寄存器，被移位的寄存器，或一个立即数。 指令示例： ADD R0，R1，R2 ； R0 = R1 + R2 ADD R0，R1，#256 ； R0 = R1 + 256 ADD R0，R2，R3，LSL#1 ； R0 = R2 + (R3 << 1)

**8、 ADC 指令**

**ADC** 指令的格式为：

**ADC{条件}{S}** 目的寄存器，操作数**1**，操作数**2**

**ADC** 指令用于把两个操作数相加，再加上**CPSR** 中的**C** 条件标志位的值，并将结果存放到目的寄存器中。它使用一个进位标志位，这样就可以做比**32** 位大的数的加法，注意不要忘记设置**S** 后缀来更改进位标志。操作数**1** 应是一个寄存器，操作数**2** 可以是一个寄存器，被移位的寄存器，或一个立即数。

以下指令序列完成两个**128** 位数的加法，第一个数由高到低存放在寄存器**R7**～**R4**，第二个数由高到低存放在寄存器**R11**～**R8**，运算结果由高到低存放在寄存器**R3**～**R0**： ADDS R0，R4，R8； 加低端的字ADCS R1，R5，R9 ； 加第二个字，带进位ADCS R2，R6，R10 ； 加第三个字，带进位ADC R3，R7，R11 ； 加第四个字，带进位

**9、 SUB 指令**

**SUB** 指令的格式为： **SUB{条件}{S}** 目的寄存器，操作数**1**，操作数**2**

**SUB** 指令用于把操作数1 减去操作数2，并将结果存放到目的寄存器中。操作数1 应是一个寄存器，操作数2 可以是一个寄存器，被移位的寄存器，或一个立即数。该指令可用于有符号数或无符号数的减法运算。

指令示例： SUB R0，R1，R2 ； R0 = R1 - R2 SUB R0，R1，#256 ； R0 = R1 - 256 SUB R0，R2，R3，LSL#1 ； R0 = R2 - (R3 << 1)

**10、SBC指令**

**SBC** 指令的格式为：

**SBC{条件}{S}** 目的寄存器，操作数**1**，操作数**2**

**SBC** 指令用于把操作数1 减去操作数2，再减去**CPSR** 中的**C** 条件标志位的反码，并将结果存放到目的寄存器中。操作数1 应是一个寄存器，操作数2 可以是一个寄存器，被移位的寄存器，或一个立即数。该指令使用进位标志来表示借位，这样就可以做大于32 位的减法，注意不要忘记设置**S**后缀来更改进位标志。该指令可用于有符号数或无符号数的减法运算。

指令示例： SUBS R0，R1，R2 ； R0 = R1 - R2 -！C，并根据结果设置CPSR的进位标志位

**11、RSB指令**



PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

**RSB** 指令的格式为： **RSB{条件}{S}** 目的寄存器，操作数**1**，操作数**2** **RSB** 指令称为逆向减法指令，用于把操作数2 减去操作数1，并将结果存放到目的寄存器中。操作数1 应是一个寄存器，操作数2 可以是一个寄存器，被移位的寄存器，或一个立即数。该指令可用于有符号数或无符号数的减法运算。 指令示例： RSB R0，R1，R2； R0 = R2 – R1 RSB R0，R1，#256 ； R0 = 256 – R1 RSB R0，R2，R3，LSL#1 ； R0 = (R3 << 1) - R2

**12、RSC指令**

**RSC** 指令的格式为：

**RSC{条件}{S}** 目的寄存器，操作数**1**，操作数**2**

**RSC** 指令用于把操作数2 减去操作数1，再减去**CPSR** 中的**C** 条件标志位的反码，并将结果存放到目的寄存器中。操作数1 应是一个寄存器，操作数2 可以是一个寄存器，被移位的寄存器，或一个立即数。该指令使用进位标志来表示借位，这样就可以做大于32 位的减法，注意不要忘记设置**S**后缀来更改进位标志。该指令可用于有符号数或无符号数的减法运算。

指令示例： RSC R0，R1，R2 ； R0 = R2 – R1 -！C

**13、AND指令**

**AND** 指令的格式为：

**AND{条件}{S}** 目的寄存器，操作数**1**，操作数**2**

**AND** 指令用于在两个操作数上进行逻辑与运算，并把结果放置到目的寄存器中。操作数1 应是一个寄存器，操作数2 可以是一个寄存器，被移位的寄存器，或一个立即数。该指令常用于屏蔽操作数1 的某些位。 指令示例： AND R0，R0，＃3； 该指令保持R0的0、1位，其余位清零。

**14、ORR指令**

**ORR** 指令的格式为：

**ORR{条件}{S}** 目的寄存器，操作数**1**，操作数**2**

**ORR** 指令用于在两个操作数上进行逻辑或运算，并把结果放置到目的寄存器中。操作数1 应是一个寄存器，操作数2 可以是一个寄存器，被移位的寄存器，或一个立即数。该指令常用于设置操作数1 的某些位。

指令示例：

ORR R0，R0，＃3 ； 该指令设置R0的0、1位，其余位保持不变。

**15、EOR指令**

**EOR** 指令的格式为：

**EOR{条件}{S}** 目的寄存器，操作数**1**，操作数**2**

**EOR** 指令用于在两个操作数上进行逻辑异或运算，并把结果放置到目的寄存器中。操作数1 应是一个寄存器，操作数2 可以是一个寄存器，被移位的寄存器，或一个立即数。该指令常用于反转操作数1 的某些位。 指令示例： EOR R0，R0，＃3 ； 该指令反转R0的0、1位，其余位保持不变。

**16、BIC指令**

**BIC** 指令的格式为：

**BIC{条件}{S}** 目的寄存器，操作数**1**，操作数**2**

**BIC** 指令用于清除操作数1 的某些位，并把结果放置到目的寄存器中。操作数1 应是一个寄存器，操作数2 可以是一个寄存器，被移位的寄存器，或一个立即数。操作数2 为32 位的掩码，如果在掩码中设置了某一位，则清除这一位。未设置的掩码位保持不变。 指令示例： BIC R0，R0，＃％



PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

1011 ； 该指令清除 R0 中的位 0、1、和 3，其余的位保持不变。

### 3.3.3 乘法指令与乘加指令

ARM 微处理器支持的乘法指令与乘加指令共有6 条，可分为运算结果为32 位和运算结果为64 位两类，与前面的数据处理指令不同，指令中的所有操作数、目的寄存器必须为通用寄存器，不能对操作数使用立即数或被移位的寄存器，同时，目的寄存器和操作数1 必须是不同的寄存器。

乘法指令与乘加指令共有以下6 条：

— **MUL 32** 位乘法指令
— **MLA 32** 位乘加指令
— **SMULL 64** 位有符号数乘法指令
— **SMLAL 64** 位有符号数乘加指令
— **UMULL 64** 位无符号数乘法指令
— **UMLAL 64** 位无符号数乘加指令

**1、 MUL 指令**

**MUL** 指令的格式为： **MUL{**条件**}{S}** 目的寄存器，操作数**1**，操作数**2** **MUL** 指令完成将操作数**1** 与操作数**2** 的乘法运算，并把结果放置到目的寄存器中，同时可以根据
运算结果设置CPSR 中相应的条件标志位。
其中，操作数1 和操作数2 均为32 位的有符
号数或无符号数。 指令示例： MUL R0，R1，
R2 ； R0 = R1 × R2 MULS R0，R1，R2 ； R0 =
R1 × R2，同时设置CPSR中的相关条件标志位

**2、 MLA 指令**

**MLA** 指令的格式为：

**MLA{**条件**}{S}** 目的寄存器，操作数**1**，操作数**2**，操作数**3**

**MLA** 指令完成将操作数**1** 与操作数**2** 的乘法运算，再将乘积加上操作数**3**，并把结果放置到目的寄存器中，同时可以根据运算结果设置CPSR 中相应的条件标志位。其中，操作数1 和操作数2 均为32 位的有符号数或无符号数。

指令示例：

MLA R0，R1，R2，R3 ； R0 = R1 × R2 + R3
MLAS R0，R1，R2，R3 ； R0 = R1 × R2 + R3，同时设置CPSR中的相关条件标志位

**3、 SMULL 指令**

**SMULL** 指令的格式为：

**SMULL{**条件**}{S}** 目的寄存器**Low**，目的寄存器低**High**，操作数**1**，操作数**2**

**SMULL** 指令完成将操作数**1** 与操作数**2** 的乘法运算，并把结果的低32 位放置到目的寄存器Low
中，结果的高32 位放置到目的寄存器High 中，同时可
以根据运算结果设置CPSR 中相应的条件标志位。其
中，操作数1 和操作数2 均为32 位的有符号数。 指令
示例： SMULL R0，R1，R2，R3 ； R0 = （R2 × R3）的
低32位；R1 = （R2 × R3）的高32位

**4、 SMLAL 指令**

**SMLAL** 指令的格式为：

**SMLAL{**条件**}{S}** 目的寄存器**Low**，目的寄存器低**High**，操作数**1**，操作数**2**

**SMLAL** 指令完成将操作数**1** 与操作数**2** 的乘法运算，并把结果的低32 位同目的寄存器Low 中的值相加后又放置到目的寄存器Low 中，结果的高32 位同目的寄存器High 中的值相加后又放置到目的寄存器High 中，同时可以根据运算结果设置CPSR 中相应的条件标志位。其中，操作数1 和操作数2 均为32 位



PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

的有符号数。

对于目的寄存器**Low**，在指令执行前存放**64** 位加数的低**32** 位，指令执行后存放结果的低**32** 位。对于目的寄存器**High**，在指令执行前存放**64** 位加数的高**32** 位，指令执行后存放结果的高**32**位。 指令示例： SMLAL R0，R1，R2，R3 ；R0 = （R2 × R3）的低32位＋ R0 ；R1 = （R2 × R3）的高32位＋ R1

**5、 UMULL 指令**

**UMULL** 指令的格式为：

**UMULL{条件}{S}** 目的寄存器**Low**，目的寄存器低**High**，操作数**1**，操作数**2**

**UMULL** 指令完成将操作数**1** 与操作数**2** 的乘法运算，并把结果的低32 位放置到目的寄存器Low 中，结果的高32 位放置到目的寄存器High 中，同时可以根据运算结果设置CPSR 中相应的条件标志位。其中，操作数1 和操作数2 均为32 位的无符号数。 指令示例： UMULL R0，R1，R2，R3 ；R0 = （R2 × R3）的低32位；R1 = （R2 × R3）的高32位

**6、 UMLAL 指令**

**UMLAL** 指令的格式为：

**UMLAL{条件}{S}** 目的寄存器**Low**，目的寄存器低**High**，操作数**1**，操作数**2**

**UMLAL** 指令完成将操作数**1** 与操作数**2** 的乘法运算，并把结果的低32 位同目的寄存器Low 中的值相加后又放置到目的寄存器Low 中，结果的高32 位同目的寄存器High 中的值相加后又放置到目的寄存器High 中，同时可以根据运算结果设置CPSR 中相应的条件标志位。其中，操作数1 和操作数2 均为32 位的无符号数。

对于目的寄存器**Low**，在指令执行前存放**64** 位加数的低**32** 位，指令执行后存放结果的低**32** 位。

对于目的寄存器**High**，在指令执行前存放**64** 位加数的高**32** 位，指令执行后存放结果的高**32**位。

指令示例： UMLAL R0，R1，R2，R3；R0 = （R2 × R3）的低32位＋ R0；R1 = （R2 × R3）的高32位＋ R1

### 3.3.4 程序状态寄存器访问指令

ARM 微处理器支持程序状态寄存器访问指令，用于在程序状态寄存器和通用寄存器之间传送数据，程序状态寄存器访问指令包括以下两条：

— **MRS** 程序状态寄存器到通用寄存器的数据传送指令

— **MSR** 通用寄存器到程序状态寄存器的数据传送指令

**1、 MRS 指令**

**MRS** 指令的格式为：

**MRS{条件}** 通用寄存器，程序状态寄存器（**CPSR**或**SPSR**）

**MRS** 指令用于将程序状态寄存器的内容传送到通用寄存器中。该指令一般用在以下几种情况：

－当需要改变程序状态寄存器的内容时，可用MRS 将程序状态寄存器的内容读入通用寄存器，修改后再写回程序状态寄存器。－当在异常处理或进程切换时，需要保存程序状态寄存器的值，可先用该指令读出程序状态寄存器的值，然后保存。 指令示例：

MRS R0，CPSR ；传送CPSR的内容到R0 MRS R0，SPSR ；传送SPSR的内容到R0

**2、 MSR 指令**



PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

**MSR** 指令的格式为：

**MSR{条件}** 程序状态寄存器（**CPSR**或**SPSR**）**_**<域>，操作数

**MSR** 指令用于将操作数的内容传送到程序状态寄存器的特定域中。其中，操作数可以为通用寄存器或立即数。<域>用于设置程序状态寄存器中需要操作的位，**32** 位的程序状态寄存器可分为**4** 个域：

位**[31**：**24]**为条件标志位域，用**f** 表示；

位**[23**：**16]**为状态位域，用**s** 表示；

位**[15**：**8]**为扩展位域，用**x** 表示；

位**[7**：**0]**为控制位域，用**c** 表示；

该指令通常用于恢复或改变程序状态寄存器的内容，在使用时，一般要在**MSR** 指令中指明将要操作的域。

指令示例：

MSR CPSR，R0 ；传送R0的内容到CPSR MSR SPSR，R0 ；传送R0的内容到SPSR MSR CPSR_c，R0 ；传送R0的内容到SPSR，但仅仅修改CPSR中的控制位域

## 3.3.5 加载/存储指令

ARM 微处理器支持加载/存储指令用于在寄存器和存储器之间传送数据，加载指令用于将存储器中的数据传送到寄存器，存储指令则完成相反的操作。常用的加载存储指令如下：

— **LDR** 字数据加载指令
— **LDRB** 字节数据加载指令
— **LDRH** 半字数据加载指令
— **STR** 字数据存储指令
— **STRB** 字节数据存储指令
— **STRH** 半字数据存储指令

**1、LDR指令**

**LDR** 指令的格式为： **LDR{条件}** 目的寄存器，**<**存储器地址**>** **LDR** 指令用于从存储器中将一个**32** 位的字数据传送到目的寄存器中。该指令通常用于从存储器
中读取**32** 位的字数据到通用寄存器，然后对数据进行处理。当程序计数器**PC** 作为目的寄存器时，指令从存储器中读取的字数据被当作目的地址，从而可以实现程序流程的跳转。该指令在程序设计中比较常用，且寻址方式灵活多样，请读者认真掌握。

指令示例： LDR R0，［R1］；将存储器地址为R1的字数据读入寄存器R0。LDR R0，［R1，R2］；将存储器地址为R1+R2的字数据读入寄存器R0。LDR R0，［R1，＃8］；将存储器地址为R1+8的字数据读入寄存器R0。LDR R0，［R1，R2］！ ；将存储器地址为R1+R2的字数据读入寄存器R0，并将新地址R1 ＋R2写入R1。LDR R0，［R1，＃8］！ ；将存储器地址为R1+8的字数据读入寄存器R0，并将新地址R1 ＋8写入R1。LDR R0，［R1］，R2 ；将存储器地址为R1的字数据读入寄存器R0，并将新地址R1＋ R2写入R1。LDR R0，［R1，R2，LSL＃2］！ ；将存储器地址为R1＋R2×4的字数据读入寄存器R0，并将新地址R1＋R2×4写入R1。LDR R0，［R1］，R2，LSL＃2 ；将存储器地址为R1的字数据读入寄存器R0，并将新地址R1＋ R2×4写入R1。

**2、LDRB指令**

**LDRB** 指令的格式为： **LDR{条件}B** 目的寄存器，**<**存储器地址**>** **LDRB** 指令用于从存储器中将一个**8** 位的字节数据传送到目的寄存器中，同时将寄存器的高**24**
位清零。该指令通常用于从存储器中读取**8** 位的字节数据到通用寄存器，然后对数据进行处理。当程序计数器**PC** 作为目的寄存器时，指令从存储器中读取的字数据被当作目的地址，从而可以实现程序流程的跳转。

指令示例： LDRB R0，［R1］；将存储器地址为R1的字节数据读入寄存器R0，并将R0的高24位清零。LDRB R0，［R1，＃8］；将存储器地址为R1＋8的字节数据读入寄存器R0，并将R0的高24位清零。

**3、LDRH指令**

**LDRH** 指令的格式为： **LDR{条件}H** 目的寄存器，**<**存储器地址**>** **LDRH** 指令用于从存储器中将一个**16** 位



PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

的半字数据传送到目的寄存器中，同时将寄存器的高**16**
位清零。该指令通常用于从存储器中读取**16** 位的半字数据到通用寄存器，然后对数据进行处理。当程序计数器**PC** 作为目的寄存器时，指令从存储器中读取的字数据被当作目的地址，从而可以实现程序流程的跳转。

指令示例： LDRH R0，[R1]；将存储器地址为R1的半字数据读入寄存器R0，并将R0的高16位清零。LDRH R0，[R1，#8]；将存储器地址为R1＋8的半字数据读入寄存器R0，并将R0的高16位清零。LDRH R0，[R1，R2]；将存储器地址为R1＋R2的半字数据读入寄存器R0，并将R0的高16位清零。

**4、STR指令**

**STR** 指令的格式为： **STR{**条件**}** 源寄存器，**<**存储器地址**> STR** 指令用于从源寄存器中将一个**32** 位的字数据传送到存储器中。该指令在程序设计中比较常
用，且寻址方式灵活多样，使用方式可参考指令**LDR**。 指令示例：

STR R0，[R1]，#8 ；将R0中的字数据写入以R1为地址的存储器中，并将新地址R1＋8写入R1。

STR R0，[R1，#8]；将R0中的字数据写入以R1＋8为地址的存储器中。**5、STRB指令**

**STRB** 指令的格式为：

**STR{**条件**}B** 源寄存器，**<**存储器地址**>**

**STRB** 指令用于从源寄存器中将一个**8** 位的字节数据传送到存储器中。该字节数据为源寄存器中的低**8**
位。

指令示例：

STRB R0，[R1]；将寄存器R0中的字节数据写入以R1为地址的存储器中。

STRB R0，[R1，#8]；将寄存器R0中的字节数据写入以R1＋8为地址的存储器中。**6、STRH指令**

**STRH** 指令的格式为：

**STR{**条件**}H** 源寄存器，**<**存储器地址**>**

**STRH** 指令用于从源寄存器中将一个**16** 位的半字数据传送到存储器中。该半字数据为源寄存器中的低
**16** 位。

指令示例：

STRH R0，[R1]；将寄存器R0中的半字数据写入以R1为地址的存储器中。

STRH R0，[R1，#8]；将寄存器R0中的半字数据写入以R1＋8为地址的存储器中。

### 3.3.6 批量数据加载/存储指令

ARM 微处理器所支持批量数据加载/存储指令可以一次在一片连续的存储器单元和多个寄存器之间传送数据，批量加载指令用于将一片连续的存储器中的数据传送到多个寄存器，批量数据存储指令则完成相反的操作。常用的加载存储指令如下：

— **LDM** 批量数据加载指令

— **STM** 批量数据存储指令

**LDM（或STM）指令**

**LDM**（或**STM**）指令的格式为：

**LDM**（或**STM**）**{**条件**}{**类型**}** 基址寄存器**{**！**}**，寄存器列表**{^}**

**LDM**（或**STM**）指令用于从由基址寄存器所指示的一片连续存储器到寄存器列表所指示的多个寄存器之间传送数据，该指令的常见用途是将多个寄存器的内容入栈或出栈。其中，**{**类型**}**为以下几种情况：

**IA** 每次传送后地址加**1**；

**IB** 每次传送前地址加**1**；

**DA** 每次传送后地址减**1**；

**DB** 每次传送前地址减**1**；

**FD** 满递减堆栈；



PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

**ED** 空递减堆栈；

**FA** 满递增堆栈；

**EA** 空递增堆栈；

**{！}**为可选后缀，若选用该后缀，则当数据传送完毕之后，将最后的地址写入基址寄存器，否则基址寄存器的内容不改变。

基址寄存器不允许为**R15**，寄存器列表可以为**R0**～**R15** 的任意组合。

**{^}**为可选后缀，当指令为**LDM** 且寄存器列表中包含**R15**，选用该后缀时表示：除了正常的数据传送之外，还将**SPSR** 复制到**CPSR**。同时，该后缀还表示传入或传出的是用户模式下的寄存器，而不是当前模式下的寄存器。

指令示例：

```
STMFD R13!，{R0，R4-R12，LR} ；将寄存器列表中的寄存器（R0，R4到R12，LR）存入堆栈。
LDMFD R13!，{R0，R4-R12，PC} ；将堆栈内容恢复到寄存器（R0，R4到R12，LR）。
```

## 3.3.7 数据交换指令

ARM 微处理器所支持数据交换指令能在存储器和寄存器之间交换数据。数据交换指令有如下两条：

— **SWP** 字数据交换指令

— **SWPB** 字节数据交换指令

**1、SWP指令**

**SWP** 指令的格式为：

**SWP{条件} 目的寄存器，源寄存器1，[源寄存器2]**

**SWP** 指令用于将源寄存器**2** 所指向的存储器中的字数据传送到目的寄存器中，同时将源寄存器**1**中的字数据传送到源寄存器**2** 所指向的存储器中。显然，当源寄存器**1** 和目的寄存器为同一个寄存器时，指令交换该寄存器和存储器的内容。

指令示例：

```
SWP R0，R1，[R2] ；将R2所指向的存储器中的字数据传送到R0，同时将R1中的字数据传
送到R2所指向的存储单元。
SWP R0，R0，[R1] ；该指令完成将R1所指向的存储器中的字数据与R0中的字数据交换。
```

**2、SWPB指令**

**SWPB** 指令的格式为：

**SWP{条件}B 目的寄存器，源寄存器1，[源寄存器2]**

**SWPB** 指令用于将源寄存器**2** 所指向的存储器中的字节数据传送到目的寄存器中，目的寄存器的高**24** 清零，同时将源寄存器**1** 中的字节数据传送到源寄存器**2** 所指向的存储器中。显然，当源寄存器**1** 和目的寄存器为同一个寄存器时，指令交换该寄存器和存储器的内容。

指令示例：

```
SWPB R0，R1，[R2] ；将R2所指向的存储器中的字节数据传送到R0，R0的高24位清零，
同时将R1中的低8位数据传送到R2所指向的存储单元。
SWPB R0，R0，[R1] ；该指令完成将R1所指向的存储器中的字节数据与R0中的低8位数据
交换。
```

## 3.3.8 移位指令（操作）

**ARM** 微处理器内嵌的桶型移位器（**Barrel Shifter**），支持数据的各种移位操作，移位操作在**ARM** 指令集中不作为单独的指令使用，它只能作为指令格式中是一个字段，在汇编语言中表示为指令中的选项。例如，数据处理指令的第二个操作数为寄存器时，就可以加入移位操作选项对它进行各种移位操作。移位操作包括如下**6** 种类型，ASL 和LSL 是等价的，可以自由互换：



PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

— LSL 逻辑左移
— ASL 算术左移
— LSR 逻辑右移
— ASR 算术右移
— ROR 循环右移
— RRX 带扩展的循环右移

**1、LSL（或ASL）操作**

**LSL**（或**ASL**）操作的格式为：

通用寄存器，LSL（或ASL） 操作数

**LSL**（或**ASL**）可完成对通用寄存器中的内容进行逻辑（或算术）的左移操作，按操作数所指定的数量向左移位，低位用零来填充。其中，操作数可以是通用寄存器，也可以是立即数（0～ 31）。

操作示例：MOV R0，R1，LSL#2 ；将R1中的内容左移两位后传送到R0中。

**2、LSR操作**

**LSR** 操作的格式为：

通用寄存器，LSR 操作数

**LSR** 可完成对通用寄存器中的内容进行右移的操作，按操作数所指定的数量向右移位，左端用零来填充。其中，操作数可以是通用寄存器，也可以是立即数（0～31）。

操作示例：

MOV R0，R1，LSR#2 ；将R1中的内容右移两位后传送到R0中，左端用零来填充。**3、ASR操作**

**ASR** 操作的格式为：

通用寄存器，ASR 操作数

**ASR** 可完成对通用寄存器中的内容进行右移的操作，按操作数所指定的数量向右移位，左端用第31 位的值来填充。其中，操作数可以是通用寄存器，也可以是立即数（0～31）。

操作示例：

MOV R0，R1，ASR#2 ；将R1中的内容右移两位后传送到R0中，左端用第31位的值来填充。**4、ROR操作**

**ROR** 操作的格式为：

通用寄存器，ROR 操作数

**ROR** 可完成对通用寄存器中的内容进行循环右移的操作，按操作数所指定的数量向右循环移位，左端用右端移出的位来填充。其中，操作数可以是通用寄存器，也可以是立即数（0～31）。显然，当进行32 位的循环右移操作时，通用寄存器中的值不改变。操作示例：MOV R0，R1，ROR#2 ；将R1中的内容循环右移两位后传送到R0中。

**5、RRX操作**

**RRX** 操作的格式为：

通用寄存器，RRX 操作数

**RRX** 可完成对通用寄存器中的内容进行带扩展的循环右移的操作，按操作数所指定的数量向右循环移位，左端用进位标志位C 来填充。其中，操作数可以是通用寄存器，也可以是立即数（0～31）。

操作示例：

MOV R0，R1，RRX#2 ；将R1中的内容进行带扩展的循环右移两位后传送到R0中。

### 3.3.9 协处理器指令

ARM 微处理器可支持多达16 个协处理器，用于各种协处理操作，在程序执行的过程中，每个协处理器只执行针对自身的协处理指令，忽略ARM 处理器和其他协处理器的指令。

ARM 的协处理器指令主要用于ARM 处理器初始化ARM 协处理器的数据处理操作，以及在ARM 处理器的寄存器和协处理器的寄存器之间传送数据，和在ARM 协处理器的寄存器和存储器之间传送数据。



PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

ARM 协处理器指令包括以下5 条：

- — **CDP** 协处理器数操作指令
- — **LDC** 协处理器数据加载指令
- — **STC** 协处理器数据存储指令
- — **MCR ARM** 处理器寄存器到协处理器寄存器的数据传送指令
- — **MRC** 协处理器寄存器到**ARM** 处理器寄存器的数据传送指令

**1、CDP指令**

**CDP** 指令的格式为：

**CDP{**条件**}** 协处理器编码，协处理器操作码**1**，目的寄存器，源寄存器**1**，源寄存器**2**，协处理器操作码**2**。

**CDP** 指令用于**ARM** 处理器通知**ARM** 协处理器执行特定的操作**,**若协处理器不能成功完成特定的操作，则产生未定义指令异常。其中协处理器操作码**1** 和协处理器操作码**2** 为协处理器将要执行的操作，目的寄存器和源寄存器均为协处理器的寄存器，指令不涉及**ARM** 处理器的寄存器和存储器。

指令示例： CDP P3，2，C12，C10，C3，4 ；该指令完成协处理器P3的初始化

**2、LDC指令**

**LDC** 指令的格式为：

**LDC{**条件**}{L}** 协处理器编码**,**目的寄存器，**[**源寄存器**]**

**LDC** 指令用于将源寄存器所指向的存储器中的字数据传送到目的寄存器中，若协处理器不能成功完成传送操作，则产生未定义指令异常。其中，**{L}**选项表示指令为长读取操作，如用于双精度数据的传输。 指令示例： LDC P3，C4，[R0]；将ARM处理器的寄存器R0所指向的存储器中的字数据传送到协处理器P3的寄存器C4中。

**3、STC指令**

**STC** 指令的格式为：

**STC{**条件**}{L}** 协处理器编码**,**源寄存器，**[**目的寄存器**]**

**STC** 指令用于将源寄存器中的字数据传送到目的寄存器所指向的存储器中，若协处理器不能成功完成传送操作，则产生未定义指令异常。其中，**{L}**选项表示指令为长读取操作，如用于双精度数据的传输。 指令示例： STC P3，C4，[R0]；将协处理器P3的寄存器C4中的字数据传送到ARM处理器的寄存器R0所指向的存储器中。

**4、MCR指令**

**MCR** 指令的格式为：

**MCR{**条件**}** 协处理器编码，协处理器操作码**1**，源寄存器，目的寄存器**1**，目的寄存器**2**，协处理器操作码**2**。

**MCR** 指令用于将**ARM** 处理器寄存器中的数据传送到协处理器寄存器中**,**若协处理器不能成功完成操作，则产生未定义指令异常。其中协处理器操作码**1** 和协处理器操作码**2** 为协处理器将要执行的操作，源寄存器为**ARM** 处理器的寄存器，目的寄存器**1** 和目的寄存器**2** 均为协处理器的寄存器。

指令示例：

MCR P3，3，R0，C4，C5，6 ；该指令将ARM处理器寄存器R0中的数据传送到协处理器P3的寄存器C4和C5中。

**5、MRC指令**

**MRC** 指令的格式为：

**MRC{**条件**}** 协处理器编码，协处理器操作码**1**，目的寄存器，源寄存器**1**，源寄存器**2**，协处理器操作码**2**。

**MRC** 指令用于将协处理器寄存器中的数据传送到**ARM** 处理器寄存器中**,**若协处理器不能成功完成操作，则产生未定义指令异常。其中协处理器操作码**1** 和协处理器操作码**2** 为协处理器将要执行的操作，目的寄



PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

存器为ARM 处理器的寄存器，源寄存器1 和源寄存器2 均为协处理器的寄存器。

指令示例：

MRC P3，3，R0，C4，C5，6 ；该指令将协处理器P3的寄存器中的数据传送到ARM处理器寄存器中。

### 3.3.10 异常产生指令

ARM 微处理器所支持的异常指令有如下两条：

— **SWI** 软件中断指令

— **BKPT** 断点中断指令

**1、SWI指令**

**SWI** 指令的格式为：

**SWI{条件} 24**位的立即数

**SWI** 指令用于产生软件中断，以便用户程序能调用操作系统的系统例程。操作系统在**SWI** 的异常处理程序中提供相应的系统服务，指令中24 位的立即数指定用户程序调用系统例程的类型，相关参数通过通用寄存器传递，当指令中**24** 位的立即数被忽略时，用户程序调用系统例程的类型由通用寄存器**R0** 的内容决定，同时，参数通过其他通用寄存器传递。

指令示例：

SWI 0x02 ；该指令调用操作系统编号位02的系统例程。

**2、BKPT指令**

**BKPT** 指令的格式为：

**BKPT** **16** 位的立即数

**BKPT** 指令产生软件断点中断，可用于程序的调试。

## 3.4 Thumb 指令及应用

为兼容数据总线宽度为16 位的应用系统，ARM 体系结构除了支持执行效率很高的32 位ARM 指令集以外，同时支持16 位的Thumb 指令集。Thumb 指令集是ARM 指令集的一个子集，允许指令编码为16 位的长度。与等价的32 位代码相比较，Thumb 指令集在保留32 代码优势的同时，大大的节省了系统的存储空间。

所有的Thumb 指令都有对应的ARM 指令，而且Thumb 的编程模型也对应于ARM 的编程模型，在应用程序的编写过程中，只要遵循一定调用的规则，Thumb 子程序和ARM 子程序就可以互相调用。当处理器在执行ARM 程序段时，称ARM 处理器处于ARM 工作状态，当处理器在执行Thumb 程序段时，称ARM 处理器处于Thumb 工作状态。

与ARM 指令集相比较，Thumb 指令集中的数据处理指令的操作数仍然是32 位，指令地址也为32 位，但Thumb 指令集为实现16 位的指令长度，舍弃了ARM 指令集的一些特性，如大多数的Thumb 指令是无条件执行的，而几乎所有的ARM 指令都是有条件执行的；大多数的Thumb 数据处理指令的目的寄存器与其中一个源寄存器相同。

由于Thumb 指令的长度为16 位，即只用ARM 指令一半的位数来实现同样的功能，所以，要实现特定的程序功能，所需的Thumb 指令的条数较ARM 指令多。在一般的情况下，Thumb 指令与ARM 指令的时间效率和空间效率关系为：

— **Thumb** 代码所需的存储空间约为**ARM** 代码的**60**％～**70**％

— **Thumb** 代码使用的指令数比**ARM** 代码多约**30**％～**40**％

— 若使用**32** 位的存储器，**ARM**代码比**Thumb** 代码快约**40**％

— 若使用**16** 位的存储器，**Thumb**代码比**ARM** 代码快约**40**％～**50**％

— 与**ARM** 代码相比较，使用**Thumb** 代码，存储器的功耗会降低约**30**％



PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

显然，ARM 指令集和Thumb 指令集各有其优点，若对系统的性能有较高要求，应使用32 位的存储系统和ARM 指令集，若对系统的成本及功耗有较高要求，则应使用16 位的存储系统和Thumb 指令集。当然，若两者结合使用，充分发挥其各自的优点，会取得更好的效果。

## 3.5 本章小节

本章系统的介绍了ARM 指令集中的基本指令，以及各指令的应用场合及方法，由基本指令还可以派生出一些新的指令，但使用方法与基本指令类似。与常见的如X86 体系结构的汇编指令相比较，ARM 指令系统无论是从指令集本身，还是从寻址方式上，都相对复杂一些。

Thumb 指令集作为ARM 指令集的一个子集，其使用方法与ARM 指令集类似，在此未作详细的描述，但这并不意味着Thumb 指令集不如ARM 指令集重要，事实上，他们各自有其自己的应用场合。



PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

# 第4 章 ARM 程序设计基础

**ARM** 编译器一般都支持汇编语言的程序设计和**C/C++**语言的程序设计，以及两者的混合编程。本章介绍**ARM** 程序设计的一些基本概念，如**ARM** 汇编语言的伪指令、汇编语言的语句格式和汇编语言的程序结构等，同时介绍**C/C**＋＋和汇编语言的混合编程等问题。

本章的主要内容：

－**ARM** 编译器所支持的伪指令

－汇编语言的语句格式

－汇编语言的程序结构

－相关的程序示例

## 4.1 ARM 汇编器所支持的伪指令

在ARM 汇编语言程序里，有一些特殊指令助记符，这些助记符与指令系统的助记符不同，没有相对应的操作码，通常称这些特殊指令助记符为伪指令，他们所完成的操作称为伪操作。伪指令在源程序中的作用是为完成汇编程序作各种准备工作的，这些伪指令仅在汇编过程中起作用，一旦汇编结束，伪指令的使命就完成。

在ARM 的汇编程序中，有如下几种伪指令：符号定义伪指令、数据定义伪指令、汇编控制伪指令、宏指令以及其他伪指令。

### 4.1.1 符号定义（Symbol Definition ）伪指令

符号定义伪指令用于定义ARM 汇编程序中的变量、对变量赋值以及定义寄存器的别名等操作。常见的符号定义伪指令有如下几种：

— 用于定义全局变量的GBLA、GBLL 和GBLS。

— 用于定义局部变量的LCLA、LCLL 和LCLS。

— 用于对变量赋值的SETA、SETL、SETS。

— 为通用寄存器列表定义名称的RLIST。

**1、 GBLA、GBLL和GBLS**

语法格式：GBLA（GBLL 或GBLS） 全局变量名GBLA、GBLL 和GBLS 伪指令用于定义一个ARM 程序中的全局变量，并将其初始化。其中：GBLA 伪指令用于定义一个全局的数字变量，并初始化为0；GBLL 伪指令用于定义一个全局的逻辑变量，并初始化为F（假）；GBLS 伪指令用于定义一个全局的字符串变量，并初始化为空；由于以上三条伪指令用于定义全局变量，因此在整个程序范围内变量名必须唯一。使用示例：

GBLA Test1 ；定义一个全局的数字变量，变量名为Test1 Test1 SETA 0xaa ；将该变量赋值为0xaa GBLL Test2 ；定义一个全局的逻辑变量，变量名为Test2 Test2 SETL {TRUE} ；将该变量赋值为真GBLS Test3 ；定义一个全局的字符串变量，变量名为Test3 Test3 SETS “Testing” ；将该变量赋值为“Testing”

**2、 LCLA、LCLL和LCLS**

语法格式：LCLA（LCLL 或LCLS） 局部变量名LCLA、LCLL 和LCLS 伪指令用于定义一个ARM 程序中的局部变量，并将其初始化。其中：LCLA 伪指令用于定义一个局部的数字变量，并初始化为0；LCLL 伪指令用于定义一个局部的逻辑变量，并初始化为F（假）；LCLS 伪指令用于定义一个局部的字符串变量，并初始化为空；

以上三条伪指令用于声明局部变量，在其作用范围内变量名必须唯一。使用示例：LCLA

Test4 ；声明一个局部的数字变量，变量名为Test4



PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

Test3 SETA 0xaa ；将该变量赋值为0xaa LCLL Test5 ；声明一个局部的逻辑变量，变量名为Test5 Test4 SETL {TRUE} ；将该变量赋值为真

LCLS Test6 ；定义一个局部的字符串变量，变量名为Test6 Test6 SETS “Testing” ；将该变量赋值为“Testing”

**3、 SETA、SETL和SETS**

语法格式：变量名 SETA（SETL 或SETS） 表达式伪指令SETA、SETL、SETS 用于给一个已经定义的全局变量或局部变量赋值。SETA 伪指令用于给一个数学变量赋值；SETL 伪指令用于给一个逻辑变量赋值；SETS 伪指令用于给一个字符串变量赋值；其中，变量名为已经定义过的全局变量或局部变量，表达式为将要赋给变量的值。使用示例：

LCLA Test3 ；声明一个局部的数字变量，变量名为Test3 Test3 SETA 0xaa ；将该变量赋值为0xaa LCLL Test4 ；声明一个局部的逻辑变量，变量名为Test4 Test4 SETL {TRUE} ；将该变量赋值为真

4、 RLIST 语法格式：名称RLIST {寄存器列表} RLIST 伪指令可用于对一个通用寄存器列表定义名称，使用该伪指令定义的名称可在ARM 指令LDM/STM 中使用。在LDM/STM 指令中，列表中的寄存器访问次序为根据寄存器的编号由低到高，而与列表中的寄存器排列次序无关。使用示例：RegList RLIST {R0-R5，R8，R10}；将寄存器列表名称定义为RegList，可在ARM指令LDM/STM中通过该名称访问寄存器列表。

## 4.1.2 数据定义（Data Definition）伪指令

数据定义伪指令一般用于为特定的数据分配存储单元，同时可完成已分配存储单元的初始化。常见的数据定义伪指令有如下几种：

— DCB 用于分配一片连续的字节存储单元并用指定的数据初始化。
— DCW（DCWU） 用于分配一片连续的半字存储单元并用指定的数据初始化。
— DCD（DCDU） 用于分配一片连续的字存储单元并用指定的数据初始化。
— DCFD（DCFDU ）用于为双精度的浮点数分配一片连续的字存储单元并用指定的数据初始化。
— DCFS（DCFSU） 用于为单精度的浮点数分配一片连续的字存储单元并用指定的数据初始化。
— DCQ（DCQU） 用于分配一片以8 字节为单位的连续的存储单元并用指定的数据初始化。
— SPACE 用于分配一片连续的存储单元
— MAP 用于定义一个结构化的内存表首地址
— FIELD 用于定义一个结构化的内存表的数据域

**1、 DCB**

语法格式：

标号DCB 表达式

DCB 伪指令用于分配一片连续的字节存储单元并用伪指令中指定的表达式初始化。其中，表达式可以为0～255 的数字或字符串。DCB 也可用“=”代替。使用示例：Str DCB “This is a test！” ；分配一片连续的字节存储单元并初始化。

**2、 DCW（或DCWU）**

语法格式：

标号DCW（或DCWU） 表达式

DCW（或DCWU）伪指令用于分配一片连续的半字存储单元并用伪指令中指定的表达式初始化。其中，表达式可以为程序标号或数字表达式。。

用DCW 分配的字存储单元是半字对齐的，而用DCWU 分配的字存储单元并不严格半字对齐。

使用示例：

DataTest DCW 1，2，3 ；分配一片连续的半字存储单元并初始化。



PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

**3、 DCD（或DCDU）**

语法格式：

标号DCD（或DCDU） 表达式

DCD（或DCDU ）伪指令用于分配一片连续的字存储单元并用伪指令中指定的表达式初始化。其中，表达式可以为程序标号或数字表达式。DCD 也可用“&”代替。

用DCD 分配的字存储单元是字对齐的，而用DCDU 分配的字存储单元并不严格字对齐。

使用示例：

```
DataTest DCD 4，5，6 ；分配一片连续的字存储单元并初始化。
```

**4、 DCFD（或DCFDU）**

语法格式：

标号DCFD（或DCFDU） 表达式

DCFD（或DCFDU ）伪指令用于为双精度的浮点数分配一片连续的字存储单元并用伪指令中指定的表达式初始化。每个双精度的浮点数占据两个字单元。

用DCFD 分配的字存储单元是字对齐的，而用DCFDU 分配的字存储单元并不严格字对齐。

使用示例：

```
FDataTest DCFD 2E115，-5E7 ；分配一片连续的字存储单元并初始化为指定的双精度数。
```

**5、 DCFS（或DCFSU）**

语法格式：

标号DCFS（或DCFSU） 表达式

DCFS（或DCFSU ）伪指令用于为单精度的浮点数分配一片连续的字存储单元并用伪指令中指定的表达式初始化。每个单精度的浮点数占据一个字单元。

用DCFS 分配的字存储单元是字对齐的，而用DCFSU 分配的字存储单元并不严格字对齐。

使用示例：

```
FDataTest DCFS 2E5，-5E－7 ；分配一片连续的字存储单元并初始化为指定的单精度数。
```

**6、 DCQ(或DCQU）**

语法格式：标号DCQ（或DCQU） 表达式DCQ（或DCQU ）伪指令用于分配一片以8 个字节为单位的连续存储区域并用伪指令中指定的

表达式初始化。用DCQ 分配的存储单元是字对齐的，而用DCQU 分配的存储单元并不严格字对齐。使用示例：

```
DataTest DCQ 100 ；分配一片连续的存储单元并初始化为指定的值。
```

**7、 SPACE**

语法格式：标号SPACE 表达式SPACE 伪指令用于分配一片连续的存储区域并初始化为0。其中，表达式为要分配的字节数。

SPACE 也可用“%”

代替。使用示例：

```
DataSpace SPACE
100 ；分配连续100
字节的存储单元并
初始化为0。
```

**8、 MAP**

语法格式：MAP 表达式{，基址寄存器} MAP 伪指令用于定义一个结构化的内存表的首地址。MAP 也可用“ ^ ”代替。表达式可以为程序中的标号或数学表达式，基址寄存器为可选项，当基址寄存器选项不存在时，

表达式的值即为内存表的首地址，当该选项存在时，内存表的首地址为表达式的值与基址寄存器的和。MAP 伪指令通常与FIELD 伪指令配合使用来定义结构化的内存表。使用示例：

```
MAP 0x100，R0 ；定义结构化内存表首地址的值为0x100＋R0。
```



PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

**9、 FILED**

语法格式：标号FIELD 表达式FIELD 伪指令用于定义一个结构化内存表中的数据域。FILED 也可用“#”代替。表达式的值为当前数据域在内存表中所占的字节数。FIELD 伪指令常与MAP 伪指令配合使用来定义结构化的内存表。MAP 伪指令定义内存表的首

地址，FIELD 伪指令定义内存表中的各个数据域，并可以为每个数据域指定一个标号供其他的指令引用。注意MAP 和FIELD 伪指令仅用于定义数据结构，并不实际分配存储单元。使用示例：

```
MAP 0x100 ； 定义结构化内存表首地址的值为0x100。 A FIELD 16 ； 定义A的长度为16字节，位置为0x100
B FIELD 32 ； 定义B的长度为32字节，位置为0x110 S FIELD 256 ； 定义S的长度为256字节，位置为0x130
```

## 4.1.3 汇编控制（Assembly Control ）伪指令

汇编控制伪指令用于控制汇编程序的执行流程，常用的汇编控制伪指令包括以下几条：

— IF、ELSE、ENDIF
— WHILE、WEND
— MACRO、MEND
— MEXIT

**1、 IF、ELSE、ENDIF**

语法格式：IF 逻辑

表达式指令序列1

ELSE

指令序列2 ENDIF IF、ELSE、ENDIF 伪指令能根据条件的成立与否决定是否执行某个指令序列。当IF 后面的逻

辑表达式为真，则执行指令序列1，否则执行指令序列2。其中，ELSE 及指令序列2 可以没有，此时，当IF 后面的逻辑表达式为真，则执行指令序列1，否则继续执行后面的指令。IF、ELSE、ENDIF 伪指令可以嵌套使用。使用示例：

```
GBLL Test ； 声明一个全局的逻辑变量，变量名为Test ……
 IF Test = TRUE
  指令序列1
  ELSE
  指令序列2
  ENDIF
```

**2、 WHILE、WEND**

语法格式：WHILE

逻辑表达式指令序

列

WEND

WHILE、WEND 伪指令能根据条件的成立与否决定是否循环执行某个指令序列。当WHILE 后面的逻辑表达式为真，则执行指令序列，该指令序列执行完毕后，再判断逻辑表达式的值，若为真则继续执行，一直到逻辑表达式的值为假。

WHILE、WEND 伪指令可以嵌套使用。使用示例：

```
GBLA Counter ； 声明一个全局的数学变量，变量名为
Counter Counter SETA 3 ；由变量Counter 控制循环次数 ……
 WHILE Counter < 10
  指令序列
 WEND
```

**3、 MACRO、MEND**



PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

语法格式：$标号宏名 $参数1，$参数2，……指令序列

MEND

MACRO、MEND 伪指令可以将一段代码定义为一个整体，称为宏指令，然后就可以在程序中通过宏指令多次调用该段代码。其中，$标号在宏指令被展开时，标号会被替换为用户定义的符号，宏指令可以使用一个或多个参数，当宏指令被展开时，这些参数被相应的值替换。

宏指令的使用方式和功能与子程序有些相似，子程序可以提供模块化的程序设计、节省存储空间并提高运行速度。但在使用子程序结构时需要保护现场，从而增加了系统的开销，因此，在代码较短且需要传递的参数较多时，可以使用宏指令代替子程序。

包含在MACRO 和MEND 之间的指令序列称为宏定义体，在宏定义体的第一行应声明宏的原型（包含宏名、所需的参数），然后就可以在汇编程序中通过宏名来调用该指令序列。在源程序被编译时，汇编器将宏调用展开，用宏定义中的指令序列代替程序中的宏调用，并将实际参数的值传递给宏定义中的形式参数。

MACRO、MEND 伪指令可以嵌套使用。

**4、 MEXIT**

语法格式：

MEXIT

MEXIT 用于从宏定义中跳转出去。

### 4.1.4 其他常用的伪指令

还有一些其他的伪指令，在汇编程序中经常会被使用，包括以下几条：

— AREA
— ALIGN

— CODE16 、CODE32
— ENTRY
— END
— EQU
— EXPORT（或GLOBAL）
— IMPORT
— EXTERN
— GET（或INCLUDE）
— INCBIN
— RN
— ROUT

**1、 AREA**

语法格式：

AREA 段名属性1，属性2，……

AREA 伪指令用于定义一个代码段或数据段。其中，段名若以数字开头，则该段名需用“|”括起来，如|1_test|。

属性字段表示该代码段（或数据段）的相关属性，多个属性用逗号分隔。常用的属性如下：

— CODE 属性：用于定义代码段，默认为READONLY。
— DATA 属性：用于定义数据段，默认为READWRITE。
— READONLY 属性：指定本段为只读，代码段默认为READONLY。
— READWRITE 属性：指定本段为可读可写，数据段的默认属性为READWRITE。
— ALIGN 属性：使用方式为ALIGN 表达式。在默认时，ELF（可执行连接文件）的代码段和数据段是按字对齐的，表达式的取值范围为0～31，相应的对齐方式为2 表达式次方。
— COMMON 属性：该属性定义一个通用的段，不包含任何的用户代码和数据。各源文件中



PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

同名的COMMON 段共享同一段存储单元。一个汇编语言程序至少要包含一个段，当程序太长时，也可以将程序分为多个代码段和数据段。使用示例：

AREA Init，CODE，READONLY 指令序列；该伪指令定义了一个代码段，段名为Init，属性为只读

**2、 ALIGN**

语法格式：ALIGN {表达式{，偏移量}} ALIGN 伪指令可通过添加填充字节的方式，使当前位置满足一定的对其方式|。其中，表达式的

值用于指定对齐方式，可能的取值为2 的幂，如1、2、4、8、16 等。若未指定表达式，则将当前位置对齐到下一个字的位置。偏移量也为一个数字表达式，若使用该字段，则当前位置的对齐方式为：2 的表达式次幂＋偏移量。

使用示例：AREA Init，CODE，READONLY，ALIEN＝3 ；指定后面的指令为8字节对齐。指令序列

END

**3、 CODE16、CODE32**

语法格式：CODE16 （或CODE32） CODE16 伪指令通知编译器，其后的指令序列为16 位的Thumb 指令。CODE32 伪指令通知编译器，其后的指令序列为32 位的ARM 指令。若在汇编源程序中同时包含ARM 指令和Thumb 指令时，可用CODE16 伪指令通知编译器其后

的指令序列为16 位的Thumb 指令，CODE32 伪指令通知编译器其后的指令序列为32 位的ARM 指令。因此，在使用ARM 指令和Thumb 指令混合编程的代码里，可用这两条伪指令进行切换，但注意他们只通知编译器其后指令的类型，并不能对处理器进行状态的切换。

使用示例：AREA Init，CODE，READONLY

```
                              ……
 CODE32                 ；通知编译器其后的指令为32位的ARM指令
 LDR     R0，＝NEXT＋1   ；将跳转地址放入寄存器R0
 BX      R0             ；程序跳转到新的位置执行，并将处理器切换到Thumb工作状态
   ……
 CODE16                 ；通知编译器其后的指令为16位的Thumb指令
NEXT     LDR   R3，＝0x3FF
   ……
 END                    ；程序结束
```

**4、 ENTRY**

语法格式：

ENTRY

**ENTRY** 伪指令用于指定汇编程序的入口点。在一个完整的汇编程序中至少要有一个ENTRY（也可以有多个，当有多个ENTRY 时，程序的真正入口点由链接器指定），但在一个源文件里最多只能有一个ENTRY（可以没有）。

使用示例：

AREA Init，CODE，READONLY

ENTRY ；指定应用程序的入口点

……

**5、 END**

语法格式：

END

END 伪指令用于通知编译器已经到了源程序的结尾。

使用示例：

AREA Init，CODE，READONLY ……

END ；指定应用程序的结尾



PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

**6、 EQU**

语法格式：名称EQU 表达式{，类型} EQU 伪指令用于为程序中的常量、标号等定义一个等效的字符名称，类似于C 语言中的＃define 。

其中EQU 可用“*”代替。名称为EQU 伪指令定义的字符名称，当表达式为32 位的常量时，可以指定表达式的数据类型，

可以有以下三种类型：CODE16 、CODE32 和DATA 使用示例：

Test EQU 50 ；定义标号Test的值为50 Addr EQU 0x55，CODE32 ；定义Addr的值为0x55，且该处为32位的ARM指令。

**7、 EXPORT（或GLOBAL）**

语法格式：EXPORT 标号{[WEAK]} EXPORT 伪指令用于在程序中声明一个全局的标号，该标号可在其他的文件中引用。EXPORT

可用GLOBAL 代替。标号在程序中区分大小写，[WEAK] 选项声明其他的同名标号优先于该标号被引用。使用示例：AREA Init，CODE，READONLY

**EXPORT** **Stest** ；声明一个可全局引用的标号**Stest**

……

END

**8、 IMPORT**

语法格式：

IMPORT 标号{[WEAK]}

**IMPORT** 伪指令用于通知编译器要使用的标号在其他的源文件中定义，但要在当前源文件中引用，而且无论当前源文件是否引用该标号，该标号均会被加入到当前源文件的符号表中。

标号在程序中区分大小写，[WEAK] 选项表示当所有的源文件都没有定义这样一个标号时，编译器也不给出错误信息，在多数情况下将该标号置为0，若该标号为B 或BL 指令引用，则将B 或BL 指令置为NOP 操作。

使用示例：AREA Init，CODE，READONLY **IMPORT Main** ；通知编译器当前文件要引用标号**Main**，但**Main** 在其他源文件中定义

……

END

**9、 EXTERN**

语法格式：EXTERN 标号{[WEAK]} EXTERN 伪指令用于通知编译器要使用的标号在其他的源文件中定义，但要在当前源文件中引用，如果当前源文件实际并未引用该标号，该标号就不会被加入到当前源文件的符号表中。

标号在程序中区分大小写，[WEAK] 选项表示当所有的源文件都没有定义这样一个标号时，编译器也不给出错误信息，在多数情况下将该标号置为0，若该标号为B 或BL 指令引用，则将B 或BL 指令置为NOP 操作。

使用示例：AREA Init，CODE，READONLY

**EXTERN Main** ；通知编译器当前文件要引用标号**Main**，但**Main** 在其他源文件中定义 ……

END

**10、GET（或INCLUDE）**

语法格式：GET 文件名GET 伪指令用于将一个源文件包含到当前的源文件中，并将被包含的源文件在当前位置进行汇编处理。可以使用INCLUDE 代替GET。

汇编程序中常用的方法是在某源文件中定义一些宏指令，用EQU 定义常量的符号名称，用MAP 和



PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

FIELD 定义结构化的数据类型，然后用GET 伪指令将这个源文件包含到其他的源文件中。使用方法与C 语言中的“include”相似。

GET 伪指令只能用于包含源文件，包含目标文件需要使用INCBIN 伪指令

使用示例：AREA Init，CODE，READONLY

```
GET          a1.s           ；通知编译器当前源文件包含源文件a1.s
GE T         C：\a2.s        ；通知编译器当前源文件包含源文件C：\ a2.s
……
END
```

**11、 INCBIN**

语法格式：INCBIN 文件名INCBIN 伪指令用于将一个目标文件或数据文件包含到当前的源文件中，被包含的文件不作任何

变动的存放在当前文件中，编译器从其后开始继续处理。使用

示例：AREA Init，CODE，READONLY **INCBIN a1.dat** ；通知编译器当前源文件包含文件**a1.dat INCBIN C：\a2.txt**；通知编译器当前源文件包含文件**C：\a2.txt** ……

END

**12、RN**

语法格式：名称RN 表达式RN 伪指令用于给一个寄存器定义一个别名。采用这种方式可以方便程序员记忆该寄存器的功

能。其中，名称为给寄存器定义的别名，表达式为寄存器的编码。使用示例：

**Temp RN R0** ；将**R0** 定义一个别名**Temp**

**13、ROUT**

语法格式：{名称} ROUT ROUT 伪指令用于给一个局部变量定义作用范围。在程序中未使用该伪指令时，局部变量的作

用范围为所在的AREA，而使用ROUT 后，局部变量的作为范围为当前ROUT 和下一个ROUT 之间。

## 4.2 汇编语言的语句格式

ARM（Thumb ）汇编语言的语句格式为：

{标号} {指令或伪指令} {；注释} 在汇编语言程序设计中，每一条指令的助记符可以全部用大写、或全部用小写，但不用许在一条指令中大、小写混用。同时，如果一条语句太长，可将该长语句分为若干行来书写，在行的末尾用“\”表示下一行与本行为同一条语句。

### 4.2.1 在汇编语言程序中常用的符号

在汇编语言程序设计中，经常使用各种符号代替地址、变量和常量等，以增加程序的可读性。尽管符号的命名由编程者决定，但并不是任意的，必须遵循以下的约定：

— 符号区分大小写，同名的大、小写符号会被编译器认为是两个不同的符号。
— 符号在其作用范围内必须唯一。
— 自定义的符号名不能与系统的保留字相同。
— 符号名不应与指令或伪指令同名。

**1、 程序中的变量**

程序中的变量是指其值在程序的运行过程中可以改变的量。ARM



PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

（Thumb ）汇编程序所支持的变量有数字变量、逻辑变量和字符串变量。
数字变量用于在程序的运行中保存数字值，但注意数字值的大小不应超出数
字变量所能表示的
范围。逻辑变量用于在程序的运行中保存逻辑值，逻辑值只有两种取值情况：真或假。字符串变量用于在程序的运行中保存一个字符串，但注意字符串的长度不应超出字符串变量所
能表示的范围。在ARM（Thumb ）汇编语言程序设计中，可
使用GBLA、GBLL、GBLS 伪指令声明全局变量，使用
LCLA、LCLL、LCLS 伪指令声明局部变量，并可使用
SETA、SETL 和SETS 对其进行初始化。

**2、 程序中的常量**

程序中的常量是指其值在程序的运行过程中不能被改变的量。ARM
（Thumb ）汇编程序所支持的常量有数字常量、逻辑常量和字符串常量。数
字常量一般为32 位的整数，当作为无符号数时，其取值范围为$0\sim2^{32}-1$，当
作为有符号数时，
其取值范围为$-2^{31}\sim2^{31}-1$。逻辑常量只有两种取值情况：真或假。字符串常量为一个固定的字符串，一般用于程序运行时的信息提示。

**3、 程序中的变量代换**

程序中的变量可通过代换操作取得一个常量。代换操作符为“$”。如果在数字变量前面有一个代换操作符“$”，编译器会将该数字变量的值转换为十六进制的字
符串，并将该十六进制的字符串代换“$”后的数字变量。如果在逻辑变量前面有一个代换操作符“$”，编译器会将该逻辑变量代换为它的取值（真或假）。如果在字符串变量前面有一个代换操作符“$”，编译器会将该字符串变量的值代换“$”后的
字符串变量。使用示例：**LCLS**
**S1**；定义局部字符串变量**S1** 和
**S2 LCLS S2 S1 SETS** “**Test**！”
**S2 SETS** “**This is a $S1**”；
字符串变量**S2** 的值为“**This is**
**a Test**！”

## 4.2.2 汇编语言程序中的表达式和运算符

在汇编语言程序设计中，也经常使用各种表达式，表达式一般由变量、常量、运算符和括号构
成。常用的表达式有数字表达式、逻辑表达式和字符串表达式，其运算次序遵循如下的优先级：
— 优先级相同的双目运算符的运算顺序为从左到右。
— 相邻的单目运算符的运算顺序为从右到左，且单目运算符的优先级高于其他运算符。
— 括号运算符的优先级最高。

**1、 数字表达式及运算符**

数字表达式一般由数字常量、数字变量、数字运算符和括号构成。与数字表达式相关的运算符如下：
— “＋”、“－”、“×”、“/” 及“MOD”算术运算符以上的算术运算符分别代表加、减、乘、除和取余数运算。例如，以X 和Y 表示两个数字表达
式，则：X＋Y 表示X 与Y 的和。X－Y 表示X 与Y 的差。X×Y 表示X 与Y 的乘积。X/Y 表示X 除以Y 的商。X：MOD：Y 表示X 除以Y 的余数。
— “ROL”、“ROR”、“SHL”及“SHR”移位运算符
以X 和Y 表示两个数字表达式，以上的移位运算符代表的运算如下： X：ROL：Y 表示将X 循环左移Y 位。 X：ROR：Y 表示将X 循环右移Y 位。 X：SHL：Y 表示将X 左移Y 位。 X：SHR：Y 表



PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

示将X 右移Y 位。

— “AND”、“OR”、“NOT”及“EOR”按位逻辑运算符

以X 和Y 表示两个数字表达式，以上的按位逻辑运算符代表的运算如下： X：AND：Y 表示将X 和 Y 按位作逻辑与的操作。 X：OR：Y 表示将X 和Y 按位作逻辑或的操作。：NOT：Y 表示将Y 按位作逻辑非的操作。 X：EOR：Y 表示将X 和Y 按位作逻辑异或的操作。

**2、 逻辑表达式及运算符**

逻辑表达式一般由逻辑量、逻辑运算符和括号构成，其表达式的运算结果为真或假。与逻辑表达式相关的运算符如下：

— “=”、“>”、“<”、“>=”、“<= ”、“/=”、“ <>” 运算符

以X 和Y 表示两个逻辑表达式，以上的运算符代表的运算如下：X = Y 表示X 等于Y。X > Y 表示X 大于Y。X < Y 表示X 小于Y。X >= Y 表示X 大于等于Y。X <= Y 表示X 小于等于Y。X /= Y 表示X 不等于Y。X <> Y 表示X 不等于Y。

— “LAND”、“LOR”、“LNOT”及“LEOR”运算符

以X 和Y 表示两个逻辑表达式，以上的逻辑运算符代表的运算如下： X：LAND：Y 表示将X 和Y 作逻辑与的操作。 X：LOR：Y 表示将X 和Y 作逻辑或的操作。：LNOT：Y 表示将Y 作逻辑非的操作。 X：LEOR：Y 表示将X 和Y 作逻辑异或的操作。

**3、 字符串表达式及运算符**

字符串表达式一般由字符串常量、字符串变量、运算符和括号构成。编译器所支持的字符串最大长度为512 字节。常用的与字符串表达式相关的运算符如下：

— LEN 运算符

LEN 运算符返回字符串的长度（字符数），以X 表示字符串表达式，其语法格式如下：：LEN：X

— CHR 运算符

CHR 运算符将0～255 之间的整数转换为一个字符，以M 表示某一个整数，其语法格式如下：：CHR：M

— STR 运算符

STR 运算符将将一个数字表达式或逻辑表达式转换为一个字符串。对于数字表达式，STR 运算符将其转换为一个以十六进制组成的字符串；对于逻辑表达式，STR 运算符将其转换为字符串T 或F，其语法格式如下：

：STR：X 其中，X 为一个数字表达式或逻辑表达式。

— LEFT 运算符

LEFT 运算符返回某个字符串左端的一个子串，其语法格式如下： X：LEFT：Y 其中：X 为源字符串，Y 为一个整数，表示要返回的字符个数。

— RIGHT 运算符

与LEFT 运算符相对应，RIGHT 运算符返回某个字符串右端的一个子串，其语法格式如下：X：RIGHT：Y 其中：X 为源字符串，Y 为一个整数，表示要返回的字符个数。

— CC 运算符CC 运算符用于将两个字符串连接成一个字符串，其语法格式如下：

X：CC：Y 其中：X 为源字符串1，Y 为源字符串2，CC 运算符将Y 连接到X 的后面。

**4、 与寄存器和程序计数器（PC）相关的表达式及运算符**

常用的与寄存器和程序计数器（PC）相关的表达式及运算符如下：

— BASE 运算符

BASE 运算符返回基于寄存器的表达式中寄存器的编号，其语法格式如下：：BASE：X 其中，X 为与寄存器相关的表达式。

— INDEX 运算符

INDEX 运算符返回基于寄存器的表达式中相对于其基址寄存器的偏移量，其语法格式如下：：INDEX：



PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

X 其中，X 为与寄存器相关的表达式。

**5、 其他常用运算符**

— ? 运算符

? 运算符返回某代码行所生成的可执行代码的长度，例如：

?X

返回定义符号X 的代码行所生成的可执行代码的字节数。

— DEF 运算符

DEF 运算符判断是否定义某个符号，例如：：DEF：X 如果符号X 已经定义，则结果为真，否则为假。

# 4.3 汇编语言的程序结构

## 4.3.1 汇编语言的程序结构

在ARM（Thumb ）汇编语言程序中，以程序段为单位组织代码。段是相对独立的指令或数据序列，具有特定的名称。段可以分为代码段和数据段，代码段的内容为执行代码，数据段存放代码运行时需要用到的数据。一个汇编程序至少应该有一个代码段，当程序较长时，可以分割为多个代码段和数据段，多个段在程序编译链接时最终形成一个可执行的映象文件。

可执行映象文件通常由以下几部分构成：

— 一个或多个代码段，代码段的属性为只读。

— 零个或多个包含初始化数据的数据段，数据段的属性为可读写。

— 零个或多个不包含初始化数据的数据段，数据段的属性为可读写。链接器根据系统默认或用户设定的规则，将各个段安排在存储器中的相应位置。因此源程序中段之间的相对位置与可执行的映象文件中段的相对位置一般不会相同。以下是一个汇编语言源程序的基本结构：AREA Init，CODE，READONLY ENTRY Start LDR R0，=0x3FF5000 LDR R1，0xFF STR R1，[R0] LDR R0，=0x3FF5008 LDR R1，0x01

```
STR R1，[R0] -------
END
```

在汇编语言程序中，用AREA 伪指令定义一个段，并说明所定义段的相关属性，本例定义一个名为Init 的代码段，属性为只读。ENTRY 伪指令标识程序的入口点，接下来为指令序列，程序的末尾为END 伪指令，该伪指令告诉编译器源文件的结束，每一个汇编程序段都必须有一条END 伪指令，指示代码段的结束。

## 4.3.2 汇编语言的子程序调用

在ARM 汇编语言程序中，子程序的调用一般是通过BL 指令来实现的。在程序中，使用指令：

BL 子程序名

即可完成子程序的调用。

该指令在执行时完成如下操作：将子程序的返回地址存放在连接寄存器LR 中，同时将程序计数器PC 指向子程序的入口点，当子程序执行完毕需要返回调用处时，只需要将存放在LR 中的返回地址重新拷贝给程序计数器PC 即可。在调用子程序的同时，也可以完成参数的传递和从子程序返回运算的结果，通常可以使用寄存器R0～R3 完成。

以下是使用BL 指令调用子程序的汇编语言源程序的基本结构：

```
AREA Init，CODE，READONLY
ENTRY
Start
LDR R0，=0x3FF5000
LDR R1，0xFF
```



PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

```
STR R1，[R0]
LDR R0，=0x3FF5008 LDR R1，0x01 STR R1，[R0] BL PRINT_TEXT
-------
PRINT_TEXT
-------
MOV PC，BL
-------

END
```

## 4.3.3 汇编语言程序示例

以下是一个基于S3C4510B 的串行通讯程序，关于S3C4510B 的串行通讯的工作原理，可以参考第六章的相关内容，在此仅向读者说明一个完整汇编语言程序的基本结构：

```
;*************************************************************************** ;
Institute of Automation,Chinese Academy of Sciences ;Description: This example shows the UART
communication! ;Author:
JuGuang,Lee ;Date: ;*****************************************************************
*********** UARTLCON0 EQU 0x3FFD000 UARTCONT0 EQU 0x3FFD004 UARTSTAT0 EQU 0x3FFD008 UTXBUF0
EQU 0x3FFD00C UARTBRD0 EQU 0x3FFD014
    AREA Init,CODE,READONLY
    ENTRY ;**********************************************
**** ;LED
Display ;************************************************
**
    LDR R1,=0x3FF5000
    LDR R0,=&ff
    STR R0,[R1]
    LDR R1,=0x3FF5008
    LDR R0,=&ff
    STR R0,[R1]

;************************************************** ;UART
0 line control
register ;*********************************************
***
    LDR R1,=UARTLCON0
    LDR R0,=0x03
    STR R0,[R1]

;*************************************************** ;UART
0 control
regiser ;************************************************
**
    LDR R1,=UARTCONT0
    LDR R0,=0x9
    STR R0,[R1]

;*************************************************** ;UART
0 baud rate divisor regiser ;Baudrate=19200，对应于50MHz的系
统工作频率
```



PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

```
;**************************************************** LDR R1,=UARTBRD0 LDR R0,=0x500      STR
    R0,[R1]
;****************************************************;Print
the
messages! ;*********************************************
**** LOOP
    LDR R0,=Line1
    BL PrintLine
    LDR R0,=Line2
    BL PrintLine
    LDR R0,=Line3
    BL PrintLine
    LDR R0,=Line4
    BL PrintLine
    LDR R1,=0x7FFFFF

LOOP1 SUBS R1,R1,#1 BNE LOOP1 B LOOP
;****************************************************;Print
line ;****************************************************
PrintLine
    MOV R4,LR
    MOV R5,R0

Line LDRB R1,[R5],#1 AND R0,R1,#&FF TST R0,#&FF MOVEQ PC,R4 BL PutByte B Line

PutByte LDR R3,=UARTSTAT0 LDR R2,[R3] TST R2,#&40 BEQ PutByte LDR R3,=UTXBUF0 STR R0,[R3] MOV
    PC,LR
Line1 DCB
     &A,&D,"**************************************************************",0 Line2 DCB
&A,&D,"Chinese Academy of Sciences,Institute of Automation,Complex System Lab.",0 Line3 DCB
&A,&D,"  ARM Development Board Based on Samsung ARM S3C4510B.",0 Line4 DCB
     &A,&D,&A,&D,&A,&D,&A,&D,&A,&D,&A,&D,&A,&D,&A,&D,&A,&D,&A,&D,&A,&D,&A,&D,&A,&D,&
A,&D,&A,&D,0 END
```

## 4.3.4 汇编语言与C/C++的混合编程

在应用系统的程序设计中，若所有的编程任务均用汇编语言来完成，其工作量是可想而知的，同时，不利于系统升级或应用软件移植，事实上，ARM 体系结构支持C/C＋以及与汇编语言的混合编程，在一个完整的程序设计的中，除了初始化部分用汇编语言完成以外，其主要的编程任务一般都用C/C++ 完成。

汇编语言与C/C++的混合编程通常有以下几种方式：

－在C/C＋＋代码中嵌入汇编指令。

－在汇编程序和C/C＋＋的程序之间进行变量的互访。

－汇编程序、C/C＋＋程序间的相互调用。

在以上的几种混合编程技术中，必须遵守一定的调用规则，如物理寄存器的使用、参数的传递等，这对于初学者来说，无疑显得过于烦琐。在实际的编程应用中，使用较多的方式是：程序的初始化部分用汇编语言完成，然后用C/C＋＋完成主要的编程任务，程序在执行时首先完成初始化过程，然后跳转到C/C＋＋程序代码中，汇编程序和C/C＋＋程序之间一般没有参数的传递，也没有频繁的相互调用，因此，整个程序的结构显得相对简单，容易理解。以下是一个这种结构程序的基本示例，该程序基于第五、六章所描述的硬件平台：;************************************************************************ ;



PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

```
Institute of Automation, Chinese Academy of Sciences ;File Name: Init.s ;Description: ;Author:
JuGuang,Lee ;Date: ;***********************************************************************
***
    IMPORT Main ;通知编译器该标号为一个外部标号
    AREA Init,CODE,READONLY ; 定义一个代码段
    ENTRY ; 定义程序的入口点
    LDR R0,=0x3FF0000 ; 初始化系统配置寄存器，具体内容可参考第五、六章
    LDR R1,=0xE7FFFF80
    STR R1,[R0]
    LDR SP,=0x3FE1000 ; 初始化用户堆栈，具体内容可参考第五、六章
    BL Main ; 跳转到Main（）函数处的C/C++代码执行
    END ; 标识汇编程序的结束
```

以上的程序段完成一些简单的初始化，然后跳转到Main （）函数所标识的C/C＋＋代码处执行主要的任务，此处的Main 仅为一个标号，也可使用其他名称，与C 语言程序中的main （）函数没有关系。

```
/*******************************************************************************
       * Institute of Automation, Chinese Academy of Sciences

       * File Name: main.c

       * Description: P0,P1 LED flash.

       * Author: JuGuang,Lee

       * Date:
*******************************************************************************/
void Main(void)
{


      int i;
      *((volatile unsigned long *) 0x3ff5000) = 0x0000000f;

      while(1)
        {
        *((volatile unsigned long *) 0x3ff5008) = 0x00000001;
        for(i=0; i<0x7fFFF; i++);

            *((volatile unsigned long *) 0x3ff5008) = 0x00000002;
        for(i=0; i<0x7FFFF; i++);
        }

}
```

## 4.4 本章小节

本章介绍了**ARM** 程序设计的一些基本概念，以及在汇编语言程序设计中常见的伪指令、汇编语言的基本语句格式等，汇编语言程序的基本结构等，同时简单介绍了**C/C**＋＋和汇编语言的混合编程等问题，这些问题均为程序设计中的基本问题，希望读者掌握，注意本章最后的两个示例均与后面章节介绍的基于 **S3C4510B** 的硬件平台有关系，读者可以参考第五、六章的相关内容。



**PDF** 文件使用 **"pdfFactory Pro"** 试用版本创建 **www.fineprint.cn**