“十三五”普通高等教育规划教材

嵌入式系统原理与应用

第 2 版

魏权利　李丽萍　林粤伟　编著

机 械 工 业 出 版 社

本书分为13章，内容包括：嵌入式系统概述；ARM微处理器体系结构；ARM微处理器指令系统；微处理器ARM程序设计；微处理器S3C2410A体系结构；嵌入式系统应用产品开发平台；嵌入式存储器系统及扩展接口电路；通用I/O端口和中断系统；微处理器S3C2410A的定时器/计数器；A-D转换、LCD触摸屏与液晶显示器；嵌入式系统I/O总线接口；嵌入式应用程序设计举例；ARM9实验项目及内容。

本书翔实地介绍了ARM系统在启动过程中涉及的硬件原理以及通过软件进行配置的程序。全书内容简练、概念清晰、逻辑性强、深入浅出，具有很强的专业性、技术性与实用性。

本书可以作为高等院校电子信息工程、自动化、电气工程等专业的教材，也可以作为广大嵌入式开发工程技术人员的参考用书。

本书配套授课电子课件，需要的教师可登录www.cmpedu.com免费注册，审核通过后下载，或联系编辑索取（QQ：2850823885，电话：010-88379739）。

图书在版编目(CIP)数据

嵌入式系统原理与应用/魏权利，李丽萍，林粤伟 编著. —2版. —北京：机械工业出版社，2018.5
“十三五”普通高等教育规划教材
ISBN 978-7-111-60518-8

Ⅰ. ①嵌… Ⅱ. ①魏… ②李… ③林… Ⅲ. ①微型计算机—系统设计—高等学校—教材 Ⅳ. ①TP360.21

中国版本图书馆CIP数据核字(2018)第161821号

机械工业出版社(北京市百万庄大街22号 邮政编码 100037)
责任编辑：郝建伟 责任校对：张艳霞
责任印制：

印刷(　　装订)

2018年8月第2版·第1次印刷
184mm×260mm·20.25印张·496千字
0001-3000册
标准书号：ISBN 978-7-111-60518-8
定价：59.00元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

电话服务
服务咨询热线：(010)88379833
读者购书热线：(010)88379649

网络服务
机 工 官 网：www.cmpbook.com
机 工 官 博：weibo.com/cmp1952
教育服务网：www.cmpedu.com
金 书 网：www.golden-book.com

前　言

目前，随着计算机网络应用范围的不断扩展，中国“互联网+”时代的到来，中国制造2025战略的倡导，无线网络技术的广泛应用，几乎所有的机械制造设备、通信设备、控制设备等都将使用32位的ARM处理器嵌入其中作为它们的控制中心。32位ARM处理器的性能和CPU的处理速度的发展日新月异，而低性能、低速度的嵌入式单片机已无法承担这些外围接口繁多、响应速度极快的处理任务。而且随着开发平台和开发软件的不断完善，开发的难度将会下降，在中国制造2025战略思想的指导下，将会有越来越多的科技人员投入到嵌入式系统产品的研发中，使我国科技人员嵌入式应用系统研发的水平和国际地位不断提高，从而研制出具有世界水准或超越世界水平的信息产品。

无论是进行嵌入式应用系统的裸机开发还是基于操作系统的开发，目前都很难找到一本能全面、系统地介绍嵌入式系统启动时或启动引导Bootloader所涉及的所有硬件电路工作原理以及程序设计。本书的撰写就是为了弥补这一缺憾，并且可在本书搭建的实验平台上实验，这将大大提高广大学生的实际操作能力和学习兴趣。全书共分13章，各章的内容介绍如下。

第1章介绍了嵌入式系统的概念和组成，嵌入式微处理器的结构与类型，精简指令集计算机RISC的特点和流水线技术，最后叙述了嵌入式应用系统的开发流程。

第2章介绍了ARM处理器的结构、特点和应用选型，ARM的总线系统与接口，重点讲述了ARM9体系结构的存储器组织、ARM9微处理器的工作状态与运行模式、ARM9体系结构的寄存器组织、ARM9微处理器的异常。

第3章介绍ARM9微处理器的指令格式与特点、寻址方式，分类讲述ARM9指令的功能，并给出了大量的应用示例。

第4章主要讲述ARM伪指令、ARM汇编语言程序设计、ARM汇编语言与C语言的混合编程以及子程序或函数之间的相互调用。

第5章主要讲述微处理器S3C2410A的体系结构、内部组成、存储器控制寄存器的特性与空间分布、复位电路、电源电路、时钟电路与电源管理等。

第6章主要讲述了ARM9的软、硬件开发平台以及在实际应用中的配置。

第7章介绍了嵌入式存储器系统结构组成、MMU的功能与工作原理，重点讲述了存储器控制寄存器的功能及其实际应用中的设置编程、使用8位/16位/32位数据线存储器芯片扩展设计8位/16位/32位ARM总线系统的电路。

第8章简述了S3C2410A的I/O端口的功能，特殊功能寄存器的作用与配置。详细讲述了ARM9的中断系统以及实际应用的编程过程。

第9章讲述了S3C2410A的定时器/计数器的工作原理，重点介绍了看门狗定时器、RTC实时时钟，Timer 0～Timer 4定时/计数器的工作原理、功能寄存器以及它们的设置与应用编程。

第10章详细地介绍了A-D转换器、触摸屏、LCD的工作原理，功能寄存器及其编程。特别阐述了TFT-LCD的应用程序设计。

第11章讲述了S3C2410A的UART、I^2C、SPI总线的工作原理和功能寄存器，并结合实际

使用的总线接口芯片进行了程序设计。

第 12 章为嵌入式应用程序设计举例，详细地介绍了 S3C2410A 启动程序的设计、数字温度传感器 DS18B20 的编程原理等，在此基础上完成了实时温度监测系统的设计。

第 13 章列出了实验项目与实验内容，通过实验可加深对课程内容的理解。

本书计划需要 48~64 学时，教学过程中可以根据实际情况进行适当的调整。

本书主要由魏权利教授编写，并对全书的内容进行了审定。第 9 章由林粤伟博士编写。高级实验师李丽萍参与了本书的编写工作。第 12 章的实际应用程序在嵌入式实验开发平台上进行了调试，完成了整个程序的设计功能，该部分工作由乔方昭完成。

本书是作者从事 30 多年嵌入式系统应用研发和教学的工作总结和经验积累，本书的修订也是对作者的鼓舞。机械工业出版社为本书的修订做了大量细致而周到的工作，在此表示由衷的感谢。

由于作者的学识、经验和水平有限，书中难免有错误和疏漏之处，欢迎广大读者批评指正。

编　者

目　录

第 1 章　嵌入式系统概述

嵌入式系统是后 PC 时代被广泛应用的计算机系统。在人们的日常生活、学习和工作中所接触的仪器或设备中，都能嵌入具有强大控制能力和计算能力的嵌入式计算机系统。嵌入式系统不仅广泛应用于成熟领域，如工业控制、家用电器、通信设备、网络设备、医疗器械和军事装置等，而且随着嵌入式系统的不断发展还衍生出了许多新的应用，如 PDA、智能手机、MP4、运动控制器和无线路由器等。可以预见，嵌入式系统随着技术的不断完善、使用范围的逐步扩展、开发环境更加方便易用，必将会有大量的技术人员投入其中，使我国硬件开发人员的队伍不断壮大，促进国民经济的快速发展。

1.1　嵌入式系统的概念与组成

本节首先介绍嵌入式系统的定义与“三要素”，其次介绍嵌入式系统的应用过程和发展趋势，最后介绍嵌入式系统的组成。

1.1.1　嵌入式系统的定义

嵌入式系统的定义有许多，但它们的真正含义基本相同，以下是具体的定义内容。

根据国际电气和电子工程师协会（Institute of Electrical and Electronics Engineers，IEEE）的定义，嵌入式系统是“控制、监视或者辅助设备、机器和车间运行的装置”。

目前国内一个普遍被认同的定义是：以应用为中心，以计算机技术为基础，软件硬件可裁剪，适应应用系统对功能、可靠性、成本、体积、功耗严格要求的专用计算机系统。

也可以这样定义：嵌入式系统是一种专用的单片计算机系统，作为装置或设备的主控或监测器件，焊接在它们的印制电路板（PCB）中，完成它应具有的功能。

嵌入式系统一般由嵌入式微处理器芯片、外围硬件设备、嵌入式操作系统以及用户应用程序 4 个部分组成。

嵌入式系统的三个基本要素是指“嵌入性”“专用性”“计算机系统”。嵌入性是指它是以芯片的形式嵌入（潜伏）在 PCB 电路板中；专用性是指它是为特定的设备量身定做的软硬件系统；计算机系统是说，它虽然以芯片的形式显现，但是它具有一台计算机的软硬件功能。

目前嵌入式系统的应用无处不在，8 位单片机嵌入式系统，例如 MCS-51 系列，在低端产品中是主流，它占整个嵌入式系统的市场份额约为 70%。在中、高端产品中 ARM 使用占 70% 的份额，在移动电话、数码照相机、数字电视的机顶盒、微波炉、汽车内部的防抱死制动系统等装置或设备中都使用了 ARM 嵌入式系统。

1.1.2　嵌入式系统的应用过程和发展趋势

1. 嵌入式系统应用过程的 4 个阶段

第 1 阶段：无操作系统阶段

MCS-51 系列单片机是最早应用的嵌入式系统之一，单片机作为各类工业控制和飞机、导

弹等武器装备中的微控制器，用来执行一些单线程的程序，完成监测、伺服和设备指示等多种功能，一般没有操作系统的支持，程序设计采用汇编语言或 C51 语言。

采用汇编语言编写的程序具有效率高、占用内存少、实时性强且控制时间精准等优点。缺点是对技术人员的要求高，开发周期相对长一些。

现在使用意法半导体（ST）公司基于 ARM9 的 ARM Cortex-M3 内核产品——STM32 微处理器芯片开发的实时控制设备，大部分都是在无操作系统的情况下使用 C 语言开发的，它比在有操作系统下开发有更高的运行效率。目前开发要求具有强实时性的装备也是在“裸机”（无操作系统）情况下开发的。

第 2 阶段：简单操作系统阶段

20 世纪 80 年代，出现了大量具有高可靠性、低功耗的嵌入式 CPU。芯片上集成有 CPU、I/O 接口、串行接口及 RAM、ROM 等部件，是面向 I/O 设计的微控制器在嵌入式系统设计中的应用。一些简单的嵌入式操作系统开始出现并得到迅速发展，程序设计人员也开始基于一些简单的“操作系统”开发嵌入式应用软件，如较为常用的 μC/OS 嵌入式操作系统。此时的嵌入式操作系统虽然还比较简单，但已经初步具有了一定的兼容性和扩展性，内核精巧且效率高，大大缩短了开发周期，提高了开发效率。

第 3 阶段：实时操作系统阶段

20 世纪 90 年代，面对分布式控制、柔性制造、数字化通信和信息家电等巨大的市场需求，嵌入式系统飞速发展。随着硬件实时性要求的提高，嵌入式系统的软件规模也不断扩大，如实时操作系统（Real-Time Operating System，RTOS），从而使应用软件的开发变得更加简单。

第 4 阶段：面向 Internet 阶段

进入 21 世纪，Internet 技术与信息家电、工业控制技术等的结合日益紧密，嵌入式技术与 Internet 技术的结合正在推动着嵌入式系统的飞速发展。由于 Linux 是 UNIX 的 PC 版本，具有强大的网络功能，且为开源软件，因此嵌入式 Linux 操作系统得到了广泛的应用。微软公司也看到了嵌入式市场的广阔前景，推出 Windows CE 嵌入式操作系统，对于熟悉 Windows 环境的开发人员来讲，也可进行基于 Windows 平台的嵌入式系统开发。

2. 嵌入式系统的发展趋势

面对嵌入式技术与 Internet 技术的结合，嵌入式系统的研究和应用呈现出以下发展趋势。

1）新的微处理器层出不穷，大都朝着精简系统内核，优化关键算法，降低功耗和软硬件成本，提供更加友好的多媒体人机交互界面的方向发展。

2）Linux、Windows CE、Palm OS 等嵌入式操作系统迅速发展。嵌入式操作系统自身结构的设计体现出更加便于移植的特性，具有源代码开放、系统内核小、执行效率高、网络结构完整等特点，能够在短时间内支持更多的微处理器。

3）嵌入式系统的开发成了一项系统工程，开发厂商不仅要提供嵌入式软硬件系统本身，同时还要提供强大的硬件开发工具和软件支持包。

1.1.3 嵌入式系统的组成

嵌入式系统的组成包括嵌入式系统硬件组成和软件组成两部分。

嵌入式系统的硬件组成主要包含有嵌入式处理器、外围设备接口和执行装置（被控对象）等。

嵌入式系统的软件组成，对于裸机开发来讲主要有以下内容：嵌入式处理器芯片内部三总线频率的设置、配置存储器芯片的设置；7 种异常模式堆栈指针的设置；中断指针的传递程序；为 C 语言的运行创建环境；I/O 端口的配置与控制程序、应用程序等。以上 5 个部分也是引导启动程序（Bootloader）的主要内容。

对于基于操作系统的嵌入式软件开发，主要包括 Bootloader 的移植，操作系统内核的移植，文件系统的移植，I/O 设备驱动程序的编写以及加载，图形用户接口程序设计，应用程序的设计等。

嵌入式计算机系统是整个嵌入式系统的核心，可以分为硬件层、中间层、系统软件层和应用软件层。执行装置接收嵌入式计算机系统发出的控制命令，执行所规定的操作或任务。

嵌入式系统从整体上来讲也可以分为硬件层、中间层、系统软件层和应用软件层。

1.2 嵌入式微处理器的结构与类型

嵌入式处理器是隐藏在控制设备或装置中，完成接收现场数据，进行数据处理，并向执行装置发出控制命令的微处理器。1971 年 Intel 公司推出了 Intel4004，1974 年推出了 Intel8080，1976 年 zilog 制造了与 8080 兼容的 CPU Z-80，这类处理器（称为 CPU）所构造的是单板微型计算机系统，简称单板机，应用在控制设备中，它们都是嵌入式应用的前身。之后出现了简称单片机的单片微型计算机。例如，Intel 公司在 1976 年 9 月推出的 MCS-51 系列 8 位单片机，它内部不但集成了 CPU，还集成了存储器和 I/O 接口等计算机的元素，但这时嵌入式系统的概念还不是热点的技术名词。一直到 20 世纪 90 年代后期 32 位 ARM 微处理器的广泛使用，嵌入式系统的概念才被广大技术人员所熟知。现在人们把具有计算机基本组成元素的单片微型集成电路芯片，从 MCS-51 系列单片机开始到目前的 32 位 ARM 微处理器统称为嵌入式系统，但从技术人员的角度出发，嵌入式系统主要指的是 32 位 ARM 微处理器单片机。

嵌入式微处理器按 CPU 的处理能力可分为 8 位、16 位、32 位和 64 位。一般把处理能力在 16 位及以下的称为嵌入式微控制器（Embedded Microcontroller），32 位及以上的称为嵌入式微处理器。

嵌入式微处理器内部将 CPU、ROM、RAM 及 I/O 等部件集成到同一个芯片上，称为单芯片微控制器（Single Chip Microcontroller）。

根据用途，可以将嵌入式芯片系统分为嵌入式微控制器、嵌入式微处理器、嵌入式 DSP 处理器、嵌入式片上系统、双核或多核处理器等类型。

1.2.1 嵌入式微控制器

嵌入式微控制单元（Micro Controller Unit，MCU）又称为单片机，芯片内部集成了 ROM、RAM、总线逻辑、定时器/计数器、看门狗、I/O、串行口、脉宽调制输出（PWM）、A-D、D-A、Flash、E^2PROM 等各种必要功能和外设。嵌入式微控制器具有单片化、体积小、功耗和成本低、可靠性高等特点，约占嵌入式系统市场份额的 70%。

嵌入式微控制器的代表芯片就是 MCS-51 系列单片机，主要使用其汇编语言或 C 语言进行裸机开发。

1.2.2 嵌入式 DSP 处理器

嵌入式 DSP 处理器（Embedded Digital Signal Processor，EDSP）是专门用于信号处理方面的处理器，芯片内部采用程序和数据分开存储和传输的哈佛结构，具有专门的硬件乘法器，采用流水线操作，提供特殊的 DSP 指令，可用来快速地实现各种数字信号处理算法，使其处理速度比其他性能优异的 CPU 还快 10 倍以上。

从 20 世纪 80 年代到现在，缩小 DSP 芯片尺寸始终是 DSP 技术的发展方向。DSP 处理器已发展到第 5 代产品，多数基于精简指令集计算机（Reduced Instruction Set Computer，RISC）结构，并将几个 DSP 芯核、MPU 芯核、专用处理单元、外围电路单元和存储单元集成在一个芯片上，成为 DSP 系统级集成电路，系统集成度极高。

DSP 运算速度的提高主要依靠新工艺改进芯片结构。目前一般的 DSP 运算速度为 100MIPS（即每秒钟可运算 1 亿条指令）。TI 的 TM320C6X 芯片由于采用超长指令字（Very Long Instruction Word，VLIW）结构设计，其处理速度已高达 2000MIPS。按照发展趋势，DSP 的运算速度完全可能再提高 100 倍（达到 1600GIPS）。

目前 DSP 芯片在机械电子的控制方面运用广泛，如作为变频器、PLC 的控制核心。它的开发基本也是在裸机中进行的，主要使用 C 语言进行裸机程序设计。

1.2.3 嵌入式微处理器

嵌入式微处理器（Micro Processor Unit，MPU）由通用计算机的 CPU 发展而来，嵌入式微处理器只保留和嵌入式应用紧密相关的功能硬件，去除其他冗余功能部分，以最低的功耗和资源实现嵌入式应用的特殊要求。通常嵌入式微处理器把 CPU、ROM、RAM 及 I/O 等做到同一个芯片上。32 位微处理器采用 32 位的地址总线和数据总线，其地址空间达到了 $2^{32}=4$GB。目前主流的 32 位嵌入式微处理器系列主要有 ARM 系列、MIPS 系列、PowerPC 系列，以下进行简要介绍。属于这些系列的嵌入式微处理器产品很多，有千种以上。

1. 嵌入式 ARM 系列

ARM（Advanced RISC Machine）公司的 ARM 微处理器体系结构目前被公认为是嵌入式应用领域领先的 32 位嵌入式 RISC 微处理器结构。ARM 体系结构目前发展并定义了 7 种不同的版本。从版本 v1 到版本 v7，ARM 体系的指令集功能不断扩大。ARM 处理器系列中的各种处理器，虽然在实现技术、应用场合和性能方面都不相同，但只要支持相同的 ARM 体系版本，基于它们的应用软件是兼容的。

目前，大量的移动电话、游戏机、平板电脑和机顶盒等都已采用了 ARM 处理器，许多一流的芯片厂商都是 ARM 的授权用户，如 Intel、Samsung、TI、Freescale、ST 等公司。

2. 嵌入式 MIPS 系列

美国斯坦福大学的 Hennessy 教授领导的研究小组研制的无互锁流水级微处理器（Microprocessor without Interlocked Piped Stages，MIPS）是世界上非常流行的一种 RISC 处理器，其机制是尽量利用软件办法避免流水线中的数据相关问题。

从 20 世纪 80 年代初期 MIPS 处理器发明至今的 30 多年里，MIPS 处理器以其高性能的处理能力被广泛应用于路由器、调制解调设备、电视、游戏、打印机、DVD 播放器等广泛领域。

3. 嵌入式 PowerPC 系列

PowerPC 是 Freescale（原 Motorola）公司的产品。PowerPC 的 RISC 处理器采用了超标量处

理器设计和调整内存缓冲器，修改了指令处理设计，完成一个操作所需的指令数比复杂指令集计算机（Complex Reduced Instruction Set Computer，CISC）结构的处理器要多，但完成操作的总时间却减少了。

PowerPC 内核采用独特分支处理单元可以让指令预取效率大大提高，即使指令流水线上出现跳转指令，也不会影响到其运算单元的运算效率。PowerPC RISC 处理器设计了多级内存高速缓冲区，以便让那些正在访问（或可能会被访问）的数据和指令总是存储在调整内存中。这种内存分层和内存管理设计，使指令系统的内存访问性能非常接近调整内存，但其成本却与低速内存相近。

1.2.4 嵌入式片上系统

嵌入式片上系统（System On Chip，SOC）最大的特点是成功实现了软硬件无缝结合，直接在处理器片内嵌入操作系统的代码模块，而且具有极高的综合性，在一个芯片内部运用超高速硬件描述语言，如 VHDL 等，即可实现一个复杂的系统。与传统的系统设计不同，用户不需要绘制庞大复杂的电路板来一点点地连接焊制，只需要使用精确的语言，综合时序设计直接在器件库中调用各种通用处理器的标准，然后在仿真之后就可以直接交付芯片厂商进行生产，设计生产效率高。

在 SOC 中，绝大部分系统构件都是在系统内部，系统简洁，系统的体积和功耗小，可靠性高。SOC 芯片已在声音、图像、影视、网络及系统逻辑等领域中广泛应用。

1.3 计算机组成、体系结构与嵌入式处理器

计算机组成主要是指计算机的硬件组成部件以及实现这些硬件功能所使用的材料、实现的理论与技术方法，以及各组成部分的逻辑关系，它决定着计算机的性能和功能。

目前正在使用的现有计算机是由电子来传递和处理信息的。电子在导线中传播的速度虽然比人们看到的任何运载工具运动的速度都快，但是，从发展高速率计算机来说，采用电子做传输信息载体还不能满足更快的要求，提高计算机运算速度也明显表现出它的能力是有限的。而光子计算机以光子作为传递信息的载体，光互连代替导线互连，以光硬件代替电子硬件，以光运算代替电子运算，利用激光来传送信号，并由光导纤维与各种光学元件等构成集成光路，从而进行数据运算、传输和存储。在光子计算机中，不同波长、频率、偏振态及相位的光代表不同的数据，这远胜于电子计算机中通过电子“0”“1”状态变化进行的二进制运算，可以对复杂度高、计算量大的任务实现快速的并行处理。光子计算机的主板中不存在电磁干扰，使信道的传输速率更快，将使运算速度远远高于现有的计算机速度。目前正在研究中的计算机还有量子计算机、超导计算机和多值计算机等。

计算机体系结构主要指计算机的系统化设计和构造，不同的计算机体系结构适用于不同的需求或应用。从传统意义的指令界面上来看，现代计算机的体系结构基本划分成两大类：复杂指令集计算机系统 CISC 体系（如 X86 芯片）和简化指令集计算机系统 RISC 体系（如 ARM 芯片）。

因此可以说嵌入式处理器是一种结合了 X86 个人计算机的 PC 体系结构，实时控制系统的要求和简化指令集之后产生的，满足实时控制系统应用的计算机体系结构。以下将分别介绍冯·诺依曼结构、哈佛结构、精简指令集计算机（RISC）和流水线计算机等内容。

1.3.1 冯·诺依曼结构与哈佛结构

1. 冯·诺依曼（Von Neumann）结构

冯·诺依曼结构的计算机由 CPU 和存储器构成，其程序和数据共用一个存储空间，程序指令存储地址和数据存储地址指向同一个存储器的不同物理位置；采用单一的地址及数据总线，程序指令和数据的宽度相同。程序计数器（PC）是 CPU 内部指示指令和数据的存储位置的寄存器。

目前使用冯·诺依曼结构的 CPU 和微控制器的品种有很多，例如 Intel 公司的 X86 系列及其他 CPU、ARM 公司的 ARM7、MIPS 公司的 MIPS 处理器等。

2. 哈佛（Harvard）结构

哈佛结构的主要特点是将程序和数据存储在不同的存储空间中，即程序存储器和数据存储器是两个相互独立的存储器，每个存储器独立编址、独立访问。系统中具有程序的数据总线与地址总线，数据的数据总线与地址总线。这种分离的程序总线和数据总线可允许在一个机器周期内同时获取指令字和操作数，从而提高执行速度，提高数据的吞吐率。又由于程序和数据存储器在两个分开的物理空间中，因此取指和执行能完全重叠，具有较高的执行效率。

目前使用哈佛结构的 CPU 和微控制器品种有很多，除 DSP 处理器外，还有摩托罗拉公司的 MC68 系列、ATMEL 公司的 AVR 系列和 ARM 公司的 ARM9、ARM10 和 ARM11 等。

1.3.2 精简指令集计算机（RISC）

1. 2/8 规律

早期的计算机采用复杂指令集计算机（CISC）体系。采用 CISC 体系结构的计算机各种指令的使用频率相差悬殊，统计表明，大概有 20%比较简单的指令被反复使用，使用量约占整个程序的 80%；而有 80%左右的指令则很少使用，其使用量约占整个程序的 20%，即指令的 2/8 规律。

2. RISC

精简指令集计算机（RISC）体系结构是 20 世纪 80 年代提出来的。目前 Intel 等公司都在研究和发展 RISC 技术，RISC 已经成为计算机发展不可逆转的趋势。

3. RISC 的特点

RISC 是在 CISC 的基础上产生并发展起来的，RISC 的着眼点不是简单地放在简化指令系统上，而是通过简化指令系统使计算机的结构更加简单合理，从而提高运算效率。

- 在 RISC 中，优先选取使用频率最高的、很有用但不复杂的指令，避免使用复杂指令。
- 固定指令长度，减少指令格式和寻址方式种类。
- 指令之间各字段的划分比较一致，各字段的功能也比较规整。
- 采用 Load/Store 指令访问存储器，其余指令的操作都在寄存器之间进行。
- 增加 CPU 中通用寄存器数量，算术逻辑运算指令的操作数都在通用寄存器中存取。
- 大部分指令控制在一个或小于一个机器周期内完成。

尽管 RISC 架构与 CISC 架构相比具有较多的优点，但 RISC 架构也不可以取代 CISC 架构。事实上，RISC 和 CISC 各有优势。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 两者的优势，成为未来的 CPU 发展方向之一。在 PC 和服务器领域，CISC 体系结构是市场的主流。

在嵌入式系统领域，由于注重的是实时性效果，要求在系统主频一定的情况下有较高的信息处理能力，精简指令集计算机（RISC）系统可以使所有的机器指令具有相同的长度，易于进行流水线处理，大大提高了计算机执行指令的速度。因此 RISC 结构的微处理器在该领域将占有重要的位置。

1.3.3 流水线计算机

1. 流水线的基本概念

精简指令集计算机（RISC）为微处理器的指令流水线执行提供了先决条件。流水线技术应用于计算机体系结构的各个方面，流水线技术的基本思想是将一个重复的时序分解成若干个子过程，而每一个子过程都可有效地在其专用功能段上与其他子过程同时执行。

流水线结构的类型众多。指令流水线就是将一条指令分解成一连串执行的子过程，例如，把指令的执行过程细分为取指令、指令译码、取操作数和执行 4 个子过程，每个过程的执行时间相同。

在 CPU 中把一条指令的串行执行子过程变为若干条指令的子过程在 CPU 中重叠执行。如果能做到每条指令均分解为 m 个子过程，且每个子过程的执行时间都一样，则利用此条流水线可将一条指令的执行时间由原来的 T 缩短为 T/m。指令流水线处理的时空图如图 1-1 所示，其中的 1、2、3、4、5 表示要处理的 5 条指令。从图 1-1 中可见采用流水方式可同时执行多条指令。

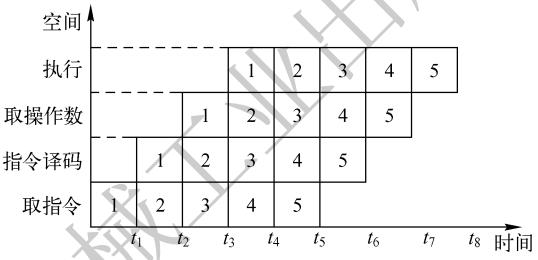

图 1-1 五级流水线指令执行示意图

2. 流水线处理机的主要指标

（1）吞吐率

在单位时间内，流水线处理机流出的结果数称为吞吐率。对指令而言就是单位时间内执行的指令数。如果流水线的子过程所用时间不一样长，则吞吐率 P 应为最长子过程的倒数，即

$$P = 1/\max\{\Delta t_0, \Delta t_1, \cdots, \Delta t_m\}$$

（2）建立时间

流水线开始工作，须经过一定时间才能达到最大吞吐率，这就是建立时间。若 m 个子过程所用时间一样，均为 t_0，则建立时间 $T_0 = m\Delta t_0$。

1.3.4 嵌入式微处理器的信息存储方式

1. 大端和小端存储方式

大多数计算机使用 8 位数据块作为最小的可寻址存储器单位，称为 1 字节。存储器的每一个字节都用一个唯一的地址（address）来标识。所有可能地址的集合称为存储器空间。

对于软件而言，它将存储器看作一个大的字节数组，称为虚拟存储器。在实际应用中，虚拟存储器可以划分成不同单元，用来存放程序、指令和数据等信息。例如，在 C 语言中定义的整型数据变量：int x，表示变量 x 在内存中占有 4 字节。

在微处理器中，使用一个字长（word）表明整数和指令数据的大小。字长决定了微处理器的寻址能力，即虚拟地址空间的大小。对于一个字长为 n 位的微处理器，它的虚拟地址范围为 $0 \sim 2^n - 1$。例如一个 32 位的微处理器，可访问的虚拟地址空间为 2^{32}，即 4 GB。

对于一个多字节类型的数据，在存储器中有两种存放方式：小端方式与大端方式。

小端方式：是指低字节数据存放在内存低地址位置处，高字节数据存放在内存高地址位置处，称为小端字节顺序存储法或简称小端方式。

大端方式：是指高字节数据存放在低地址位置处，低字节数据存放在高地址位置处，称为大端字节顺序存储法或简称大端方式。

例如，假设在一个 32 位字长的微处理器上定义一个 int 类型的常量 a，其内存地址位于 0x1000 处，其值用十六进制表示为 0x12345678。如果按小端方式存储，则其最低字节数据 0x78 存放在内存低地址 0x1000 处，最高字节数据 0x12 存放在内存高地址 0x1003 处，如图 1-2a所示。如果按大端方式存储，则其最高字节数据 0x12 存放在内存的低地址 0x1000 处，而最低字节数据 0x78 存放在内存的高地址 0x1003 处，如图 1-2b 所示。

地址	0x1000	0x1001	0x1002	0x1003
数据（十六进制）	0x78	0x56	0x34	0x12
数据（二进制）	01111000	01010110	00110100	00010010

a)

地址	0x1000	0x1001	0x1002	0x1003
数据（十六进制）	0x12	0x34	0x56	0x78
数据（二进制）	00010010	00110100	01010110	01111000

b)

图 1-2　数据存储的小端与大端存储方式

a）小端方式存储数据格式　b）大端方式存储数据格式

采用大端存储方式还是小端存储方式，各处理器厂商的立场和习惯不同，并不存在技术原因。Intel 公司 X86 系列微处理器都采用小端存储法，而 IBM、Motorola 和 Sun Microsystems 公司的大多数微处理器采用大端存储法。此外，还有一些微处理器，如 ARM、MIPS 和 Motorola 的 PowerPC 等，可以通过芯片上电启动时确定的字节存储顺序规则，来选择存储模式。

另外，是大端存储方式还是小端存储方式，不但可以由计算机系统的硬件决定，也可以由工具语言的编译器来决定。

对于大多数程序员而言，机器的字节存储顺序是完全不可见的，无论哪一种存储方式的微处理器编译出的程序都会得到相同的结果。不过，当不同存储方式的微处理器之间通过网络传送二进制数据时，在有些情况下，字节顺序会成为问题，会出现所谓的“UNIX”问题。字符“UNIX”在 16 位字长的微处理器上被表示为两个字节，当被传送到不同存储模式的机器上时，则会变为“NUXI”。

为了避免这类问题，网络应用程序代码编写必须遵循已建立好的网络信道传输数据的字节顺序的协议，以保证发送方微处理器先在其内部将发送的数据转换成网络标准，而接收方微处

理器再将网络标准转换为它的内部表示。也就是说，不管网络主机本身采用何种存储方式，还是网络工具软件采用何种存储方式，在网络信道上必须遵循统一的规则或称协议。

2. 可移植性问题

当在不同存储顺序的微处理器间进行程序移植时，要特别注意存储模式的影响。把从软件得到的二进制数据写成一般的数据格式往往会涉及存储顺序的问题。

在多台不同存储顺序的主机之间共享信息可以有两种方式：一种是以单一存储方式共享数据，另一种是允许主机以不同的存储方式共享数据。使用单一存储顺序只要解释一种格式，解码简单。使用多种存储方式不需要对数据的原顺序进行转化，使得编码容易。当编码器和解码器采用同一种存储方式时，因为不需要变换字节顺序，能提高通信效率。

3. 网络信道中的字节顺序问题

在网络通信中，Internet 协议（IP）定义了标准的网络字节顺序。该字节顺序被用于所有设计使用在 IP 上的数据包、高级协议和文件格式上。

很多网络设备也存在存储顺序问题，即字节中的位采用大端法（最重要的位优先）或小端法（最不重要的位优先）发送。这取决于 OSI 模型最底层的数据链路层。

在以太网络的网卡中，字节数据的传输顺序同书写顺序，相当于大端方式，位数据高位在前、低位在后。

1.4 嵌入式应用系统的开发流程

嵌入式系统开发分为硬件开发和软件开发。应用系统的开发一般都采用“宿主机/目标板”的开发模式，即利用宿主机（PC）上丰富的软硬件资源、良好的集成开发环境和调试工具来调试硬件和目标板上的程序，然后通过交叉编译环境生成目标代码和可执行文件，通过联合测试行动小组（Joint Test Action Group，JTAG）接口/串行接口/USB 接口/网络接口等下载到目标板上，利用交叉调试器监控程序运行，根据调试器来观察运行的状态，实时分析、处理软硬件出现的问题。调试完成后，将目标程序下载到目标板上，完成整个开发过程。

当前嵌入式系统开发已经逐步规范化，开发过程主要包括用户系统需求、体系结构设计、系统软硬件设计、外围控制装置电路设计、抗干扰设计、系统集成、硬件调试、软件调试、系统联合调试、系统最终测试，最后形成产品。具体描述如下。

1）用户系统需求分析。根据用户的需求，确定设计任务与设计目标，提炼出系统设计说明书文本，作为设计依据和验收标准。系统的需求一般分为功能性需求和非功能性需求，功能性需求是系统的基本功能，如输入的开关量个数、输出的开关量个数、模拟量的输入/输出路数、操作方式、与外部设备的连接与通信方式等；非功能性需求包括系统的稳定性、成本、功耗、体积和重量等。

2）系统结构设计。描述系统如何实现所述的功能性需求和非功能性需求，包括对硬件、软件和执行装置的功能划分，以及系统软件、硬件的选取等。有些功能既可以用硬件实现，也可以用软件实现。用硬件实现的特点是元器件费用的投入大，但系统的运行速度快；用软件实现，系统开发的人工投入大，器件开销小，系统运行的速度较慢。一个好的结构设计是设计成功与否的关键所在。

3）系统软硬件的详细设计。为了加快产品的开发速度，软硬件的设计往往是同步进行的。对于嵌入式系统的开发，硬件的开发难度大，涉及的知识面较宽，但是它占用总开发时间

也就 10%~20%；软件的开发难度小一些，但却占用了开发的大部分时间。如果要求系统具有较高的程序运行效率且复杂度不高，可以采用裸机开发，但是它的开发周期相对长一些；如果要求系统开发周期短且系统的复杂度高，可以在移植操作系统后进行软件开发，但是系统程序的运行效率相对低一些。

4）系统软硬件联合调试。一般先进行硬件、控制装置的调试，必要时也需要控制程序的配合；然后进行软件的调试，可以先在宿主机上进行软件的仿真调试，这里主要调试一些算法或数据处理程序的结果，最后把硬件、软件和控制装置集成在一起调试。实际上集成调试时并没有严格的层次，发现哪有问题就及时解决，不断地完善系统设计存在的不足之处。

5）系统实验室测试和现场测试。对设计调试好的系统按照系统确定的任务和目标进行逐一测试，看是否满足系统的功能要求。最后客户进行现场测试，检查能否在实际的工况环境下可靠运行，否则还要进行硬件和软件的修改。

习题

1-1　简述嵌入式系统的定义。
1-2　举例说明嵌入式系统的“嵌入性”“专用性”“计算机系统”的基本特征。
1-3　简述嵌入式系统发展各阶段的特点。
1-4　简述嵌入式系统的组成。
1-5　简述嵌入式系统的类型与特点。
1-6　简述计算机组成的基本概念。
1-7　简述计算机体系结构的基本概念。
1-8　冯·诺依曼结构与哈佛结构各有什么特点？
1-9　何为 2/8 规律？
1-10　RISC 架构与 CISC 架构相比有哪些优点？
1-11　简述流水线技术的基本概念。
1-12　试说明指令流水线的执行过程。
1-13　大端存储方式与小端存储方式有什么不同？
1-14　简述嵌入式应用系统的开发流程。

第 2 章　ARM 微处理器体系结构

本章介绍 ARM 微处理器的体系结构与特点、ARM 微处理器系列芯片以及应用选型、ARM 的总线系统与接口；讲述 ARM9 体系结构的存储器组织和寄存器组织、ARM9 微处理器的工作状态与运行模式；最后介绍 ARM9 微处理器的异常。

2.1　ARM 微处理器的体系结构与特点

ARM（Advanced RISC Machines）公司 1991 年成立于英国剑桥，该公司专门从事基于 RISC 技术芯片的设计开发，主要出售芯片设计技术的授权。作为知识产权供应商，ARM 公司本身不直接从事芯片生产，靠转让设计许可由合作公司生产各具特色的芯片。半导体生产商从 ARM 公司购买其设计的 ARM 微处理器核，根据各自不同的应用领域，加入适当的外围电路，从而形成自己的 ARM 微处理器芯片（如 Samsung S3C2410X、S3C2440 等微处理器芯片都采用 ARM9 内核）进入市场，这就是 ARM 公司的“Chipless”模式。

由于全球几十家大的半导体公司（包括 Intel、Samsung、Motorola 等）普遍使用 ARM 公司的授权，因此使得 ARM 技术开发获得更多的第三方开发工具、制造和软件的支持，使得整个系统的开发成本降低，产品更容易开发，更容易被市场和消费者接受，更具有竞争力。

2.1.1　ARM 微处理器体系的结构

ARM 微处理器体系结构设计的总体思想是在不牺牲性能的情况下，尽量简化处理器，同时从体系结构的层面上灵活支持处理器扩展。这种简化和开放的思想使得 ARM 微处理器采用了很简单的结构来实现。目前，ARM 32 位体系结构被公认为是业界领先的 32 位嵌入式 RISC 微处理器内核，所有 ARM 微处理器均共享这一体系结构内核。

ARM 体系结构采用 RISC 结构，在简化处理器结构、减少复杂功能指令的同时，提高了处理器的速度。

ARM 体系结构均使用固定长度 32 位指令，使用流水线技术执行指令，大大提高了指令的执行速度；所有的指令执行都是有条件的，大大提高了指令的执行效率。

ARM 体系结构使用大量的寄存器，均为 32 位。共有 37 个物理寄存器，在逻辑上被分为若干组，这就大大加快了处理器执行指令和运行程序的速度。

ARM 体系结构采用先进的微控制器总线架构（Advanced Microcontroller Bus Architecture，AMBA）来扩展不同体系结构、具有不同读写速度的 I/O 部件。AMBA 已成为事实上的片上总线（On Chip Bus，OCB）标准。

2.1.2　ARM 微处理器体系的特点

ARM 微处理器与其他微处理器相比主要有以下特点。

- 支持 Thumb（16 位）/ARM（32 位）双指令集，能很好地兼容 8 位/16 位器件。
- 内含 32×32 位的桶形移位寄存器，左移/右移 n 位、环移 n 位和算术右移 n 位等都可以

一次完成，可以有效减少移位的延迟时间。

- 指令执行采用 3 级流水线/5 级流水线技术。
- 带有指令 Cache 和数据 Cache，大量使用寄存器，指令执行速度更快。大多数数据操作都在寄存器中完成。寻址方式灵活简单，执行效率高，指令长度固定。
- 支持大端和小端两种方式存储字数据。
- 支持 Byte（字节，8 位）、Halfword（半字，16 位）和 Word（字，32 位）三种数据类型。
- 支持用户、快速中断、普通中断、管理、中止、系统和未定义等 7 种处理器模式，除了用户模式外，其余均为特权模式。
- 处理器芯片上都嵌入了在线仿真（In Circuit Emulator-Real Time，ICE-RT）逻辑，便于通过 JTAG 来仿真调试 ARM 体系结构芯片，可以避免使用昂贵的在线仿真器。
- 具有片上总线 AMBA。AMBA 定义了 3 组总线，可以连接具有不同处理速度的集成芯片。3 组总线分别是先进高性能总线（AHB）、先进系统总线（ASB）和先进外围总线（APB）。
- 采用存储器映像 I/O 的方式，即把 I/O 端口地址作为特殊的存储器地址。
- 具有协处理器接口。ARM 允许接 16 个协处理器，如 CP15 用于系统控制，CP14 用于调试控制器。
- 采取了一些措施以降低功耗，例如降低电源电压，可工作在 3.0V 以下；减少门的翻转次数；减少门的数目，即降低芯片的集成度；降低时钟频率等。
- 体积小、成本低、性能高。

2.2 ARM 微处理器系列介绍及应用选型

ARM 微处理器系列主要有 ARM7 微处理器系列、ARM9 微处理器系列、ARM10E 微处理器系列、ARM11 微处理器系列、SecurCore 微处理器系列、Intel 的 XScale 微处理器系列、ARM 的 Cortex 微处理器系列等。其中 ARM7、ARM9、ARM10E、ARM11 为 4 个通用的处理器系列，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；ARM 的 Cortex 系列为各种不同性能要求的应用提供了一套完整的优化解决方案。

2.2.1 ARM7 微处理器系列

ARM7 微处理器系列包括 ARM7TDMI、ARM7TDMI-S、ARM720T、ARM7EJ 几种类型。其中，ARM7TMDI 是目前使用最广泛的 32 位嵌入式 RISC 微处理器，主要具有以下特点。

- 工作主频最高可达 130 MHz，高速的运算处理能力可胜任绝大多数的复杂应用。
- 采用能够提供 0.9MIPS/MHz 的三级流水线结构。
- 内嵌硬件乘法器（Multiplier），支持 16 位压缩指令集 Thumb。
- 嵌入式 ICE-RT，支持片上 Debug，支持片上断点和调试点，调试开发方便。
- 指令系统与 ARM9 系列、ARM9E 系列和 ARM10E 系列兼容，便于用户产品的升级换代。
- 支持 Windows CE、Linux、Palm OS 等操作系统。

其中命名系列中的组成字母所表示的意义如下。

T 表示支持 16 位的压缩指令集 Thumb；D 表示支持片上调试（Debug）；M 表示具有增强型乘法器（Multiplier），支持乘加运算，产生全 64 位的结果；I 表示嵌入式 ICE 芯片，可提供片上断点和调试点的支持。

ARM7 微处理器系列主要应用在工业控制、网络设备和移动电话等嵌入式系统中。

2.2.2 ARM9 微处理器系列

ARM9 微处理器系列包含 ARM920T、ARM922T 和 ARM940T 几种类型，可以在高性能和低功耗特性方面提供最佳的性能。主要具有以下特点。

- 工作主频最高可达 533 MHz，运算处理速度极高。
- 采用 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。
- 支持数据 Cache 和指令 Cache，具有更高的指令和数据处理能力。
- 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。

ARM920T 处理器核在 ARM9TDMI 处理器内核基础上，增加了分离式的指令 Cache 和数据 Cache，并带有相应的存储器管理单元 I-MMU 和 D-MMU、写缓冲器及 AMBA 接口等。

ARM9 系列微处理器主要应用于无线通信设备、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。

2.2.3 ARM 更为高级的微处理器系列

1. ARM9E 系列

ARM9E 系列微处理器包括 ARM926EJ-S、ARM946E-S 和 ARM966E-S 三种类型，以适用于不同的应用场合。

2. ARM10E 系列

ARM10E 系列主要包括 ARM1020E、ARM1022E 和 ARM1026EJ-S 三种类型，以适用于不同的应用场合。

3. ARM11 系列

ARM11 系列微处理器的新内核有 ARM1156T2-S 内核、ARM1156T2F-S 内核、ARM1176JZ-S 内核和 ARM1176JZF-S 内核。

1）ARM1156T2-S 内核和 ARM1156T2F-S 内核都基于 ARMv6 指令集体系结构，将是首批含有 ARM Thumb-2 内核技术的产品，可令合作伙伴进一步减少与存储系统相关的生产成本，主要用于多种深嵌入式存储器、汽车网络等。该体系结构中增加了汽车安全系统内安全应用产品开发非常重要的存储器容错能力。

2）ARM1176JZ-S 内核和 ARM1176JZF-S 内核也是基于 ARMv6 指令集体系结构，是首批以 ARM Trust-Zone 技术实现手持装置和消费电子装置中公开操作系统的超强安全性产品。主要为服务供应商和运营商提供新一代消费电子装置和为安全的网络下载提供支持。

2.2.4 ARM 微处理器的应用选型

鉴于 ARM 微处理器的类型和种类较多且各有千秋，随着我国嵌入式系统应用领域的逐步

扩展，ARM 微处理器将会得到极大的应用。但是，ARM 微处理器目前已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组合，给开发人员在开发选择方案时带来了一定的困难，所以对 ARM 芯片做一些对比研究还是十分必要的。一般应用时选取的原则是，先做技术层面的考虑，再考虑经济层面，然后考虑其他一些因素，如功耗、体积、可靠性等。下面叙述选择 ARM 微处理器时主要考虑的问题。

1. ARM 微处理器内核的选择

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。如果是进行裸机开发，选取的范围可以大一些，只要满足系统的要求，哪一种内核结构均可行；如果用户要使用标准的 Linux 操作系统或 Windows CE 操作系统等以减少软件的开发周期，就需要选择具有存储器管理单元（Memory Management Unit，MMU）的微处理器内核结构，如 ARM720T、ARM920T、ARM922T、Strong ARM 微处理器等。而 ARM7DMTI 内核没有 MMU，不支持标准的嵌入式 Linux 操作系统或 Windows CE 操作系统的运行。但 uCLinux 等嵌入式操作系统不需要 MMU 的支持也可很好地运行在 ARM7DMTI 硬件平台上，而且运行的稳定性很好。

2. 微处理器的工作频率

微处理器的工作频率在很大程度上决定着 ARM 微处理器的处理能力，即执行指令的速度。控制系统选取工作频率时，主要根据它的控制周期来决定，尤其对于多任务的操作系统而言，必须有足够的速度富裕度。

ARM7 系列微处理器的典型处理速度是 0.9MIPS/MHz，常见的 ARM7 系列芯片的系统主时钟频率为 20~133 MHz；ARM9 系列微处理器的典型处理速度是 1.1MIPS/MHz，常见的 ARM9 系列芯片的系统主时钟频率为 100~233 MHz。ARM10 系列芯片最高可达 700 MHz。不同芯片的微处理器外接的晶振个数不同，有的芯片只需要一个外接晶振来产生主时钟频率，可以通过锁相环（PLL）芯片分别为 ARM 内核和 USB、UART、DSP 等功能部件提供不同的时钟频率。

3. 微处理器片内外存储器和外围接口的选择

ARM 微处理器芯片有的内部含有存储器，有的内部没有。选择含有内部 RAM、ROM 的微处理器芯片可以简化电路的设计，提高系统工作的稳定性。大部分的微处理器芯片内部存储器的容量都不太大，需要用户在使用时外扩，但也有部分微处理器芯片内具有较大的存储器空间，如 ATMEL 公司的 AT91F40162 就具有高达 2MB 的片内存储器，用户在设计电路时可以进行优化选取。

几乎所有的 ARM 芯片都根据不同的应用领域而设计，扩展了相关外围电路的功能，并集成在芯片内，称之为片内外围电路。如 USB 接口、LCD 控制器、键盘接口、实时时钟电路（RTC）、模-数转换器（ADC）、数-模转换器（DAC）、集成电路内部总线控制器（I^2C）、通用异步串行接口（UART）等等。设计者应分析系统的需求，尽量采用芯片内部具有的外围接口芯片，以简化系统的硬件设计，提高系统工作的稳定性和可靠性。目前内部含有较多的存储器和外围电路的微处理器芯片是意法半导体（ST）公司生产的 STM32 芯片。

2.3 ARM 的总线系统与接口

ARM 采用先进的微控制器总线架构（AMBA），为系统应用提供了 3 个总线接口并为它们配置不同的工作频率，以适应于不同速度的芯片接入使用。为了方便调试与代码的下载，节省开发设备投入，提供了 JTAG 接口。为了扩充 ARM 系统的功能，提供了 16 个协处理器扩展

接口。

2.3.1 ARM 的总线系统

ARM 微处理器内核可以通过先进的微控制器总线架构（AMBA）来扩展不同体系架构的宏单元及 I/O 部件。AMBA 已成为事实上的片上总线（OCB）标准。AMBA 的典型系统结构如图 2-1 所示。

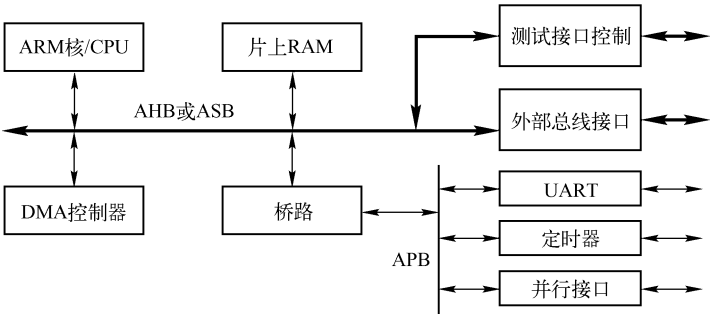

图 2-1　AMBA 的典型系统结构

AMBA 有先进系统总线（Advanced System Bus，ASB）、先进高性能总线（Advanced High-performance Bus，AHB）和先进外围总线（Advanced Peripheral Bus，APB）等 3 类总线。

- ASB 是目前 ARM 常用的系统总线，用来连接高性能系统模块，支持突发（Burst）方式数据传送。
- AHB 不但支持突发方式的数据传送，还支持分离式总线事务处理，以进一步提高总线的利用效率。特别在高性能的 ARM 架构系统中，AHB 有逐步取代 ASB 的趋势，例如在 ARM1020E 处理器核中。
- APB 为外围宏单元提供了简单的接口，也可以把 APB 看作 ASB 的余部。

AMBA 通过测试接口控制器（Test Interface Controller，TIC）提供了模块测试的途径，允许外部测试者作为 ASB 总线的主设备来分别测试 AMBA 上的各个模块。

AMBA 中的宏单元也可以通过 JTAG 方式进行测试。虽然 AMBA 的测试方式通用性稍差，但其通过并行口的测试比 JTAG 的测试代价也要低些。

2.3.2 ARM 的 JTAG 调试接口

1. JTAG 接口介绍

联合测试行动小组（JTAG）是一种国际标准测试协议，主要用于芯片内部测试及对系统进行仿真、调试。JTAG 技术是一种嵌入式调试技术，它在芯片内部封装了专门测试电路，即集成了测试访问口（Test Access Port，TAP），通过专用 JTAG 测试工具对内部节点进行测试。目前大多数比较复杂的器件都支持 JTAG 协议，如 ARM、FPGA 器件等。

JTAG 测试允许多个器件通过 JTAG 接口串联在一起，形成一个 JTAG 链，能实现对各个器件分别测试。JTAG 接口还常用于实现在系统编程（In-System Programmable，ISP）功能，如对 Flash 器件进行编程等。

通过 JTAG 接口，可对芯片内部的所有部件进行访问，因而是开发调试嵌入式系统的一种简洁高效的手段。

目前 ARM 公司提供的 JTAG 接口有 14 针接口和 20 针接口两种标准，具体硬件电路和引脚功能请参考其他资料。

2. ARM 的 JTAG 接口的实际使用

ARM 的 JTAG 接口就像单片机的仿真器和编程器（或称烧录器），单片机仿真器和编程器的价格平均都在千元以上，ARM 的 JTAG 接口价格与之相比非常低廉。仿真器主要用于单片机软硬件的调试，编程器用于将编译好的十六进制或二进制程序烧写在单片机的 ROM 中。JTAG 接口在 ARM 的裸机程序开发中作用巨大，同时在 ARM 的 Linux 操作系统下开发应用必不可少。

在使用 ARM 公司提供的集成开发环境 ADS1.2 进行裸机开发应用系统时，PC 并行口与 ARM 的 JTAG 接口相连，可以进行 ARM 目标板的软硬件调试。调试完成后，编译好的 ARM 机器码也要通过 JTAG 接口下载（烧写）到目标板的 Flash 存储器中运行。

进行 Linux 操作系统环境下的应用系统开发时，编译好的引导启动程序（Bootloader）必须通过 ARM 的 JTAG 接口下载到 ARM 芯片外扩的 NOR Flash（称“非或”Flash）ROM 或 NAND Flash（称“非与”Flash）ROM 中，其后的操作系统内核、文件系统、应用程序可以通过 RS-232 接口、USB 接口或 RJ45 网络接口下载到 ARM 的 Flash ROM 中。也就是说，Bootloader 中已经编写了有关接口的驱动、应用程序才能完成其后的操作。

2.3.3 ARM 的协处理器接口

为了便于片上系统（System on a Chip，SoC）的设计，ARM 可以通过协处理器（CP）来支持一个通用功能指令集的扩充，通过增加协处理器来增加 ARM 系统的功能。

在逻辑上，ARM 可以扩展 16 个协处理器（CP0～CP15），其中 CP15 作为系统控制，CP14 作为调试控制器，CP4～CP7 作为用户控制器，CP8～CP13 和 CP0～CP3 保留。每个协处理器可有 16 个寄存器。例如，MMU 和保护单元的系统控制都采用 CP15 协处理器，JTAG 调试中的协处理器为 CP14，即调试通信通道（Debug Communication Channel，DCC）。

ARM 微处理器内核与协处理器接口有以下 4 类。

1）时钟和时钟控制信号：MCLK、nWAIT、nRESET。

2）流水线跟随信号：nMREQ、SEQ、nTRANS、nOPC、TBIT。

3）应答信号：nCPI、CPA、CPB。

4）数据信号：双向数据信号 D[31:0]、输入数据信号 DIN[31:0]、输出数据信号 DOUT[31:0]。

在协处理器的应答信号中，部分信号说明如下。

- nCPI 为 ARM 微处理器至 CPn 协处理器的信号，该信号低电平有效代表“协处理器指令”，表示 ARM 微处理器内核标识了 1 条协处理器指令，希望协处理器去执行它。
- CPA 为协处理器至 ARM 处理器的内核信号，表示协处理器不存在，目前协处理器无能力执行指令。
- CPB 为协处理器至 ARM 处理器的内核信号，表示协处理器忙，还不能开始执行指令。

协处理器也采用流水线结构，为了保证与 ARM 微处理器内核中的流水线同步，在每一个协处理器内需有 1 个流水线跟随器（Pipeline Follower），用来跟踪 ARM 微处理器内核流水线中的指令。由于 ARM 的 Thumb 指令集无协处理器指令，协处理器还必须监视 TBIT 信号的状态，以确保不把 Thumb 指令误解为 ARM 指令。

协处理器也采用 Load/Store 结构，用指令来执行寄存器的内部操作，从存储器取数据至寄

存器或把寄存器中的数据保存至存储器中，以及实现与 ARM 处理器内核中寄存器之间的数据传送。而这些指令都由协处理器指令来实现。

2.4 ARM9 体系结构的存储器组织

ARM9 的存储器层次结构从微处理器的 CPU 到外依次是寄存器组、Cache 存储器、主存储器和辅助存储器。寄存器组的访问一般需要几个纳秒，Cache 存储器的访问需要十几个纳秒，主存储器一般为几兆字节到 1 GB 的动态存储器，访问时间约 50 ns。

寄存器组主要辅助 CPU 进行运算，以加快 CPU 的处理速度；Cache 主要预存将要执行的指令以及相关的数据；主存储器是运行程序和访问数据的所在地。

ARM9 处理器有的内置指令 Cache 和数据 Cache，但不带有片内 RAM 和片内 ROM。系统所需的 RAM 和 ROM（包括 Flash）都通过总线外接。由于系统的地址范围较大（$2^{32}=4$ GB），有的片内还带有存储器管理单元（Memory Management Unit，MMU）。ARM 架构处理器还允许外接 PC 内存卡国际联合会（Personal Computer Memory Card International Association，PCMCIA）卡。

2.4.1 ARM 体系结构的存储器空间

ARM9 体系结构的存储器和 I/O 端口采用统一编址。通常对于 I/O 端口的编址有 2 种方式：独立编址、与存储器采用统一编址。前者不占用处理器的存储器空间，但需要设计专用的指令，访问速度快，但它不太适用于 RISC 系统；后者占用了存储器空间的一部分，与存储器采用同一指令进行访问，访问速度相对慢一些，它符合 RISC 系统的要求。

ARM9 体系结构存储器与 I/O 端口采用的统一编址所构成的地址空间称为平面地址空间，亦可叫作线性地址空间，如由 32 根地址线组成的 2^{32} 字节地址单元，范围从 0x00000000～0xffffffff，将字节地址作为无符号数对待。

2.4.2 ARM9 中的大端存储与小端存储

前面已经讲到，ARM9 的每个地址单元是对应一个存储字节单元而不是一个存储字（占 4 字节的存储单元），但 ARM9 可以按存储字访问，也可以按半字（2 字节）访问或单字节访问。在按字进行访问时，要求其地址是字对齐的，即字地址可以被 4 整除，也就是说字地址的最低 2 位 A1A0=00，程序计数器 PC 指针的高 30 位使用地址线 $A_{32}A_{31}\cdots A_3A_2$。，最低 2 位地址线的值取 00，是默认的，也是必须的，因为按字地址访问时低 2 位必须有值。这样按字访问的第 1 字节的数据默认存储在 A1A0=00 的字节地址单元、第 2 字节的数据存储在 A1A0=01 的字节地址单元、第 3 字节的数据存储在 A1A0=10 的字节地址单元、第 4 字节的数据存储在 A1A0=11 的字节地址单元。因此 PC 地址指针最低 2 位为 00，暗示用字地址访问时其对应字节数据的内存连续访问单元次序是 00，01，10，11 顺序组合，其他均为非字对齐。注意连续字地址存储最低 2 位的变化是 00，01，10，11，00，01，10，11，00，…，每一个 00，01，10，11 的组合都对应着一个固定的高 30 位字地址值。如果取地址最低 2 位的组合是 01，10，11，00 字节单元的值，则前 3 个组态对应一个高 30 位的字地址值，最后一个 00 对应的值是前高 30 位字地址值+4，字数据不在同一个高 30 位地址所指的范围内，属于非字对齐。

一个字是由 4 个字节组成，如果某个字的地址是 A（A 必须能被 4 整除），那么该字的 4 个字节对应的地址依次是 A、A+1、A+2、A+3。

同理，对于由 2 个字节组成的半字，访问地址使用地址线高 31 位，最低位默认取 0，即该地址值能被 2 整除。半字对齐时要求高 31 位地址不变，第 1 字节数据的默认最低位地址值必须为 0，第 2 字节数据的默认最低位地址值必须为 1。

ARM 存储系统可以使用小端存储或者大端存储两种方式。大端存储就是将字数据的最高 8 位字节数据存放在字节地址最小的存储单元中，将字数据的最低 8 位字节数据存放在字节地址最大的存储单元中，如图 2-2 所示。

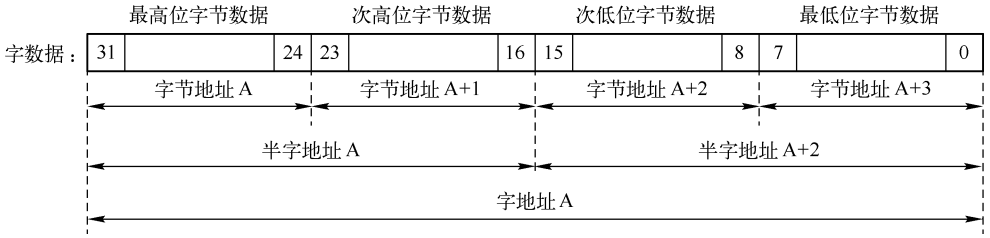

图 2-2　大端存储方式

而小端存储与大端存储正好相反，它的定义是，将字数据的最高 8 位字节数据存放在字节地址最大的存储单元中，将字数据的最低 8 位字节数据存放在字节地址最小的存储单元中，如图 2-3 所示。

图 2-3　小端存储方式

小端存储方式是 ARM9 处理器的默认方式。ARM9 指令集中，没有相应的指令集来选择是采用大端存储方式还是小端存储方式，但可以通过外部硬件接入引脚来配置它。如果外部引脚 BIGEND 接高电平，则使用大端存储方式；如果外部引脚 BIGEND 接低电平，则使用小端存储方式。

ARM9 对于存储器单元的访问要求字对齐或半字对齐，即访问字存储单元时，要求字对齐（PC 指针的值能被 4 整除）；访问半字存储单元时，要求半字对齐（PC 指针的值能被 2 整除）。如果没有按照这种对齐方式对存储单元访问，称为非对齐存储器访问。非对齐的存储器访问可能会引起不可预知的状态。

2.4.3　I/O 端口的访问方式

对于 I/O 端口的访问，ARM9 体系结构是应用存储器映射的方式来实现的。由于 I/O 端口与存储器采用统一编址，I/O 端口必然要占用存储器空间。存储器映射法就是为每个 I/O 端口分配特定的存储器地址，当从这些地址读出或向该地址写入数据时，实际完成的是 I/O 端口的操作功能。

注意：存储器映射 I/O 端口地址的行为通常不同于对一个正常存储器地址操作所期望的行为。例如，从一个正常存储器地址两次连续的读入，2 次返回的值相同。而对于存储器映射 I/O

地址，第 2 次读入的返回值可以不同于第 1 次读入的返回值，因为第 2 次读入时对应端口的电平可能发生了变化。

2.5 ARM9 微处理器的工作状态与运行模式

ARM9 微处理器有两种工作状态，以支持 32 位 ARM 或 16 位紧凑型长度 Thumb 指令的运行，可以在程序的执行过程中任意切换。ARM9 具有 7 种异常运行模式，各种异常模式下都有自己的寄存器组，以便于异常程序的执行和返回。

2.5.1 ARM9 微处理器的工作状态

ARM9 微处理器有 32 位 ARM 和 16 位 Thumb 两种工作状态。在 32 位 ARM 状态下执行字对齐的 ARM 指令，在 16 位 Thumb 状态下执行半字对齐的 Thumb 指令。在 ARM 指令集和 Thumb 指令集中均有切换处理器状态的指令，并可在两种状态之间进行互相切换。在系统上电或复位时，微处理器处于 ARM 状态。

进入 Thumb 状态：当操作数寄存器的状态位，即位［0］为 1 时，执行 BX 指令使微处理器从 ARM 状态切换到 Thumb 状态。

当微处理器处于 Thumb 状态时发生了异常（如 IRQ、FIQ、SWI 等），则当异常处理返回时，自动切换到 Thumb 状态。

在 Thumb 状态下，程序计数器 PC 使用位［1］选择另一个半字。

切换到 ARM 状态：当操作数寄存器的状态位，即位［0］为 0 时，执行 BX 指令使微处理器从 Thumb 状态切换到 ARM 状态。

当处理器进行异常处理时，把当前的 PC 指针存入相应的异常模式连接寄存器（该模式下的 R14 寄存器）中，并从异常向量入口地址开始执行程序，执行完毕后将其连接寄存器的值送入 PC（还要做一些偏移量的处理），程序返回到主程序。

ARM 处理器在两种工作状态之间可以切换，切换不影响处理器的模式或寄存器的内容。

2.5.2 ARM9 微处理器的运行模式

1. ARM9 微处理器支持 7 种运行模式

- 用户模式 usr：ARM 微处理器正常程序执行模式，用户程序都在这种模式下执行。
- 快速中断模式 fiq：当一个高优先级的快速中断源产生中断时进入这种模式，主要用于高速数据传输或通道处理。
- 普通中断模式 irq：当一个普通优先级的中断源产生中断时进入这种模式，用于一般的中断事务处理。
- 管理模式 svc：当复位或软中断指令执行时将进入这种模式，是供操作系统使用的一种保护模式。
- 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，用于虚拟存储及存储保护。
- 系统模式 sys：供需要访问资源的操作系统任务使用，运行具有特权的操作系统任务。
- 未定义指令终止模式 und：当执行未定义的指令时进入该模式。

ARM 处理器的运行模式可以在特权模式下通过软件改变，也可以通过外部中断和异常处理改变；大多数的应用程序运行在用户模式下，当运行在用户模式下时，被保护的系统资源是

不能访问的。

2. 特权模式与异常模式

（1）特权模式

除用户模式外，其余 6 种模式被称为特权模式。

用户模式的特点是，用户程序不能访问受操作系统保护的系统资源，也不能进行处理器模式的切换。

特权模式的特点是，应用程序可以访问所有的系统资源，可以任意地进行处理器模式的切换。

（2）异常模式

除用户和系统模式外，其余模式被称为异常模式。

系统模式的特点是，不能通过异常进入该模式，可以访问系统的所有资源，可以任意地进行处理器模式的切换。

异常模式的特点是，以各自的异常方式或中断方式进入，并且处理各自异常或中断。对于管理模式 svc 异常进入方式和处理的内容如下。

1）系统上电复位或按下 RESET 按钮后进入管理模式。处理任务有 ARM 系统初始化、关闭中断、设置系统 3 个总线的频率、配置动态存储器 SDRAM、各种运行模式下堆栈区的设置、各个模块的初始化、为 C/C++应用程序提供运行环境等。

2）当执行软中断指令（SWI）异常时，也可进入管理模式。

3. 处理器运行模式间的切换

处理器运行模式间有 2 种切换方式：一种是通过软件控制进行切换，另一种是由外部中断或内部的异常触发进行切换。前者是通过编写软件来实现的，后者是自动触发的。系统启动时运行模式的转换流程如下。

1）上电复位或按复位按钮，进入到管理模式，此时主要工作是关闭中断，设置系统 3 个总线的频率、配置动态存储器（SDRAM）等。

2）设置 1）模式下的堆栈指针后，通过使用软件改变 CPSR 的最低 5 位的模式控制位 M[4:0]，即模式字，系统进入某种特权模式，设置它的堆栈指针；再一次改变模式字，进入相应的运行模式，进行该模式的堆栈指针设置，直到设置完所有模式下的堆栈指针等。需要注意的是，此期间系统一直处于特权模式下。

3）最后通过软件模式字的更改，使系统进入到用户模式工作，运行应用系统的软件。此时若各种异常模式产生了异常，进入相应的异常模式处理，处理完后返回到异常发生时的模式程序处，继续执行原异常模式程序。

注意：异常模式是有优先级的，如果同时产生异常，优先级高的模式优先处理执行；另外，模式优先级高的可以在模式优先级低的模式中产生异常，处理完后返回原处。

ARM 微处理器在每一种处理器模式下均有一组相应的寄存器与之对应。即在任意一种处理器模式下，可访问的寄存器包括 15 个通用寄存器（R0~R14）、1~2 个状态寄存器和程序计数器。在所有的寄存器中，有些是在 7 种处理器模式下共用的同一个物理寄存器，而有些寄存器则是在不同的处理器模式下有不同的物理寄存器。

2.6 ARM9 体系结构的寄存器组织

ARM 微处理器的 37 个物理寄存器被安排成部分重叠的组，它们不是在任何模式下都可以

使用的，寄存器的使用与处理器状态和工作模式有关。如图 2-4 所示，每种处理器模式使用不同的寄存器组。其中 15 个通用寄存器（R0～R14）、1 或 2 个状态寄存器和程序计数器是通用的。图 2-4中有背景阴影的寄存器均为独立的物理寄存器，R0～R14、PC、CPSR 这 17 个寄存器也是独立的物理寄存器，共有 37 个物理寄存器。

ARM运行模式

特权模式

异常模式

用户	系统	管理	终止	未定义	中断	快中断
R0	R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7	R7
R8	R8	R8	R8	R8	R8	R8_fiq
R9	R9	R9	R9	R9	R9	R9_fiq
R10	R10	R10	R10	R10	R10	R10_fiq
R11	R11	R11	R11	R11	R11	R11_fiq
R12	R12	R12	R12	R12	R12	R12_fiq
R13	R13	R13_svc	R13_abt	R13_und	R13_irq	R13_fiq
R14	R14	R14_svc	R14_abt	R14_und	R14_irq	R14_fiq
PC	PC	PC	PC	PC	PC	PC
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
		SPSR_svc	SPSR_abt	SPSR_und	SPSR_irq	SPSR_fiq

图 2-4　寄存器组织结构图

2.6.1　通用寄存器

通用寄存器（R0～R15）可分成未分组寄存器 R0～R7、分组寄存器 R8～R14 和程序计数器 R15 三类。

1. 未分组寄存器（R0～R7，8 个）

在所有的运行模式下，未分组寄存器都指向同一物理寄存器，它们没有被系统作特殊的用途。因此，在中断或异常处理进行运行模式转换时，由于不同的微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，这点在程序设计时要引起注意。未分组寄存器 R0～R7 是真正的通用寄存器，可以工作在所有的处理器模式下，没有隐含的特殊用途。

2. 分组寄存器（R8～R14）

对于分组寄存器，它们每一次所访问的物理寄存器与微处理器当前的运行模式相关。分组寄存器 R8～R14 的使用取决于当前的处理器模式，每种模式有专用的分组寄存器以加快异常处理的速度。

寄存器 R8～R12 可分为两组物理寄存器。一组用于 FIQ 模式，另一组用于除 FIQ 以外的

其他模式。第一组是 R8_fiq~R12_fiq，在进行快速中断处理时使用。第二组是在除 FIQ 模式以外的其他处理器运行模式中直接使用 R8~R12，需要注意在运行模式切换时寄存器中内容的使用。寄存器 R8~R12 没有任何指定的特殊用途。

寄存器 R13~R14 可分为 6 个分组的物理寄存器。一组用于用户模式和系统模式，而其他 5 组分别用于 svc、abt、und、irq 和 fiq 五种异常模式。访问时需要指定它们的模式，如：R13_<mode>，R14_<mode>；其中<mode>可以是 svc、abt、und、irq 和 fiq 模式中的一个。

寄存器 R13 通常用作堆栈指针，称作 SP。每种异常模式都有自己独立的物理寄存器 R13，在 Bootloader 中或在 ARM 应用系统的初始化过程中，一般都要初始化每种模式下的 R13，即堆栈指针，使其指向该运行模式下的内存堆栈空间。在异常处理程序的入口处，将用到的其他寄存器的值保存到该指针所指向的内存堆栈中；返回时，重新将这些值加载到寄存器。这种异常处理方法保证了异常出现后不会导致执行程序的状态不可靠。

寄存器 R14 用作子程序链接寄存器，也称为链接寄存器（Link Register，LR），在上述的 6 个分组中都具有独立的物理寄存器，用于保存程序在发生异常时当前程序的 PC 指针值，使微处理器在执行完异常处理程序时能够返回到原程序。当执行带链接分支（BL）指令时，得到 R15 的备份。

当中断或异常出现时，或者当中断或异常程序执行 BL 指令时，相应的分组寄存器 R14_svc、R14_irq、R14_fiq、R14_abt 和 R14_und 用来保存 R15 的返回值。在其他情况下，R14 也可以作为通用寄存器使用。

FIQ 模式有 7 个分组的寄存器 R8~R14，映射为 R8_fiq~R14_fiq，它们是独立的物理寄存器。在 ARM 状态下，FIQ 异常处理程序没有必要保存其他运行模式下的 R8~R14 寄存器，达到了快速中断的要求。其他运行模式中都包含两个分组的寄存器 R13 和 R14 的映射，它们也是独立的物理寄存器，允许每种模式都有自己的堆栈和链接寄存器。

3. 程序计数器（R15）

寄存器 R15 用作程序计数器（PC）。在 ARM 状态，位[1:0]为 0，位[31:2]保存 PC 值。在 Thumb 状态，位[0]为 0，位[31:1]保存 PC 值。R15 虽然也可用作通用寄存器，但一般不这样用，因为对 R15 的使用有一些特殊限制，当违反了这些限制时，程序执行结果是不可预知的。

由于 ARM 体系结构采用了多级流水线技术，对于 ARM 指令而言，PC 总是指向当前执行指令的下 2 条指令的地址，即 PC 的值是当前指令的值+8（字节）。

- 读程序计数器。指令读出的 R15 值是指令地址加上 8 字节。由于 ARM 指令始终是字对齐的，所以读出结果值的位[1:0]总是 0。读 PC 主要用于快速地对临近的指令和数据进行位置无关寻址，包括程序中的位置无关转移。
- 写程序计数器。写 R15 的通常结果是将写到 R15 中的值作为指令地址，并以此地址发生转移。由于 ARM 指令要求字对齐，通常希望写到 R15 中值的位[1:0]=0b00。

2.6.2 程序状态寄存器

程序状态寄存器有两类，一类是当前程序状态寄存器（Current Program Status Register，CPSR），通常也可以叫作 R16 寄存器，它是所有运行模式下的公用寄存器，用于保存当前运行模式下的状态字，在所有处理器运行模式下都可以使用；还有一类叫作程序状态保存寄存器（Saved Program Status Register，SPSR），从图 2-4 中可以看出 SPSR 几乎在每种运行模式下都

以独立的物理寄存器而存在，它在异常发生转换时保存 CPSR 的值，以保证在异常返回时程序的状态字不变，它与 CPSR 有相同的内容格式，它们的字结构组织格式如图 2-5 所示。模式位控制字及其使用的寄存器见表 2-1。

b31	b30	b29	b28	b27 … b8	b7	b6	b5	b4	b3	b2	b1	b0
N	Z	C	V	系统保留部分位	I	F	T	M4	M3	M2	M1	M0

图 2-5　程序状态寄存器格式图

CPSR 包含有条件标志位（N、Z、C、V）、中断禁止位（I、F）、当前处理器模式位（T）以及其他状态和控制信息位（M4～M0）。具体代表的物理意义如下。

1. 条件标志位

N、Z、C、V（Negative、Zero、Carry、oVerflow）均为条件码标志位（Condition Code Flags），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。

表 2-1　模式位控制字与运行模式及使用的寄存器

M[4:0]	处理器运行模式	可访问的寄存器
10000	用户模式	R0～R14，PC，CPSR
10001	FIQ 模式	R0～R7，R8_fiq～R14_fiq，PC，CPSR，SPSR_fiq
10010	IRQ 模式	R0～R12，R13_irq，R14_irq，PC，CPSR，SPSR_irq
10011	管理模式	R0～R12，R13_svc，R14_svc，PC，CPSR，SPSR_svc
10111	中止模式	R0～R12，R13_abt，R14_abt，PC，CPSR，SPSR_abt
11011	未定义模式	R0～R12，R13_und，R14_und，PC，CPSR，SPSR_und
11111	系统模式	R0～R14，PC，CPSR（ARM v4 及以上版本）

CPSR 中的条件标志位是由 ARM 指令进行清 0 置 1 的，大部分的汇编指令只有带上对条件标志位有影响的字符“S”时才能起作用，少数的比较指令（CMN、CMP、TEQ、TST）不带影响字符“S”也可对条件标志位起作用。

在 ARM 状态下，绝大多数指令的执行，CPSR 条件标志位是决定因素之一，另一个因素就是在汇编指令中必须带上执行的条件标识字符。ARM 为了提高汇编指令的执行效率，指令的执行是有条件的。

条件标志位的通常含义如下。

1）负号标志位 N：如果结果是带符号的二进制补码，那么，若结果为负数，则 N=1；若结果为正数或 0，则 N=0。

2）零标志位 Z：若指令的执行结果为 0，则置 1（通常表示比较的结果为“相等”），否则清 0。

3）进位标志位 C：可用如下 4 种方法之一进行设置。

- 执行加法指令（包括比较指令 CMN）时，若加法产生进位（即无符号溢出），则 C 置 1；否则置 0。
- 执行减法指令（包括比较指令 CMP）时，若减法产生借位（即无符号溢出），则 C 置 0；否则置 1。
- 对于结合移位操作的非加法/减法指令，C 值为移出值的最后 1 位。
- 对于其他非加法/减法指令，C 通常不改变。

4）溢出标志位 V：可用如下两种方法设置。

- 对于加法或减法指令，如果操作数和结果都是补码形式的带符号整数，当发生带符号溢出时，V 置 1。
- 对于非加法/减法指令，V 通常不改变。

2. 控制位

程序状态寄存器 CPSR 的最低 8 位 I、F、T 和 M[4:0]用作控制位。当异常出现时改变控制位。处理器在特权模式下时也可由软件改变。

（1）中断禁止位 I、F

1）普通中断控制位 I：置 1 禁止 IRQ 普通中断；清 0 允许 IRQ 普通中断。

2）快速中断控制位 F：置 1 禁止 FIQ 快速中断；清 0 允许 FIQ 快速中断。

（2）ARM/Thumb 状态控制位 T

T=0：指示工作在 ARM 状态；T=1：指示工作在 Thumb 状态。

（3）模式控制位

M4、M3、M2、M1 和 M0（M[4:0]）是模式控制位，也是状态信息位。它的取值组合决定处理器的运行模式，表 2-1 列出了模式位控制字以及该模式下可以使用的寄存器。并非所有的模式位组合都能定义一种有效的处理器模式。其他组合的结果不可预知。

3. 其他位

程序状态寄存器的其他位目前保留，用做以后功能的扩展。

有关 Thumb 寄存器的组织结构请参考相关书籍，这里不再赘述。

2.7 ARM9 微处理器的异常

2.7.1 ARM9 微处理器异常的概念

1. ARM 异常

在一个正常的程序执行过程中，由内部或外部源产生的一个事件使正常的程序执行产生暂时停止的状态时，称之为异常。异常是由内部或外部源产生并引起处理器处理的一个事件，例如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，当异常处理完成之后，恢复保留的当前处理器状态，继续执行当前程序。多个异常同时发生时，处理器将会按固定的优先级进行处理。

2. ARM 体系结构中的异常与中断

此处所指异常与 8 位单片机的中断有相似之处，但异常与中断的概念并不完全等同，例如，外部中断或试图执行未定义指令都会引起异常。中断包含在异常中。

2.7.2 ARM 体系结构的异常类型

ARM 体系结构支持 7 种类型的异常，异常类型、所处的运行模式、异常向量入口地址、优先级的对应关系见表 2-2。

异常出现后，强制从异常类型对应的固定存储器地址开始执行程序，在特定异常向量地址 ROM 空间存放一条跳转指令，跳转到异常处理程序的入口处。这些固定的地址称为异常向量（Exception Vectors）。

优先级是指当有多个异常发生时，优先级高的先执行。

表 2-2　异常类型和异常处理模式表

异 常 类 型	所处的运行模式	异常向量地址	优　先　级
复位	管理模式（svc）	0x00000000	1（最高）
未定义指令	未定义模式（und）	0x00000004	6（最低）
软件中断	管理模式（svc）	0x00000008	6（最低）
指令预取终止	终止模式（abt）	0x0000000C	5
数据访问终止	终止模式（abt）	0x00000010	2
IRQ（外部中断）	IRQ 模式（irq）	0x00000018	4
FIQ（快速中断）	FIQ 模式（fiq）	0x0000001C	3

2.7.3　各种异常类型的含义

1. 复位异常

当处理器的复位电平有效时，产生复位异常，ARM 处理器立刻停止执行当前指令。复位后，ARM 处理器在禁止中断的管理模式下，程序跳转到复位异常处理程序处执行（从地址 0x00000000 或 0xFFFF0000 开始执行指令）。

2. 未定义指令异常

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。当 ARM 处理器执行协处理器指令时，必须等待任一外部协处理器应答后，才能真正执行这条指令。

若协处理器没有响应，就会出现未定义指令异常。

若试图执行未定义的指令，也会出现未定义指令异常。

未定义指令异常可用于在没有物理协处理器（硬件）的系统上，对协处理器进行软件仿真，或在软件仿真时进行指令扩展。

3. 软件中断（SWI）异常

软件中断异常由执行 SWI（SoftWare Interrupt）指令产生，可使用该异常机制实现系统功能调用，用于用户模式下的程序调用特权操作指令，以请求特定的管理（操作系统）函数。

4. 指令预取中止异常

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存储器中止（Abort）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。

5. 数据访问中止异常

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，产生数据访问中止异常。存储器系统发出存储器中止信号。响应数据访问（加载或存储）激活中止，标记数据为无效。

6. 外部中断请求（IRQ）异常

当处理器的外部中断请求引脚 nIRQ 有效，且 CPSR 中的 I 位为有效电平 0 时，IRQ 才会产生异常。系统的外设可通过该异常请求中断服务。IRQ 异常的优先级比 FIQ 异常的低。当进入 FIQ 处理时，会屏蔽掉 IRQ 异常。

注意：CPSR 中的 I 位，只有在特权模式下才能使用软件设置。

7. 快速中断请求（FIQ）异常

当处理器的快速中断请求引脚 nFIQ 有效，且 CPSR 中的 F 位为有效电平 0 时，FIQ 才会出现异常。FIQ 支持数据传送和通道处理，并有足够的私有寄存器，处理速度将会大大提高。**注意**事项同上。

2.7.4 异常的响应过程

当一个异常发生后，ARM 微处理器执行时有以下几步操作。

1）系统根据异常类型调整当前的 PC 指针值后，存入链接寄存器（R14），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。

2）系统硬件自动将当前程序状态寄存器（CPSR）的条件状态位、控制位、运行模式字复制到相应的 SPSR 中保存，以便异常返回时使用。

3）根据异常类型，系统硬件强制设置当前程序状态寄存器（CPSR）的 I、F、T 位及运行模式 M[4:0]位，禁止普通中断和快速中断。

前已讲述，ARM 的 PC 指针是指向当前指令的下 2 条指令地址，即 PC+8。由于指令的执行是按流水线进行的，每种异常模式下指令最终执行完了哪一条是与异常类型有关的，所以送入到 RL 中的内容是不同的。还有返回时也可对 RL 中的内容进行调整，实现指令的无缝响应与返回。表 2-3 是进入异常模式前和返回时 RL 值的调整以及使用的 ARM 指令。若异常是从 Thumb 状态进入，则在 RL 寄存器中保存的是当前 PC 值的偏移量。

表 2-3 RL 值变化与使用的 ARM 指令

异常类型	进入前的 RL 值		返回时使用的 ARM 指令
	ARM R14_x	Thumb R14_x	
软件中断（SWI）异常	PC+4	PC+2①	MOVS PC，R14_svc
未定义指令异常	PC+4	PC+2①	MOVS PC，R14_und
程序终止异常	PC+4	PC+4①	SUBS PC，R14_abt，#4
数据访问终止异常	PC+8	PC+8③	SUBS PC，R14_abt，#8
普通中断异常	PC+4	PC+4②	SUBS PC，R14_irq，#4
快速中断异常	PC+4	PC+4②	SUBS PC，R14_fiq，#4
子程序调用指令 BL	PC+4	PC+2①	MOV PC，R14
复位异常	与 RL 的值无关		上电或系统复位④

① 在此 PC 应是预取终止的 BL 指令/SWI/未定义指令所取的地址。

② 在此 PC 是从 IRQ/FIQ 取得不能执行的指令的地址。

③ 在此 PC 是产生访问数据终止时加载或存储指令的地址。

④ 强制 PC 指针指向相关的异常向量地址处，取出下一条指令执行，跳转到相应的异常处理程序。

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到 PC 时，处理器自动切换到 ARM 状态。

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）系统是在执行上述异常发生后第二步中的指令（如：SUBS PC，R14_fiq，#4）时，硬件自动将 SPSR 内容送回 CPSR 中，恢复原来的运行模式值。或者说恢复 CPSR 的内容和 RL 值送 PC 使用一条指令就可完成。

2）使用指令将连接寄存器（LR）的值减去相应的偏移量后送到 PC 中，每种异常减去的

具体偏移量值见表 2-3。

3）若在进入异常处理时设置了普通中断禁止位、快速中断禁止位，要在此清除。但有新的异常发生时，处理器可以进行新的异常处理。

可以认为应用程序总是从复位异常处理程序开始执行，因此复位异常处理程序不需要返回。

2.7.5 应用程序中的异常处理

在应用程序的设计中，异常处理采用的方式是在异常向量入口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，从而跳转到异常入口处理程序，当异常处理完成以后，返回到主程序继续执行。

习题

2-1 简述 ARM 微处理器的特点。
2-2 试分析 ARM920T 内核结构特点。
2-3 简述 ARM7 微处理器的主要特点。
2-4 简述 ARM9 微处理器的主要特点。
2-5 简述 ARM 微处理器的应用选型。
2-6 ARM 使用的先进微控制器总线结构 AMBA 的主要内容是什么？
2-7 JTAG 接口的主要作用是什么？
2-8 ARM 可以扩展的协处理器有多少？协处理器的主要作用是什么？
2-9 简述 ARM 中的字对齐，半字对齐。
2-10 简述 ARM 访问 I/O 端口的方式。
2-11 ARM 微处理器支持哪几种运行模式？各运行模式有什么特点？
2-12 ARM 处理器有几种工作状态？各工作状态有什么特点？
2-13 ARM 运行模式中的特权模式和异常模式的定义。
2-14 简述 ARM 寄存器组织结构图，并说明寄存器分组与功能。
2-15 简述程序状态寄存器的位功能。
2-16 简述 ARM 中的程序计数器 PC、各模式下的堆栈指针寄存器和连接寄存器是什么？
2-17 ARM 体系结构支持几种类型的异常？并说明其异常处理模式和优先级状态。
2-18 简述异常类型的含义。
2-19 简述 ARM 的异常响应过程。
2-20 简述 ARM 在应用程序中是如何进行异常处理的。

第 3 章　ARM 微处理器指令系统

ARM 指令系统有标准 32 位的 ARM 指令集和 16 位的 Thumb 指令集，通常默认为前者。ARM 指令系统是本章所介绍的主要内容，以后简称 ARM 指令集。

不管是进行裸机的 ARM 应用系统开发，还是在移植了操作系统的基础上进行开发，学习 ARM 汇编指令都是必不可少的。因为 ARM 微处理器要正常工作，就必须配置好它的硬件环境，这相当于 PC 主板上的 BIOS 固化程序；在 ARM 中是系统启动引导程序（Bootloader），就需要使用 ARM 汇编指令来编写。因此学好 ARM 指令是做好 ARM 应用系统开发的关键。

本章介绍 ARM9 微处理器的指令格式与特点、ARM9 的寻址方式，分类讲述 ARM9 指令的功能，并给出了应用示例进行具体介绍。

3.1　ARM9 的指令格式

ARM 指令集中的指令均为单字指令，它为指令的流水线执行创造了条件。大部分指令都以寄存器作为其操作数，指令执行速度快，寻址方式灵活多样。但是对于 32 位的 ARM 处理器来讲，指令中的立即数却有特殊的限制，必须满足一定的条件。当要向某一寄存器送入任意 32 位的立即数时，必须使用 ARM 中定义的“假伪指令”，LDR 来实现。之所以称为“假伪指令”，是因为它与其他伪指令不同，编译器通过文字池将它等效为 ARM 指令，因此它有机器码。

3.1.1　ARM9 微处理器的指令格式与特点

1. ARM 指令的特点

- ARM 指令都是单字指令，占 32 位。基本指令只有 36 条。
- 指令可以有条件执行，也可以无条件执行。ARM 指令的一个重要特点是几乎所有的指令都带有一个可选的条件码，可根据当前程序状态寄存器（CPSR）中的条件标志位来决定是否执行该指令。当指令带有条件码并且条件满足时执行该指令，条件不满足时该指令被当作一条空操作（NOP）指令。
- 灵活的寻址方式。计算机的寻址方式是计算机的主要性能特征之一，ARM 指令有 7 种基本寻址方式，5 种复合寻址方式。由于 ARM 指令都占 32 位，在大多数情况下可以有 3 个操作数，其中第一个操作数（目的操作数）一般为基本操作数寻址方式，第 2 个和第 3 个操作数采用复合寻址方式。ARM 指令的重要特点是具有灵活的第 2 个操作数，既可以是立即数，也可以是逻辑运算数，使得 ARM 指令可以在读取数值的同时进行算术或移位操作。
- 对协处理器的支持。在很多以 ARM 为内核的微处理器中，都集成了增强浮点运算功能的协处理器。ARM 内核提供了协处理器接口，通过扩展协处理器可以增加许多新的功能。因此，ARM 指令中还包含许多条协处理器指令。
- Thumb 指令集。ARM 中有两种工作状态决定着它有两种指令集：32 位的 ARM 指令集和

16 位的 Thumb 指令集。Thumb 指令集是重新编码的 ARM 指令集的子集，通常运行于 16 位或低于 16 位的内存数据总线上。使用 16 位的存储器可以降低成本，同时 Thumb 指令集的执行速度比 ARM 32 位指令集要快，而且提高了代码密度。

2. ARM 指令的一般编码格式与语法格式

1）ARM 指令使用固定的 32 位字长，典型的 ARM 指令编码格式如图 3-1 所示。

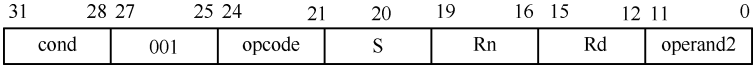

图 3-1　典型的 ARM 指令编码格式

2）ARM 指令的语法格式如下：

```
{label}    <opcode>{<cond>}{S}<Rd>,<Rn>{, <operand2>}{;comment}
```

其中，{}中的内容为可选项，<>中的内容为必选项。以下对具体项目内容进行说明。

{label}：标号代表一个地址，可选项。段内标号的地址值在汇编时确定，段外标号的地址在连接时确定。需注意的是，必须使用标号时不可省略。

<opcode>：指令助记符中的操作码，说明指令完成的功能，必选项。

{<cond>}：说明指令执行的条件。可选项，具体可选字符组合见表 3-1，均由两个字符标识。如果有，指令必须在满足条件时才执行，不满足条件时指令执行空操作（NOP）指令；如果无，则指令是无条件执行的。

{S}：可选项，决定指令的操作是否对 CPSR 的条件标志码产生影响。选中助记符 S，表示指令的执行对条件码产生影响；否则无影响。

注意：有几个特殊的指令不需要助记符 S，也对条件标志码有影响，主要是比较类指令。

<Rd>：必选项，常作为存储指令执行结果的目的寄存器。

<Rn>：必选项，常作为存储指令第 1 操作数的寄存器。

{,<operand2>}：可有可无，取决于 ARM 指令。

{;comment}：可选项，指令注释行。以分号开头，说明指令完成的功能等。

3）应用举例：以下是一小段汇编程序。

```
LDR   R0, [R1]          ;以 R1 的内容为内存地址,读出该单元的数据并送 R0,无条件执行
BEQ   SUBPROG           ;B 是跳转指令,当条件 EQ 满足即标志位 Z=1 时程序跳转到 SUBPROG
ADDS R1, R1, #10        ;ADD 加法指令,实现 R1+10→R1,带有 S,影响条件标志位
SUBNES R1, R1, #0xC     ;SUB 减法指令,当 NE 条件满足即标志位 Z=0 时执行该指令,并影响标志位
```

对第 2 操作数使用的一些说明：它共占 12 位，可以用来存储立即数，具体的形式参见下文所述；它还可以进行寄存器移位的偏移量操作，这时移位的寄存器 Rm 编码占 4 位，移位类型操作码占 3 位，移位常数占 5 位。

在 ARM 指令中为了提高代码的执行效率，可以灵活地使用第 2 操作数，上面的后 2 条指令就使用了第 2 操作数，它们是以立即数的形式出现的。第 2 操作数也可以实现复合寻址方式。以下较为详细地介绍其操作数形式。

（1）立即数形式

用#immed_8r 表示，即数据前使用符号“#”。immed_8r 对应一个 8 位位图的常数，即一个 8 位的二进制常数向右循环移动偶数位得到的结果。具体的移位由 4 位二进制数乘以 2 确定，取值范围是$(0\sim2^4-1)\times2=0\sim30$。由图 3-1 典型的 ARM 编码格式可以看出，第 2 操作立

即数共占用 12 位(b11~b0)，这里的 8 位指的是 b7~b0，4 位指的是 b11~b8。

注意：在写汇编程序时，只要指令中的立即数通过上述运算可以获得就是正确的，程序员不需要考虑立即数在指令代码中的具体存储，由 ARM 汇编编译器确定。还有一点也要特别留心，就是最后形成的是一个 32 位的数据常量。

合法的常数有：0x2FC、0xFF000000、0xF0000001、0x00~0xFF。

非法的常数有：0x1FE、511、0xFFFF、0x10100 和 xF000010。

常数表达式应用举例如下。

```
MOV   R0, #20                  ;立即数 20→R0
AND   R1,   R2, #0xAF          ;R2 逻辑"与"立即数 0xAF→R1
LDR   R0,  [R1],  #4           ;以 R1 内容为地址的存储单元数据→R0,之后 R1←R1+4
```

(2) Rm 寄存器形式

在这种方式下，第 2 操作数就是 Rm 寄存器的内容，具体应用实例如下。

```
SUB R1, R1,   R2               ; R1←R1-R2
MOV PC, R1                     ; PC←R1, 程序跳转到指定地址去执行
LDR   R0,  [R1], -R2           ;以 R1 内容为地址的存储单元数据→R0,之后 R1←R1-R2
```

(3) Rm 寄存器移位形式

Rm 寄存器移位形式需要算术移位或逻辑移位的操作符与常数结合完成，操作完成后，Rm 中的内容不变。移位的方法如下。

```
ASR #n   ;算术右移(Arithmetic Shift Right)n 位,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。要求 1≤n≤31
LSL  #n  ;逻辑左移(Logical Shift Left)n 位,寄存器中字的低端空出的位补 0。要求 1≤n≤31
LSR  #n  ;逻辑右移(Logical Shift Right)n 位,寄存器中字的高端空出的位补 0。要求 1≤n≤31
ROR #n   ;循环右移(Rotate Right)n 位,由字的低端移出的位填入字的高端空出的位。要求 1≤n
         ≤31
RRX #n   ;带扩展的循环右移(Rotate Right eXtended by 1 place),操作数每右移一位,高端空出的位
         用进位位 C 标志值填充
```

注意：当 n 不在规定的取值范围时，它们运算的结果为 0。

寄存器移位形式应用举例如下。

```
ADD   R1, R1, R1, LSL  #3    ; R1←R1 左移 3 位(即×8)+R1
SUB   R0, R0, R1, LSL  #2    ; R0←R0-R1 左移 2 位(即×4)
```

3.1.2 指令执行的条件码

大多数 ARM 指令都是有条件执行的，也就是说根据当前程序状态寄存器（CPSR）中的条件标志位来决定是否执行该指令。另外还必须在指令格式中包含条件助记符。

从指令的编码格式中可以得知，条件码 cond 占 32 位指令的最高 4 位，也就是说它可以组合成 16 种条件码。每种条件码的含义和助记符见表 3-1。

表 3-1　ARM 指令的条件码含义和助记符表

操作码[31:28]	条件码助记符	CPSR 中的标志	含 义
0000	EQ	Z=1	相等
0001	NE	Z=0	不相等

（续）

操作码[31:28]	条件码助记符	CPSR 中的标志	含 义
0010	CS/HS	C=1	无符号数大于或等于
0011	CC/LO	C=0	无符号数小于
0100	MI	N=1	负数
0101	PL	N=0	正数或零
0110	VS	V=1	溢出
0111	VC	V=0	没有溢出
1000	HI	C=1，Z=0	无符号数大于
1001	LS	C=0，Z=1	无符号数小于或等于
1010	GE	N=V	带符号数大于或等于
1011	LT	N!=V	带符号数小于
1100	GT	Z=0，N=V	带符号数大于
1101	LE	Z=1，N!=V	带符号数小于或等于
1110	AL	无条件执行	任何版本
1111	AL	无条件执行	任何（v5 以上版本）

注意：在 ARMv5 之前的版本中，ARM 指令都是有条件执行的，但从 ARMv5 版本以后引入了一些无条件执行的指令。

条件执行指令应用示例如下。

```
CMP R0, R10          ;比较指令,不加 S 但影响标志位。进行 R0-R10 操作,但 R0、R10 内容不变
ADDNE R0, R0, R1     ;如果 NE 为真,则 R0←R0+R1。指令执行不影响标志位
SUBEQS R0, R0, #02   ;如果 EQ 为真,则 R0←R0-2 。指令执行影响标志位
```

3.2 ARM9 微处理器指令的寻址方式与应用

寻址方式是计算机的主要性能特征表现之一。所谓寻址方式就是微处理器在指令码中寻找操作数地址的方式。标准的 ARM 都是 32 位的指令，它们可以有 3 个操作数，其中第 1 个操作数一般为基本操作数寻址方式，第 2、第 3 个操作数可采用复合寻址方式。以下介绍 ARM 微处理器的 9 种寻址方式。

3.2.1 立即数寻址方式与应用示例

立即数寻址即操作数就在指令的代码中。立即寻址指令的操作码字段后面的地址码部分就是操作数本身，取出指令也就取出了可以立即使用的操作数（也称为立即数）。立即数要以“#”为前缀，表示十六进制数值时以“0x”表示。

应用示例：

```
ADD     R0, R0, #1           ;R0←R0+1, 不影响标志位
MOV     R0, #0xff00          ;R0←0xff00,不影响标志位
```

注意：指令中的立即数是用 12 位表示的。前已介绍它是由一个 8 位的常数（图中的 immed_8）乘以一个 4 位二进制数（图中的 rotate_imm）的 2 倍后获得的。即 8 位常数×（0~

15）×2。并不是所有的 32 位都可以作为合法的立即数。具体描述如图 3-2 所示。

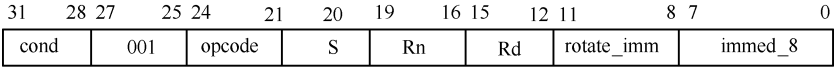

图 3-2　立即数寻址时的 ARM 指令编码格式

书写立即数时，必须以“#”开头。对于不同进制的立即数有不同的书写方式。

对于十六进制数，“#”后加“0x”或“&”，如#0xaf 或 &af。

对于二进制数，“#”后加“0b”或“%”，如#0b10101011 或#%10101011。

对于十进制数，“#”后加“0d”或省略，如#0d123 或 567。

3.2.2　寄存器寻址方式与应用示例

寄存器寻址的操作数就是寄存器的内容。指令中的地址码字段给出的是寄存器编号，寄存器的内容就是操作数，指令执行时直接取出寄存器值操作。

应用示例：

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

3.2.3　寄存器偏移寻址方式与应用示例

寄存器偏移寻址是 ARM 指令集特有的寻址方式。当第 2 个操作数是寄存器偏移方式时，它在与第 1 个操作数结合之前，可以按指令中的操作码进行移位操作。以下用其中的一条指令进行说明。

```
MOV    Rd,Rn,Rm, {<shift>}
```

其中，Rm 被称为第 2 操作数寄存器。

<shift>被用来指定移位类型和移位位数，有 2 种形式：一是 5 位二进制构成的立即数，其值的范围在 0~31 之间；二是使用寄存器的内容，其值的范围也在 0~31 之间。

应用示例：

```
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 左移 R3 位,然后和 R1 相与操作,结果放入 R1
```

1. 第 2 操作数的移位方式

第 2 操作数的移位方式共有 6 种：逻辑左移 LSL、逻辑右移 LSR、算术左移 ASL、算术右移 ASR、循环右移 ROR、带扩展的循环右移 RRX。它们的操作示意图如下。

逻辑左移 LSL：向左移位时低端空出位补 0，在不溢出情况下等价于乘 2。如图 3-3 所示。

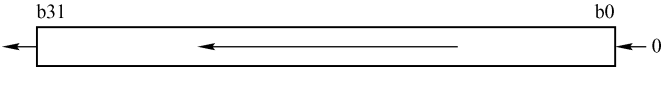

图 3-3　逻辑左移 LSL 移位示意图

逻辑右移 LSR：向右移位时高端空出的位补 0，等价于整除 2，舍去余数。如图 3-4 所示。

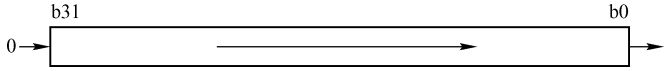

图 3-4　逻辑右移 LSR 移位示意图

应用示例：

```
SUB R3, R2, R1,   LSL #3          ; R3←R2-(R1 逻辑左移 3 位)
SUB R3, R2, R1,   LSR R0          ; R3←R2-(R1 逻辑右移 R0 位)
```

算术左移 ASL：向左移位时低端空出的位补 0，最高符号位保持不变。如图 3-5 所示。

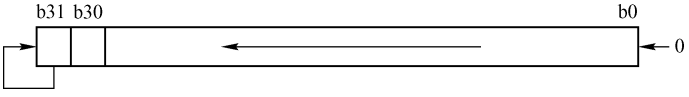

图 3-5　算术左移 ASL 移位示意图

算术右移 ASR：向右移位时若为正数，最高位为 0，移出高端空出位补 0；若为负数，最高位为 1，移出高端空出位补 1；算术右移 ASR 等价于整除 2，舍去余数。如图 3-6 所示。

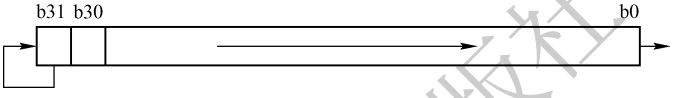

图 3-6　算术右移 ASR 移位示意图

应用示例：

```
ADD   R3, R2, R1, ASL #2          ; R3←R2+(R1 算术左移 2 位)
SUB   R3, R2, R1, ASR R0          ; R3←R2-(R1 算术右移 R0 位)
```

循环右移 ROR：由字的 b0 输出，进入字的 b31，依次进行。如图 3-7 所示。

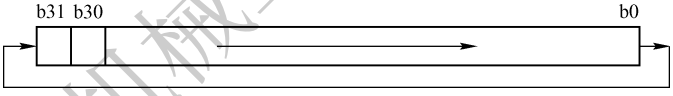

图 3-7　循环右移 ROR 示意图

带扩展的循环右移 RRX：就是带进位循环右移操作一位，高端空出位使用进位位 C 填充。如图 3-8 所示。

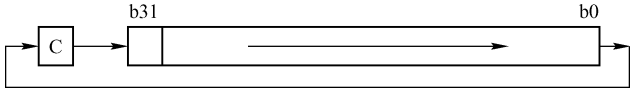

图 3-8　带扩展循环右移 RRX 示意图

应用示例：

```
SUB   R3, R2, R1, ROR #2          ; R3←R2-(R1 循环右移 2 位)
SUB   R3, R2, R1, RRX #0x04       ; R3←R2-(R1 带进位循环右移 4 位)
```

2. 第 2 操作数的移位位数

移位位数可以用立即数表示或由寄存器方式给出，其值的范围在 0~31 之间。上述的许多应用示例使用的是立即数或由寄存器给出，但是一定要注意，给定的数值必须在指定的范围之内。

3.2.4 寄存器间接寻址方式与应用示例

寄存器间接寻址就是将寄存器的内容作为操作数的地址。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，操作数存放在存储器中。

应用示例：

```
LDR   R0,[R1]    ;R0←[R1],此处的方括号是寄存器间接寻址的意思
STR   R0,[R1]    ;[R1]←R0,将 R0 的内容送入 R1 的内容作为地址的内存单元中
```

3.2.5 基址+变址寻址方式与应用示例

基址+变址寻址方式（[Rn,偏移量]）{!}也叫变址寻址方式，它是将基址寄存器 Rn 的内容与指令中给出的地址偏移量相加，形成操作数的有效地址。若使用后缀“!”，则有效地址最后写回 Rn 中，称为自动修改指针，且 Rn 不允许使用 R15。变址寻址方式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。

变址寻址方式分为 3 种，即前变址模式、自动变址模式和后变址模式；偏移量有立即数偏移量、寄存器偏移量和寄存器移位偏移量 3 种形式。ARM 指令中使用的是它们的组合体，这样前变址寻址模式[Rn,偏移量]就包括以下几种形式。自动变址模式不再单独介绍，同其他两种模式一并介绍。以下通过应用示例进行介绍。

1. 立即数偏移量应用示例

```
LDR   R2, [R3, #4]          ;R2←[R3 + 4],R3 的内容不变
LDR   R2, [R3, #4]!         ;R2←[R3 + 4],R3←R3+4
```

2. 寄存器偏移量应用示例

```
STR   R1,[R0, R2]           ;[R0+R2] ← R1。R0 作为基址,内容不变
```

如果在这对双括号后加上“!”，则完成后 R0←R0+R2。

3. 寄存器移位偏移量应用示例

```
LDR   R0,[R1, R2, LSL  #3]      ; R0←[(R1)+(R2)*8]
```

该条指令的功能是将基址寄存器 R1 的内容加上 R2 的内容乘 8 作为有效地址的存储单元内容传送到 R0 寄存器中，R1 的内容保持不变。方括号后若有“!”，则 R1 地址指针自动修改。

注意：对于后变址偏移寻址模式（[Rn]，偏移量），Rn 的值用作传送数据的存储器地址。在操作完数据后，Rn+偏移量送到 Rn，即修改了 Rn 中的地址指针。同样，后变址模式也有立即数寻址方式、寄存器寻址方式和寄存器移位寻址方式 3 种情况，且 Rn 不允许使用 R15 寄存器。

注意：后变址模式不需要加“!”就可修改地址指针。以下列举几个应用示例。

```
LDR   R0,[R1], #4           ; R0←[R1];R1← R1+4
```

此指令是将以基址寄存器 R1 内容作为有效地址单元的存储器内容加载到 R0 中，之后修改 R1 的内容，即 R1 内容+4 送 R1。

```
STR   R0,[R3], -R8          ;[R3] ←R0; R3←R3-R8
```

此指令是将 R0 的内容写到以 R3 内容作为有效地址的内存单元中，之后 R3 内容减去 R8

内容送 R3 中。其他组合这里不再赘述，后面会有更多的应用示例。

3.2.6 多寄存器寻址方式与应用示例

多寄存器寻址是 ARM 微处理器独有的寻址方式。多寄存器寻址方式就是一条指令可以完成多个寄存器值的传送，这种寻址方式用一条指令最多可以完成 16 个寄存器值的传送。

应用示例：

```
LDMIA   R0, {R1, R2, R3, R5}     ;R1←[R0]; R2←[R0 + 4] ;R3←[R0 + 8];R5←[R0 + 12]
```

该条指令以 R0 的内容作为存储器有效基地址，取出内容送入 R1，R0+4 为地址单元的内容送 R2；R0+8 为地址单元的内容送 R3；R0+12 为地址单元的内容送 R5。操作完成后 R0 的内容不变。

若要实时改变 R0 的内容，则执行的指令是：LDMIA　R0!，{R1，R2，R3，R5}，这条指令的执行，R1、R2、R3、R5 的内容同前，但最后 R0 的内容等于 R0+12。

注意：花括号中是 16 个寄存器 R0~R15 的子集，寄存器的编号从小到大排列，使用“,”隔开，编号连续时可以使用“-”连接，例如：

```
LDMIA   R1!, {R2-R9, R12}
STMIA   R0!, {R3-R8, R10}
```

第 1 条指令将 R1 的内容作为有效字存储单元首地址，字内容分别送入 R2~R9、R12，最后 R1 的内容是 R1+4×8。

第 2 条指令 R3~R8、R10 这 7 个寄存器的内容保存到以 R0 的初值为首地址的字存储单元中，最后 R0 的内容是 R0+4×6。

注意：指令中的后缀 IA（Increment After）为操作模式，意思是传送数据之后，再增加地址指针，即传送完成后地址加 4。另外还有如下后缀。

IB（Increment Before）：指传送前地址先加 4。

DA（Decrement After）：指传送后地址减 4。

DB（Decrement Before）：指传送前地址先减 4。

3.2.7 堆栈寻址方式与应用示例

堆栈是一种数据结构，堆栈是按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”，堆栈寻址是隐含的，它使用一个专门的寄存器（堆栈指针寄存器 R13）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。

1. 存储器堆栈的相关概念

- 递增堆栈：堆栈区由低地址向高地址方向生长，称为递增堆栈（Ascending Stack）。
- 递减堆栈：堆栈区由高地址向低地址方向生长，称为递减堆栈（Descending Stack）。
- 满堆栈：堆栈指针指向最后压入堆栈的有效数据项，称为满堆栈（Full Stack）。对于满堆栈的操作，在进行压栈时要先修改堆栈指针，再压入数据；在弹栈时要先弹出数据，再修改指针。否则，就会发生错误。
- 空堆栈：堆栈指针指向下一个要放入的空位置，称为空堆栈（Empty Stack）。对于空堆栈的操作，在进行压栈时要先压入数据，再修改堆栈指针；在弹栈时要先修改指针，再弹出数据。否则，就会发生错误。

2. 堆栈的四种工作方式

- 满递增堆栈（Full & Ascending stack，FA）：堆栈指针指向最后压入的数据，且栈区由低地址向高地址生成。如指令 LDMFA、STMFA 等。
- 满递减堆栈（Full & Descending stack，FD）：堆栈指针指向最后压入的数据，且栈区由高地址向低地址生成。如指令 LDMFD、STMFD 等。
- 空递增堆栈（Empty & Ascending stack，EA）：堆栈指针指向下一个将要放入数据的空位置，且栈区由低地址向高地址生成。如指令 LDMEA、STMEA 等。
- 空递减堆栈（Empty & Descending stack，ED）：堆栈指针指向下一个将要放入数据的空位置，且栈区由高地址向低地址生成。如指令 LDMED、STMED 等。

注意：上述每种堆栈工作方式的压栈指令和弹栈指令要成对使用，千万不能配错；压栈和弹栈的多寄存器列表必须一一对应，而且排列次序也是寄存器下标从小到大。需要说明的是，按照堆栈操作的原则，正确的方法是“先进后出，后进先出”，那么弹栈的寄存器列表顺序应该从大到小，实际书写编辑时仍然是从小到大的顺序，程序员不必担心，指令在操作时会自动地按照堆栈区的操作原则进行。

应用举例：

```
STMFD   SP!,{R4-R7,LR}      ;将 R4~R7、链接寄存器 LR 内容压栈(内存),属于满栈递减
LDMFD   SP!,{R4-R7,PC}      ;从栈区弹栈内容送 R4~R7、程序计数器 PC,属于满栈递减
```

3.2.8 块复制寻址方式与应用示例

块复制寻址方式就是把一块从存储器的某一位置开始的数据复制到多个寄存器中，或者把多个寄存器的内容复制到存储器的某一块中。它是多地址多寄存器寻址方式的一种应用，实际上要完成的是从存储器的某一块开始的数据复制到存储器的另外一块中去，这里中间的过渡缓冲区就是这多个寄存器列表。

它与堆栈的操作基本相同，也有 4 组配对的操作模式指令。

LDMIA/STMIA：在传送数据之后增加地址指针，块中的首地址值最小。

LDMIB/STMIB：在传送数据之前增加地址指针，块中的首地址值最小。

LDMDA/STMDA：在传送数据之后减小地址指针，块中的首地址值最大。

LDMDB/STMDB：在传送数据之前减小地址指针，块中的首地址值最大。

应用示例：

```
LDMIA    R0!,  {R2-R12}
STMIA    R1!,  {R2-R12}
```

第 1 条指令是将以 R0 为首地址的字单元存储器的内容加载到 R2~R12 中。存储器指针 R0 在每加载一个值之后增加，增加步长为 4，增长方向为向上增长。R0 的内容自动修改。

第 2 条指令是将 R2~R12 的数据保存到以 R1 初值为首地址的字存储单元中，存储器指针 R1 每保存一个值之后增加，增加步长为 4，增长方向也为向上增长。R1 的内容自动修改。

这 2 条指令结合在一起实现的功能就是将以 R0 为首地址的字单元存储器的内容复制到以 R1 为首地址的字存储单元块中，共 11×4=44 字节数据。

注意：寄存器间接寻址方式、多寄存器寻址方式、堆栈寻址方式和块复制寻址方式 4 种指令格式的差异和完成操作的特点说明如下：

1）寻址方式都是寄存器间接寻址，但是寄存器间接寻址方式中的寄存器需要方括号[ ]，而后 3 种不需要。而且其后的偏移量范围很大，可正可负，只要与基址累加不超出 ARM 存储器的地址范围就行。

2）多寄存器寻址方式中的基址寄存器不需要方括号[ ]，且根据“!”的有无决定指针是否自动修改。有则改；无则不改。修改值由指令的后缀模式来决定，模式 IA、IB 时地址指针+4；模式 DA、DB 时地址指针-4，上下偏移量只有±4。

3）堆栈寻址方式和块复制寻址方式实际上是多寄存器寻址方式的特殊应用，它们的基址寄存器也不需要使用方括号[ ]，修改的偏移量只有±4，并且正负号也由指令码后缀模式（FA、EA、FD、ED）确定。FA、EA 为+4；FD、ED 为-4。堆栈寻址方式指向存储器（栈区）的寄存器必须是 SP（R13)!。“!”必须有是因为压栈与弹栈必须有机配合才能保证堆栈操作的正确使用。

块复制寻址方式与堆栈寻址方式基本相同，差异就在将指令格式中的 SP! 修改为 Rn!，n=1，2，…，11，12。**需要注意的是，**如果基址寄存器使用了哪个寄存器，就不要出现在指令的花括号{}中。

3.2.9 相对寻址方式与应用示例

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

应用示例：

```
        BL   ROUTINE1                ;调用子程序 ROUTINE1
        BEQ   LOOP                   ;条件跳转到 LOOP 标号处执行,不返回
        …
LOOP    MOV R2,#2
        …
ROUTINE1
        …
    ;语句 BL   ROUTINE1 是调用 ROUTINE1 子程序,ROUTINE1 实际上是子程序标号,是一个相对于 PC 指针的值。执行完子程序后返回
```


3.3 ARM9 指令系统与应用

ARM 指令集可以分为 ARM 数据处理指令、寄存器装载及存储指令、ARM 跳转指令、ARM 杂项指令、ARM 协处理器指令和 ARM 伪指令。

3.3.1 ARM 数据处理指令与应用示例

数据处理指令大致可分为 3 类：数据传送指令、算术逻辑运算指令、比较指令和测试指令。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。

数据传送指令：只用于在寄存器与寄存器之间进行数据的双向传输。

比较指令和测试指令：不需要后缀 S，它们会直接影响 CPSR 的条件标志位，并且比较指令不保存运算结果，只起到更新 CPSR 中相应条件标志位的作用。

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

乘法指令：其操作数全部是寄存器。

凡是具有第 2 操作数（operand2）的指令（乘法指令除外），均有 3 种使用形式：立即数形式、寄存器形式和寄存器移位形式。

ARM 微处理器的主要数据处理指令见表 3-2，以下将分类详细地介绍它们。

1. 算术运算指令与应用示例

（1）ADD 加法运算指令

加法运算指令 ADD。将 Rn 的数值与 operand2 的数值相加，结果保存到 Rd 寄存器中。指令格式如下：

```
ADD{cond}{S} <Rd>,<Rn>,<operand2>
```

应用示例：

```
ADDS   R1,R1,#0x03          ;R1←R1+0x03,影响条件标志位
ADD    R0,R2,R4             ;R0←R2+R4,不影响条件标志位
ADDS   R1,R2,R3,LSL #3      ;R1←R2+R3×8,影响条件标志位
```

表 3-2　ARM 微处理器的主要数据处理指令

操作码［24:21］	助记符	功　能	完成的操作
0000	AND	逻辑位“与”	Rd←Rn AND Op2
0001	EOR	逻辑位“异或”	Rd←Rn EOR Op2
0010	SUB	算术减法	Rd←Rn-Op2
0011	RSB	算术反向减法	Rd←Op2-Rn
0100	ADD	算术加法	Rd←Rn+Op2
0101	ADC	带进位算术加法	Rd←Rn+Op2+C
0110	SBC	带进位算术减法	Rd←Rn-Op2-(not)C
0111	RSC	带进位反向算术减法	Rd←Op2-Rn-(not)C
1000	TST	按位测试	根据 Rn AND Op2 设置条件标志位
1001	TEQ	按位相等测试	根据 Rn EOR Op2 设置条件标志位
1010	CMP	比较	根据 Rn-Op2 设置条件标志位
1011	CMN	负数比较	根据 Rn+Op2 设置条件标志位
1100	ORR	逻辑位“或”	Rd←Rn OR Op2
1101	MOV	传送	Rd←Op2
1110	BIC	按位清 0	Rd←Rn　AND(not)Op2
1111	MVN	按位求反	Rd←(not)Op2

（2）ADC 带进位加法运算指令

ADC 指令是将 Rn 的数值与 operand2 的数值相加，再加上 CPSR 中的进位标志 C，结果保存到 Rd 寄存器中。指令格式如下：

```
ADC{cond}{S} <Rd>,<Rn>,<operand2>
```

应用示例：使用 ADC 实现 64 位的 2 个二进制数相加，即(R1,R0)=(R5,R4)+(R3,R2)。

```
ADDS   R0,R2,R4             ;R0←R2+R4,低 32 位相加,影响条件标志 C 位
ADC    R1,R3,R5             ;R1←R3+R5+C,高 32 位数相加,不影响条件标志位
```

（3）SUB 减法指令

减法运算指令 SUB，是用 Rn 的数值减去 operand2 的数值，结果保存到 Rd 寄存器中。指令格式如下：

```
SUB{cond}{S}<Rd>,<Rn>,<operand2>
```

应用示例：

```
SUBS   R10,R10,#2              ;R10←R10-2,影响条件标志位
SUBS   R0,R2,R0                ;R0←R2-R0,影响条件标志位
SUB    R1,R2,R3,LSL #0x04      ;R1←R2-R3×16,不影响条件标志位
```

（4）带借位减法运算 SBC 指令

SBC 指令是用 Rn 的数值减去 operand2 的数值，再减去 CPSR 中的进位标志 C，结果保存到 Rd 寄存器中。

注意：虽然在 ARM 中有借位时 C=0，无借位时 C=1，但是使用该条指令时，程序员不必考虑这些，由 ARM 系统自动完成。指令格式如下：

```
SBC{cond}{S}<Rd>,<Rn>,<operand2>
```

应用示例：使用 SBC 实现 64 位的 2 个二进制数相减，即(R7,R6)=(R5,R4)-(R3,R2)。

```
SUBS   R6,R4,R2                ;R6←R4-R2,低 32 位相减,影响条件标志位
SBC    R7,R5,R3                ;R7←R5-R3,高 32 位数相减,不需影响条件标志位
```

（5）RSB 反向减法运算指令

反向减法运算指令 RSB，是用 operand2 的数值减去 Rn 的数值，结果保存到 Rd 寄存器中。指令格式如下：

```
RSB{cond}{S}<Rd>,<Rn>,<operand2>
```

应用示例：

```
RSB    R3,R1,#0xFF00           ;R3←0xFF00-R1,不影响条件标志位
RSBS   R3,R2,R1                ;R3←R1-R2,影响条件标志位
RSBS   R1,R2,R3,LSL #0x02      ;R1←R3×4-R2,影响条件标志位
```

（6）RSC 反向带进位减法运算指令

RSC 指令是用 operand2 的数值减去 Rn 的数值，再减去 CPSR 中的进位标志 C，结果保存到 Rd 寄存器中。指令格式如下：

```
RSC{cond}{S}<Rd>,<Rn>,<operand2>
```

应用示例：使用 RSC 实现 64 位的 2 个二进制数的负数，源数（R5，R4），结果存（R3，R2）。

```
RSCS   R2,R4,#0                ;R2←0-R4,低 32 位相减,影响条件标志位
RSC    R3,R5,#0                ;R3←0-R5,高 32 位数相减,不影响条件标志位
```

在 ARM 中还有 6 条乘法和乘加指令，运算结果分为 32 位和 64 位两类，指令中的所有操作数只能使用通用寄存器，同时寄存器和操作数 1 必须使用不同的寄存器。

（7）32 位乘法指令 MUL

32 位乘法指令 MUL 是将 Rm 的值与 Rs 中的值相乘，结果的低 32 位保存在 Rd 中。指令的格式如下：

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

应用示例：

```
MUL R1,R2,R3              ;R1←R2×R3
MULS R1,R3,R5             ;R1←R3×R5,影响条件标志位
```

（8）32 位乘加指令 MLA

32 位乘加指令 MLA 是将 Rm 的值与 Rs 中的值相乘，再加上 Rn 的值，结果的低 32 位保存在 Rd 中。指令的格式如下：

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

应用示例：

```
MLA   R1,R2,R3,R4         ;R1←R2×R3+R4
MLAS  R1,R3,R5,R7         ;R1←R3×R5+R7,影响条件标志位
```

（9）64 位有符号乘法指令 SMULL

64 位有符号乘法指令 SMULL 是将 Rm 的值与 Rs 中的值相乘，结果的低 32 位保存在 RdLo 寄存器中，高 32 位保存在 RdHi 寄存器。指令的格式如下：

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

应用示例：

```
SMULL R1,R2,R3,R4          ;R1←(R3×R4)低 32 位;R2←(R3×R4)高 32 位
```

（10）64 位有符号数乘加指令 SMLAL

SMLAL 指令是将 Rm 的值与 Rs 的值相乘，其乘积低 32 位与寄存器 RdLo 的值相加同时影响进位标志，再回送给寄存器 RdLo；而高 32 位与寄存器 RdHi 的值相加，并加上低 32 位的进位 C，再回送给寄存器 RdHi。指令的格式如下：

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

应用示例：

```
SMLAL R2,R3,R7,R6      ;R2←(R7×R6)低 32 位+R2 ; R3←(R7×R6) 高 32 位+R3
```

（11）**64 位无符号数乘法指令 UMULL**

64 位无符号乘法指令 UMULL 是将 Rm 的值与 Rs 中的值相乘，结果的低 32 位保存在 RdLo 寄存器中，高 32 位保存在 RdHi 寄存器中。指令的格式如下：

```
UMULL   <RdLo>,<RdHi>,<Rm>,<Rs>
```

应用示例：

```
UMULL   R0,R1,R2,R3       ;R0←(R2×R3)低 32 位;R1←(R2×R3)高 32 位
```

（12）64 位无符号数乘加指令 UMLAL

UMLAL 指令是将 Rm 的值与 Rs 的值相乘，其乘积低 32 位与寄存器 RdLo 的值相加，同时影响进位标志，再回送给寄存器 RdLo；而高 32 位 RdHi 寄存器中的值是 RdHi 的寄存器值加上低 32 位相加的进位位 C。指令的格式如下：

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

应用示例：

```
UMLAL R2,R3,R4,R5          ;R2←(R4×R5)低 32 位+R2,R3←(R4×R5)高 32 位+R3
```

2. 逻辑运算指令与应用示例

（1）AND 指令

AND 逻辑“位与”操作指令，将 Rn 的值与 operand2 的值按位进行逻辑“与”操作，并将结果保存到 Rd 中。指令的格式如下：

```
AND{cond}{S}<Rd>,<Rn>,<operand2>
```

应用示例：

```
AND    R0,R0,0x01    ;R0←R0 & 0x01。“&”是 C 语言中的“位与”操作符。功能是保留最低位
ANDS   R0,R1,R2      ;R0←R1 & R2,影响条件标志位。如果逻辑运算的结果为 0,则 Z=1
```

（2）ORR 指令

ORR 逻辑“位或”操作指令，将 Rn 的值与 operand2 的值按位进行逻辑“或”操作，并将结果保存到 Rd 中。指令的格式如下：

```
ORR{cond}{S}<Rd>,<Rn>,<operand2>
```

应用示例：

```
ORR    R0,R0,0x0F    ;R0←R0|0x0F,将 R0 的低 4 位置“1”
ORR    R0,R1,R2      ;R0←R1|R2,“|”是 C 语言中的“位或”操作符
```

（3）EOR 指令

EOR 逻辑“位异或”操作指令，将 Rn 的值与 operand2 的值按位进行逻辑“异或”操作，并将结果保存到 Rd 中。指令的格式如下：

```
EOR{cond}{S}<Rd>,<Rn>,<operand2>
```

应用示例：

```
EOR    R0,R0,0xF0    ;将 R0 的 b7~b3 位取反
EORS   R0,R1,R2      ;R0←R1⊕R2,影响条件标志位。如果 R1=R2,则 Z=1
```

（4）BIC 指令

位清除指令 BIC 是将 Rn 的值与 operand2 的值按位取反后，进行逻辑“与”操作，并将结果保存到 Rd 中。指令的格式如下：

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

应用示例：

```
BIC    R0,R0,0x0F          ;R0←R0 AND (~0x0F),将 R0 的低 4 位清“0”
BICS   R2,R5,0xFFFF        ;R0←R5 AND (~(0xFFFF)),最后 R2 的结果是低 16 bit 均为 0
```

3. 数据传送指令与应用示例

（1）MOV 指令

数据传送指令 MOV 是将 operand2 操作数传送到目的寄存器 Rd 中。operand2 操作数可以

是立即数、寄存器和寄存器移位。在使用立即数时，并非所有的 32 位立即数都可以使用，具体论述请参照 3.1 节所述。

注意：在 ARM 中为了实现将任意的 32 位立即数传送到寄存器，设计了 ARM 伪指令。ARM 伪指令不同于汇编器中的伪指令，它是有机器码的，具体见 3.3.6 节。而汇编器中的伪指令只是在汇编器中使用，不产生机器码。

MOV 指令的格式如下：

```
MOV{cond}{S} <Rd>,<operand2>
```

应用示例：

```
MOV   R1,#0x100          ;R1←0x100,完成立即数送 R1
MOVS  R2,R1              ;R2←R1,完成寄存器之间传送。影响条件标志位,R1=0x0 时,Z=1
MOV   R3,R4,LSR #0x2     ;R3←R4÷4,完成寄存器右移位的传送
MOV   PC,LR              ;PC←LR 实现子程序返回
```

注意：在 ARM 中没有设计专门的子程序返回指令和异常（特别是 IRQ 和 FIQ 中断服务程序）返回指令，而是使用 MOV 指令或其他方式完成从子程序返回和中断服务程序返回。只要向程序计数器 PC 赋值后，就会跳转到相应的地址处执行程序。

（2）MVN 指令

数据取反传送指令 MVN 是将 operand2 操作数按位取反后，传送到目的寄存器 Rd 中。指令格式如下：

```
MVN{cond}{S} <Rd>,<operand2>
```

应用示例：

```
MVN   R1,#0xFF      ;R1=0xFFFFFF00
MVN   R2,R3         ;R3 按位取反送 R2
```

4. 比较指令与应用示例

（1）CMP 指令

比较指令 CMP 是用 Rn 的值减去 operand2 操作数，操作的结果影响 CPSR 中的相应条件标志位，以便其后的指令根据其条件判断是否执行。这里要说明的是，该指令并不改变其两个操作数的内容。指令格式如下：

```
CMP{cond} <Rn>,<operand2>   ;注意这里没有{S}选项,但是也影响条件标志位
```

应用示例：

```
CMP     R0,#20              ;R0-20,影响条件标志位
ADDEQ   R3,R2,R1            ;如果 R0=20,则 Z=1,EQ 为真,执行该指令,R3←R2+R1
```

（2）CMN 指令

负数比较指令 CMN 是将 Rn 的值加上 operand2 操作数的值，根据操作的结果影响 CPSR 中的相应条件标志位，以便其后的指令根据其条件判断是否执行。该指令并不改变其两个操作数的内容。指令格式如下：

```
CMN{cond} <Rn>,<operand2>   ;注意这里没有{S}选项,但是也影响条件标志位
```

使用方法：Rn 中存放的是欲比较的负数，并且以补码的形式表示，operand2 是另一个要

比较的数。

应用示例：

```
CMN     R0,#1          ;影响条件标志位
MOVEQ   R3,R2          ;如果 R0 是-1 的补码,则 Z=1,EQ 为真,执行该指令,R3←R2
```

注意：-1 的补码就是 0xFFFFFFFF，与 1 相加自然就等于 0，所以 Z=1，满足 MOV 指令的执行条件。

推论：假如立即数 1 的位置用正数 N 替换，那么，如果 R0 是-N 的补码，则标志位 Z=1。

5. 测试指令与应用示例

（1）TST 指令

TST 是位测试指令，是将 Rn 的值与 operand2 操作数的值按位做逻辑“与”操作，根据操作的结果影响 CPSR 中的相应条件标志位，以便其后的指令根据其条件判断是否执行。这里要说明的是，该指令并不改变其两个操作数的内容。指令格式如下：

```
TST{cond}  <Rn>,<operand2>   ;注意这里没有{S}选项,但是也影响条件标志位
```

使用方法：该指令主要用于判断 Rn 中的某一比特位或多个比特位的位值是否为 0，只要将操作数 operand2 的值对应的位取“1”，然后组成一个 32 位立即数。如果指令作用的结果使得条件标志位 Z=1，说明对应取值为“1”的比特位全为“0”，达到了检测的目的。

应用示例：

```
TST     R0,#0x01           ;影响条件标志位。如果 R0[0]=0,则结果标志位 Z=1
SUBEQ   R3,R2,R1           ;Z=1,说明 EQ 为真,执行该指令,R3←R2-R1
```

（2）TEQ 指令

TEQ 是测试相等指令，是将 Rn 的值与 operand2 操作数的值按位进行逻辑“异或”操作，根据操作的结果影响 CPSR 中的相应条件标志位，以便其后的指令根据其条件判断是否执行。该指令并不改变其两个操作数的内容。指令格式如下：

```
TEQ{cond}  <Rn>,<operand2>   ;注意无{S}选项,但是也影响条件标志位
```

使用方法：该指令主要用于测试 Rn 中的值是否与 operand2 操作数的值相等，通过按位进行逻辑“异或”运算，如果两者的所有对应比特位都相同，“异或”结果为全“0”，则条件标志位 Z=1，说明两者相等；否则，说明两者不相等。

应用示例：

```
TEQ   R0,R1      ;影响条件标志位,测试 R0 是否等于 R1。如果相等,则结果标志位 Z=1
```

3.3.2 寄存器装载及存储指令与应用示例

ARM 微处理器系统对于存储器的操作只能使用寄存器装载和存储指令。基本的装载/存储指令仅有 5 条，其他的指令都是由它们派生出来的，可将其分为 3 种，分别是 LDR 和 STR 指令，称为单寄存器装载/存储指令；LDM 和 STM 指令，称为批量装载/存储指令；SWP 指令，称为寄存器与寄存器数据交换指令。

LDR 和 STR 指令派生的指令最多，可以进行字节操作、半字操作和字操作。LDR 指令的功能是将存储器中的内容装载到单个寄存器中去；STR 指令的功能是从单个寄存器向内存写数据。

LDM 和 STM 指令只能进行字的操作，它们派生的指令一类是对存储器块的操作，另一类是对堆栈区数据块的操作。LDM 指令的功能是将存储器或堆栈区中的块数据装载到 N 个寄存器中；STM 的作用是将 N 个寄存器的内容写入块存储器或块堆栈区中。

SWP 指令有 2 种形式：SWP 和 SWPB。

表 3-3 列出了寄存器装载和存储指令，以下对其进行较为详细的介绍。

表 3-3　寄存器装载和存储指令表

指令助记符	功　能	完成的操作	条件码位置
LDR　Rd,<addr>	装载字数据	Rd←[addr]	LDR{cond}
LDRB　Rd,<addr>	装载字节数据	Rd←[addr]	LDR{cond}B
LDRT　Rd,<addr>	以用户模式装载字数据	Rd←[addr]	LDR{cond}T
LDRBT Rd,<addr>	以用户模式装载字节数据	Rd←[addr]	LDR{cond}BT
LDRH　Rd,<addr>	装载半字数据	Rd←[addr]	LDR{cond}H
LDRSB　Rd,<addr>	装载有符号字节数据	Rd←[addr]	LDR{cond}SB
LDRSH　Rd,<addr>	装载有符号半字数据	Rd←[addr]	LDR{cond}SH
STR Rd,<addr>	存储字数据	[addr]←Rd	STR{cond}
STRB　Rd,<addr>	存储字节数据	[addr]←Rd	STR{cond}B
STRT　Rd,<addr>	以用户模式存储字数据	[addr]←Rd	STR{cond}T
STRBT　Rd,<addr>	以用户模式存储字节数据	[addr]←Rd	STR{cond}BT
STRH　Rd,<addr>	存储半字数据	[addr]←Rd	STR{cond}H
LDM{mode} Rn{!},reglist	块数据装载到列表寄存器中	reglist←[Rn…]	LDM{cond}{mode}
STM{mode} Rn{!},reglist	存储列表数据到存储器块	[Rn…]←reglist	STM{cond}{mode}
SWP　Rd,Rm,[Rn]	寄存器与存储器字交换	Rd←[Rn], [Rn]←Rm	SWP{cond}
SWPB　Rd,Rm,[Rn]	寄存器与存储器字节交换	Rd←[Rn], [Rn]←Rm	SWP{cond}B

以下对表 3-3 中的部分内容进行说明。

<addr>：代表的是存储器中的地址单元，它可以由寄存器或寄存器+偏移量，或寄存器+寄存器移位偏移量组成，在 3.2 节 ARM 寻址方式中已经介绍。

{mode}：如果对存储器进行块操作，则模式 mode 应是 IA、IB、DA、DB 其中之一；如果对堆栈区进行块操作，则模式 mode 应是 FA、FD、EA、ED 其中之一。

“以用户模式”：指在特权模式下可以以用户的身份操作用户寄存器组中的寄存器。在用户模式下，后缀带 T 的指令无效。

1. LDR 和 STR 指令

寄存器装载和存储指令 LDR/STR 可分为按字操作指令、按半字操作指令、按字节操作指令，以下简要介绍它们的格式与功能以及寻址方式等。

（1）LDR/STR 的指令格式与功能

以字方式操作的指令格式与功能：

```
LDR{cond}{T}    Rd,<addr>    ;将存储器地址为 addr 的内容装载到寄存器 Rd 中
STR{cond}{T}    Rd,<addr>    ;将寄存器 Rd 的内容写入存储器地址 addr 单元中
```

以半字方式操作的指令格式与功能：

```
LDR{cond}H    Rd,<addr>   ;将存储器地址 addr 的无符号数半字装载到 Rd 中,高 16 位补 0
LDR{cond}SH   Rd,<addr>   ;将存储器地址 addr 的有符号数半字装载到 Rd 中,高 16 位用其符号位
                            填充
STR{cond}H    Rd,<addr>   ;将寄存器 Rd 的半字数据写入存储器地址 addr 半字单元中
```

注意：半字操作时，地址值必须为偶数，即按半字对齐。非半字对齐的操作地址不可靠。

以字节操作的指令格式与功能：

```
LDR{cond}B{T}Rd,<addr>   ;将内存地址 addr 中的无符号字节数据装载到 Rd,高 24 位用 0 补充
LDR{cond}SB   Rd,<addr>  ;将内存地址 addr 中的有符号字节数装载到 Rd,高 24 位用其符号位
                           填充
STR{cond}B{T}Rd,<addr>   ;将寄存器 Rd 内容低 8 位字节数据写入内存地址 addr 的字节单元中
```

（2）LDR/STR 的指令寻址

LDR/STR 的指令寻址方式非常灵活，由两部分组成，一部分是基址寄存器，可以使用任意一个通用寄存器；另一部分是基址偏移量。它有 3 种形式，以下进行简要介绍。

1）**立即数形式。**立即数用一个无符号数表示，它既可以与基址寄存器 Rn 相加，也可以与基址寄存器相减，从而形成一个有效的地址存储器操作地址。例如：

```
LDR   Rd,[Rn,#0x08]   ;将 Rn+0x08 地址单元中的内容读出,装载到 Rd 寄存器,Rn 的内容不变
LDR   Rd,[Rn,#-0x08]  ;将 Rn-0x08 地址单元中的内容读出,装载到 Rd 寄存器,Rn 的内容不变
LDR   Rd,[Rn]         ;将 Rn 地址单元中的内容读出,装载到 Rd 寄存器,0 偏移
```

2）**寄存器形式。**即用寄存器的内容作为偏移量，与基址寄存器 Rn 的内容相加或相减，从而形成一个有效地址作为存储器操作地址。例如：

```
LDR   Rd,[Rn,Rm]      ;将 Rn+Rm 地址单元中的内容读出,装载到 Rd 寄存器,Rn 的内容不变
LDR   Rd,[Rn,-Rm]     ;将 Rn-Rm 地址单元中的内容读出,装载到 Rd 寄存器,Rn 的内容不变
```

3）**寄存器移位偏移量形式。**即将寄存器 Rm 的内容经过移位操作后，与 Rn 的内容相加，从而形成一个有效地址作为存储器操作地址。移位的方法在 3.1.1 节已经介绍，主要有逻辑左移 LSL、逻辑右移 LSR、算术右移 ASR、循环右移 ROR 和带扩展的循环右移 RRX。例如：

```
LDR   Rd,[Rn,Rm,LSL #3]   ;将 Rn+Rm * 8 的内容作为存储器地址,取其内容装载到 Rd
```

小结：在 3.1.1 节已经介绍过指向存储器地址指针的修改，即 Rn 值的变化可分为前变址模式、自动变址模式和后变址模式。结合寄存器装载/存储指令可对存储器中字节数据、半字数据和字数据进行操作。还有就是对它们可以进行基址寄存器 Rn 加上立即数偏移量、寄存器偏移量和寄存器移位偏移量的操作，这样可以组合成许许多多的具体指令操作。以下列举一些指令，帮助读者进一步掌握各种指令。

```
LDR   R0,[R1]         ;将存储器地址为 R1 内容的字数据→R0,零偏移,寄存器间址寻址
LDR   R0,[R1,R2]      ;将存储器地址为 R1+R2 内容的字数据→R0,属于前变址模式
LDR   R0,[R1,#8]      ;将存储器地址为 R1+8 的字数据→R0,属于前变址模式
LDR   R0,[R1,R2]!     ;将存储器地址为 R1+R2 内容的字数据→R0,并将 R1+R2→R1,自动变址
                         模式
LDR   R0,[R1,#8]!     ;将存储器地址为 R1+8 的字数据→R0,并将 R+8→R1,自动变址模式
LDR   R0,[R1],R2      ;将存储器地址为 R1 内容的字数据→R0,并将 R1+R2→R1,属于后变址
                         模式
```

```
LDR   R0,[R1,R2,LSL#2]!  ;将存储器地址为 R1+R2×4 的字数据→R0,并将 R1+R2×4→R1。偏移量使用寄存器移位偏移量,属于自动变址模式
LDRB  R0,[R1,#8]         ;将存储器地址为 R1+8 的无符号字节数据→R0,并将 R0 的高 24 位(无效位)清零。属于前变址模式
LDRSB R0,[R1,#1]!        ;将存储器地址为 R1+1 的有符号字节数据→R0,并将 R0 的高 24 位用符号位填充。若为正数使用全"0";若为负,使用全"1"。属于自动变址模式
LDRH  R0,[R1,R2]         ;将存储器地址为 R1+R2 内容的无符号半字数据→R0,并将 R0 的高 16 位(无效位)清零。属于前变址模式
LDRSH R0,[R1,#2]!        ;将存储器地址为 R1+2 的有符号半字数据→R0,并将 R0 的高 16 位用其符号位填充。若为正数使用全"0";若为负,使用全"1"。同时 R0←R1+2,属于自动变址模式
STR   R0,[R1],#8         ;将 R0 中的字数据→R1 为地址的存储器中,并将 R1+8→R1。属于后变址模式
STRB  R0,[R1,#8]         ;将寄存器 R0 中的字节数据→以 R1+8 为地址的存储器中。属于前变址模式
```

2. LDM 和 STM 指令与应用示例

批量寄存器装载指令 LDM 完成的操作是，将存储器块中的 n 个字数据装载到 n 个寄存器中。n 个寄存器在指令中组成一个寄存器列表。批量数据存储指令 STM，就是将 n 个寄存器中的值写入到地址连续的存储器块中。

LDM 和 STM 相配合主要完成 2 项工作：一是可以完成将存储器中某一个首地址连续的数据块传送到存储器中的另一个数据区域中，实现数据的复制；二是用于堆栈区数据的压栈与弹栈。指令的格式如下：

```
LDM{cond}<mode>  Rn{!},reglist {^}
STM{cond}<mode>  Rn{!},reglist {^}
```

1）<mode>根据完成的工作，可分为 2 个类型。

类型 1 是在进行存储器块的复制工作时，使用以下 4 种模式之一。

IA：每次传送数据后地址寄存器 Rn 加 4，即先传送数据后修改地址指针（+4）。

IB：每次传送数据前地址寄存器 Rn 加 4，即先修改地址指针（+4）后传送数据。

DA：每次传送数据后地址寄存器 Rn 减 4，即先传送数据后修改地址指针（-4）。

DB：每次传送数据前地址寄存器 Rn 减 4，即先修改地址指针（-4）后传送数据。

类型 2 是当进行堆栈操作时，使用以下 4 种模式之一。

FA：满栈增堆栈；FD：满栈减堆栈；EA：空栈增堆栈；ED：空栈减堆栈。它们代表的物理操作前已讲述。

注意：压栈与弹栈指令选取的 mode 必须相同。

2）Rn：基址寄存器，装有传送数据的开始地址。当 Rn 使用 SP（R13）时，主要用于对堆栈区的操作；当 Rn 使用其他通用寄存器时，用于存储器区内数据块之间的复制。

3）{!}:若选取，则自动修改 Rn 内的有效地址值；否则指令执行后 Rn 中的内容不变。对于大于列表中寄存器个数的数据块复制一般都要选取使用，以减小汇编程序的额外开销；对于堆栈区域的操作必须使用，以保证堆栈指针的正确性。

4）reglist：一般是在 R0~R12 中，且除 Rn 之外的通用寄存器中选取，PC 寄存器、LR 寄存器也是列表中可包含的元素。列表的书写方法在 3.2.6 节多寄存器寻址方式中已经介绍。

5）{^}:"^"后缀不允许在用户模式和系统模式下使用，在其他模式下使用时，如果 LDM 指令中的寄存器列表包含有 PC 时，这时除了完成列表中的寄存器数据装载外，还将 SPSR 复

制到 CPSR 中，主要用于异常处理返回。

使用“^”后缀进行数据传送，且寄存器列表不包含 PC 时，装载/存储的是用户模式下的寄存器，而不是当前模式下的寄存器。

注意：LDM 和 STM 指令操作时，要求字对齐，否则会出现意想不到的问题；使用这两条指令时，使用的指令应该配对，即选取的 mode 相同。

堆栈操作时有 4 对指令：LDMFA/STMFA、LDMFD/STMFD、LDMEA/STMEA、LDMED/STMED，它们必须配对使用。

存储器操作时也有 4 对指令：LDMIA/STMIA、LDMIB/STMIB、LDMDA/STMDA、LDMDB/STMDB，它们最好配对使用。

应用示例：

```
LDMIA   R0!,{R2-R9}      ;将以 R0 为首地址的 8 个字单元内容分别装载到 R2~R9 中,R0 内容
                          更新
STMIA   R1!,{R2-R9 }     ;将 R2~R9 中的内容存到以 R1 为首地址的连续 8 个字单元中,R1 内
                          容更新
STMFA   SP!,{R0-R9,LR }  ;保护现场。压栈时 SP 先修改地址(+4)后压栈数据
LDMFA   SP!,{R0-R9,PC}   ;恢复现场,异常处理返回。弹栈时 SP 先弹出数据,再修改指针
```

以下再列举一例，功能是将以 R0 为首地址的 256 字节数据复制到以 R1 为首地址的存储器单元中。

```
        …
        MOV   R8,#256           ;计数器赋初值
        LDR   R0,=ScrData       ;LDR 是 ARM 伪指令,将 32 位的源数据地址值送 R0
        LDR   R1,=DstData       ;LDR 是 ARM 伪指令,将 32 位的目的数据地址值送 R1
LOOP    LDMIA   R0!,{R2-R7,R9,R10}  ;装载内容送 R2-R9。自动修改 R0
        STMIA   R1!,{R2-R7,R9,R10};存储 R2-R9 数据到以 R1 为首地址的单元中。自动修改 R1
        SUBS    R8,R8,#32       ;每次传送 32 字节数据。修改计数器值,影响标志 Z
        BNE   LOOP ;如果 R8≠0,Z=0,条件码 NE 为真,返回到 LOOP 继续复制。B 为跳转指令
        …
```

3.3.3 ARM 跳转指令与应用示例

在 ARM 指令中要实现汇编程序的跳转有两种方法：一种是直接向 PC 寄存器赋值，可实现程序在 4G 范围内的任意跳转；另一种是使用跳转指令，可实现相对于当前程序计数器 PC 指针的跳转，这是本节介绍的主要内容。

ARM 的跳转指令主要有以下 4 条。

B：跳转分支指令。

BL：带链接的跳转分支指令。

BX：带状态切换的跳转分支指令。

BLX：带链接和状态切换的跳转分支指令。

在通常状态下，处理器都是按顺序执行指令，当它执行到跳转分支指令时，将直接跳转到分支程序开始的地方去执行，程序也不会回到原来跳出的程序处继续执行。当指令执行到带链接的跳转分支指令时，程序将跳到分支程序开始的地方去执行，在这种情况下该分支程序是一个子程序，执行完后将返回到原来跳出的程序处继续执行。

1. B 指令和 BL 指令与应用示例

B 指令是一条最为简单的指令。指令在执行过程中一旦遇到 B 指令，ARM 微处理器将立

即跳转到指令给定的目标地址，从那里开始执行指令。该指令也可以选择根据其附带的条件码来执行，为程序的分支执行控制提供可能。

BL 指令是一条带有链接的跳转分支指令，除具有 B 指令的分支跳转功能外，还具有异常返回或返回主程序的功能。在执行跳转（即调用子程序）前，处理器自动地将当前程序计数器 PC（R15）的值存储在链接寄存器 LR（R14）中，作为将来子程序返回的指针。当子程序执行完毕后，将 LR 的内容复制到 PC 中，就可以实现子程序的返回。其指令格式如下：

```
B{L}{cond}    <Label>
```

其中，<Label>为程序跳转的相对于当前 PC 值的偏移量，两者结合构成一个相对转移地址，而不是绝对地址。它是将一个 24 位有符号数左移 2 位形成一个 26 位有符号的地址值，这样它的偏移量就是±32 MB，也就是说前偏移 32 MB、后偏移 32 MB 范围内的程序转移。由于 PC 寄存器是 32 位的，所以对于它的最高 6 位使用符号位进行填充，其数值的大小维持不变，使其构成一个 32 位的有效地址值。正数时填充值为全“0”，负数时为全“1”。它的值是由汇编器计算出来的，程序员只需要写上标号地址即可，但是它的取值范围程序员必须心中有数。

B 指令应用示例：

```
CMP   R0,#5
BEQ   Branch1   ;如果 R0=5,即比较后标志位 Z=1,则转移到分支 Branch1 处
BNE   Branch2   ;如果 R0≠5,即比较后标志位 Z=0,则转移到分支 Branch2 处
```

BL 指令应用示例：

```
              BL    Sub_Route1
              …
Sub_Route1    MOV   R1,R2
              …
              MOV   PC,LR
```

注意：当在 Sub_Route1 子程序中调用了 Sub_Route2，此时的 LR 又被调用 Sub_Route2 时的程序 PC 值所代替，使得子程序 Sub_Route1 的最后一条指令 MOV　PC，LR 执行有误。解决的方法如下：

```
              BL    Sub_Route1
                …
Sub_Route1    STMFA   SP!,{R1-R12,LR}   ;压栈保存 R1~R12,还有 Sub_Route1 的 LR
              MOV   R1,R2
                …
              BL   Sub_Route2   …        ;调用 Sub_Route2 子程序
                …                        ;Sub_Route2 的返回处
              LDMFA   SP!,{R1-R12,PC}    ;弹栈恢复 R1~R12,Sub_Route1 子程序返回
Sub_Route2      …
              MOV   PC,LR                ;Sub_Route2 子程序返回
```

如果在 Sub_Route2 子程序中还要调用子程序，处理方法相同。

2. BX 和 BLX 指令与应用示例

仿照上面的指令 B 和 BL，一个是实现程序的跳转不返回主程序指令，另一个是实现子程序调用而返回主程序指令，这里的 X 表示在跳转或子程序调用时 ARM 微处理器的工作状态也会发生变化，即由 ARM 指令工作状态转换到 Thumb 指令工作状态。

这两条指令在使用的过程中有两种形式。

形式 1：B{L}X{cond}　<Label>

形式 2：B{L}X{cond}　<Rm>

它们使用时的具体格式有 BX Label 指令、BLX Label 指令、BX Rm 指令、BLX Rm 指令。

BX Label 指令是由 ARM 指令程序跳转到 Thumb 指令程序的 Label 标号处执行，且不能返回到跳转时的 ARM 程序指令处，跳转的范围是±32 MB。

BLX Label 指令是由 ARM 指令程序跳转到 Thumb 指令程序处执行，在跳转时当前程序计数器 PC 的值已经存入 LR，当执行完 Thumb 子程序后，恢复 PC 的原值将会返回到跳转时的 ARM 程序指令处继续执行。跳转的范围是±32 MB。

BX Rm 指令既可以跳转到 ARM 指令程序处执行，也可以跳转到 Thumb 指令处执行，跳转后不会返回到主程序。当寄存器 Rm 的 b0=1 时，指令自动将 CPSR 的 T 位置“1”，程序转到 Thumb 指令处执行；当 Rm 的 b0=0 时，程序转到 ARM 指令处执行。注意跳转的地址由 Rm 确定。

BLX Rm 指令既可以跳转到 ARM 指令程序处执行，也可以跳转到 Thumb 指令处执行，在跳转时当前程序计数器 PC 的值已经存入 LR，执行完子程序后将链接寄存器 LR 的值复制给 PC，将返回到主程序。当寄存器 Rm 的 b0=1 时，指令自动将 CPSR 的 T 位置“1”，程序转到 Thumb 指令子程序入口处执行；当 Rm 的 b0=0 时，程序转到 ARM 指令子程序入口处执行。注意跳转的地址由 Rm 确定。

应用示例：

```
                CODE32                  ;ARM 代码程序
                ……
                BLX   Thumb_Sub1        ;调用 Thumb 子程序
                ……
                CODE16                  ;Thumb 代码程序
Thumb_Sub1                              ;Thumb 子程序入口
                ……
                BX   R14                ;返回 ARM 代码程序。R14 即 LR
                ……
```

3.3.4　ARM 杂项指令与应用示例

ARM 杂项指令主要包括程序状态寄存器操作和异常中断操作两种类型的指令，共有 4 条。

程序状态寄存器操作指令有 2 条，分别是程序状态寄存器传送到通用寄存器指令 MRS、通用寄存器或立即数传送到程序状态寄存器指令 MSR。

异常中断指令有软件中断指令 SWI 和断点中断指令 BKPT。

1. 程序状态寄存器操作指令

在 ARM 指令系统中，程序状态寄存器操作指令主要用于程序状态寄存器与通用寄存器间的数据传送，它们之间相互配合，通过 MRS 读程序状态寄存器→通用寄存器→修改→通过 MSR 再写回到程序状态寄存器，完成对程序状态寄存器的内容修改，从而进行异常模式的切换或修改普通中断控制位 I 和快速中断控制位 F。

（1）MRS 指令与应用示例

MRS 指令的功能是把状态寄存器 psr（包括 CPSR 和 SPSR）传送到通用目标寄存器。指令的格式如下：

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

注意：在 ARM 中只有通过该条指令才能读出状态寄存器（CPSR 或 SPSR）的内容到通用寄存器中；Rd 不允许使用 R15（PC）。

该指令一般在以下几种情况下使用。

- 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，然后压栈保存。

应用示例：

```
MRS    R1,CPSR      ;R1←CPSR
MRS    R2,SPSR      ;R2←SPSR
```

（2）MSR 指令与应用示例

MSR 指令的功能是把通用寄存器内容传送到状态寄存器 psr（包括 CPSR 和 SPSR）中。指令的格式如下：

```
MSR{cond}    < psr_fields>,< #immed|Rm >
```

其中<psr_fields>:psr 指的是 CPSR 或 SPSR 寄存器，fields 域是将这 2 个 32 位的寄存器按每 8 位进行划分，共划分为 4 个域，分别如下。

- 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（预留备用位）。
- 位[15:8]为扩展位域，用 x 表示（预留备用位）。
- 位[7:0]为控制位域，用 c 表示。

<#immed|Rm>：此地方操作数可以是一个 8 位的立即数，正好对应着上述的某一个域，为程序员的编程提供了便利。也可以使用寄存器，操作时仅将它对应的域值传送到 psr 的域中。

应用示例：

```
MSR   CPSR_f,#0xF0        ;CPSR[31:28]=0b1111,即 N、Z、C、V 均被置 1
```

2. 异常中断指令与应用示例

（1）SWI 指令

软件中断指令 SWI（Software Interrupt）用于产生软件中断，从而实现在用户模式下对操作系统中特权模式的程序调用等，以便用户程序能调用操作系统的系统例程。指令的格式如下：

```
SWI{cond}    24 位立即数
```

该指令的工作机制是这样的，当条件满足时执行该指令，处理器将进入管理模式中，需要完成以下任务。

- 将指令 SWI 后面的指令地址保存到 R14_svc 中。
- 将当前的 CPSR 保存到特权模式中相应的异常模式 SPSR_svc 中。
- 进入 SVC 管理模式，将 CPSR[4:0]设置为 0b10011 并将 CPSR［7］置 1，禁止 IRQ 中断。

- 将程序计数器（PC）的指针指向 0x00000008 的异常向量处，开始执行其中的程序，这里一般是一条跳转指令，跳转到某一标号后执行从 SWI 指令格式中析出 24 位立即数的程序中，根据析出的 24 位立即数，查找执行对应的处理程序。处理完毕后子程序返回。

由于在用户模式下不能执行所有特权模式下的一些操作，如模式之间的切换、各种异常模式堆栈的指针设置，T、IRQ、FIQ 位的清 0 或置 1 等。SWI 指令允许用户从用户模式进入特权模式，进行相应的操作，如关闭 IRQ 等。

操作系统也可以在 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的例程，相关参数通过通用寄存器传递。

应用示例：

```
            b HandlerSWI              ;该指令地址是 0x00000008,跳转到软件中断异常程序入口
            …
            T_bit   EQU 0x20          ;定义 Thumb 检测位
HandlerSWI
            STMFD   SP!,{R0-R12,LR}   ;保护现场
            MRS     R0,SPSR           ;当前 CPSR 内容送 R0
            TST     R0,#T_bit         ;检测 Thumb
            LDRNEH  R0,[LR,-2]        ;Thumb 指令,读取 16 位指令码
            BICNE   R0,R0,#0xFF00     ;析出其 SWI 指令中的低 8 位立即数,Thumb 指令只使用这低
                                       8 位
            BLXNE   R0
            LDREQ   R0,[LR,-4]        ;ARM 指令,读取 32 位指令码
            BICEQ   R0,R0,#0xFF000000 ;析出其 SWI 指令中的低 24 位立即数
            CMP   R0,#0x00
            BLEQ subroutine0          ;如果 R0=0,则调用 SWI  #0 指令对应的子程序 subroutine0
            CMP   R0,#0x01
            BLEQ subroutine1          ;如果 R0=1,则调用 SWI  #1 指令对应的子程序 subroutine1
            CMP   R0,#0x02
            BLEQ subroutine2          ;如果 R0=2,则调用 SWI  #2 指令对应的子程序 subroutine2
            …
            LDMFD   SP!,{R0-R12,PC}   ;SWI 异常软中断返回

SWI_sub     DCD   subroutine0         ;立即数是 0  SWI  #0
            DCD   subroutine1         ;立即数是 1  SWI  #1
            DCD   subroutine2         ;立即数是 2  SWI  #2
subroutine0
            …
            MOV   PC,LR
subroutine1
            …
            MOV PC,LR
subroutine2
            …
            MOV PC,LR
```

需要说明的是，软中断是由用户调用的，具有可知性；IRQ 或 FIQ 是由外部条件触发的，具有随机性。

当指令中 24 位的立即数被忽略时，用户程序调用的系统例程也可由通用寄存器 R0 的内容决定，同时，参数通过其他通用寄存器传递。

```
MOV   R0,#12      ;通过 R0 传递 12 号软中断的参数
MOV   R1,#34      ;设置子功能号
SWI   0           ;等价于 24 位的立即数被忽略,0 在这里代表特定的含义,其他地方不能再使用
```

（2）BKPT 指令

断点中断指令 BKPT 主要用于产生软件中断，供调试程序使用。指令格式如下：

```
BKPT        16 位立即数
```

该 16 位立即数被调试软件用来保存额外的断点信息。

3.3.5 杂项指令在 Bootloader 中配置各种异常栈顶指针综合应用示例

下列程序段是利用它们之间的配合，通过“读取→修改→写回”操作完成各种异常堆栈指针的设置。这也是 Bootloader 中主要完成的任务之一。

1. 异常模式字定义（EQU 是汇编伪指令）

```
USERMODE    EQU    0x10        ;用户模式
FIQMODE     EQU    0x11        ;快速中断模式
IRQMODE     EQU    0x12        ;普通中断模式
SVCMODE     EQU    0x13        ;管理模式
ABORTMODE EQU      0x17        ;终止模式
UNDEFMODE EQU      0x1b        ;未定义模式
MASKMODE    EQU    0x1f        ;系统模式
NOINT       EQU    0xc0        ;普通中断、快速中断屏蔽字
```

2. 系统上电进入到管理模式

```
HandlerReset
    MRS   R0,  CPSR
    BIC   R0,  R0,  #NOINT|MASKMODE ;R0[7:0]=0b00x00000,IRQ、FIQ 开中断
    ORR   R2,R0,  #USERMODE         ;R2 低 5 位是用户模式字
;初始化快速中断模式栈顶指针
    ORR   R1,R0,  #NOINT|FIQMODE    ;R1 的低 8 位是普通中断、快速中断屏蔽位和模式字
    MSR   CPSR_cf,  R1              ;cf(control flag)是 ADS1.2 中使用的域定义符
    MSR   SPSR_cf,R2                ;将用户模式字保存在 SPSR 中
    LDR   SP,=FIQStack              ;FIQStack 是一个 32 位的二进制数,快中断栈顶指针
;初始化普通中断模式栈顶指针
    ORR   R1,R0,  #IRQMODE|NOINT
    MSR   CPSR_cf,R1
    MSR   SPSR_cf,R2
    LDR   SP,=IRQStack              ;IRQStack 是一个 32 位的二进制数,普通中断栈顶指针
;初始化管理模式栈顶指针
    ORR   R1,  R0,  #SVCMODE|NOINT
    MSR   CPSR_cf,R1
    MSR   SPSR_cf,R2
    LDR   SP,=SVCStack              ;SVCStack 是一个 32 位的二进制数,是管理模式栈顶
                                     指针
;初始化其他模式的栈顶指针:省略…
;最后初始化用户模式栈顶指针后,系统进入用户模式
    MRS   R0,  CPSR
    BIC   R0,  R0,  #MASKMODE|NOINT
    ORR   R1,R0,  #USERMODE
    MSR   CPSR_cf,R1
    LDR SP,=UserStack               ;UserStack 是一个 32 位的二进制数,是用户模式栈顶
                                     指针
      …
```

注意：只有在特权模式下才能修改状态寄存器控制域[7:0]的值，以实现处理器模式的转换或禁止/允许中断异常。

控制域中的 T（Thumb）位在 MSR 指令中不能赋值修改，而将 ARM 工作状态切换到 Thumb 工作状态。只有使用 BX 指令才能实现前述工作状态的切换。

在用户模式下，只能修改“条件标志位域”，不能修改其他域，即 CPSR 的[24:0]位。

在 MRS 和 MSR 指令中不能使用后缀“S”。

3.3.6 ARM 协处理器指令与应用示例

ARM 微处理器支持协处理器操作，协处理器的控制要通过协处理器命令实现。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 微处理器和其他协处理器的指令。如果协处理器不能成功地执行其操作，将产生未定义指令异常。

ARM 协处理器指令的主要作用包括：①ARM 处理器初始化；②ARM 协处理器的数据处理操作；③ARM 微处理器寄存器到协处理器寄存器之间的数据传送；④ARM 协处理器寄存器到 ARM 存储器之间的数据传送。

协处理器共有 5 条指令，见表 3-4，下面分别介绍。

表 3-4 ARM 协处理器指令表

助 记 符	功 能	完成的操作	条件码位置
CDP coproc,opcode1,CRd,CRn,CRm {,opcode2}	协处理器操作	协处理器决定	CDP{cond}
LDC{L} coproc,CRd,<addr>	协处理器数据装载	协处理器决定	LDC{cond}{L}
STC{L} coproc,CRd,<addr>	协处理器数据存储	协处理器决定	STC{cond}{L}
MCR coproc,opcode1,CRd,CRn,CRm {,opcode2}	ARM 寄存器到协处理器寄存器的数据传送	协处理器决定	MCR{cond}
MRC coproc,opcode1,CRd,CRn,CRm {,opcode2}	协处理器寄存器到 ARM 寄存器的数据传送	协处理器决定	MRC{cond}

1. CDP 指令与应用示例

CDP 指令为协处理器操作指令。ARM 微处理器通过 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，即对命令参数的解释与协处理器有关，指令的使用也取决于协处理器。指令格式如下：

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

其中，coproc 是协处理器名，书写格式为 Pn,n=0~15；opcode1 为协处理器操作码；CRd 为协处理器目标寄存器；CRn、CRm 为协处理器的第 1、2 操作数寄存器；opcode2 是 opcode1 的可选子操作码。

注意：协处理器寄存器使用时使用 Cn,n=0,1,2,3……

应用示例：

```
CDP   P1,10,C1,C2,C3      ;协处理器 P1 完成操作 10,C2 和 C3 为源操作数,结果送 C1
CDP   P3,5,C1,C2,C3,2     ;协处理器 P3 完成操作 5(子操作 2),C2 和 C3 同上,结果送 C1
```

2. LDC/STC 指令与应用示例

协处理器数据装载指令 LDC 是从连续的存储单元将数据读取到协处理器寄存器中。协处理器数据传送的字数由协处理器控制。协处理器数据存储指令 STC 与 LDC 的功能相反。指令的格式如下：

```
LDC/STC{cond}{L}  coproc,CRd,<addr>
```

其中，<addr>是 ARM 微处理器的基址寄存器 Rn 的间接寻址或移位寻址，其他同上。选取后缀“L”时，表示为长整数传送，用于双精度的数据传输。

应用示例：

```
LDC   P3,C2,[R2]       ;将地址为 R2 的存储单元数据传送到 P3 协处理器的 C2 寄存器
STC   P7,C5,[R1,#4]    ;将 P7 协处理器的 C5 寄存器内容存储到[R1+4]的单元中
```

3. MCR/MRC 指令与应用示例

传送指令 MCR 是将 ARM 寄存器的内容传送到协处理器寄存器中去。传送指令 MRC 是将协处理器寄存器的内容传送到 ARM 寄存器中去。指令格式如下：

```
MCR/MRC{cond}   coproc,  opcode1,  Rd,  CRn,  CRm  {,opcode2}
```

应用示例：

```
MCR   P6,2,R7,C1,C2      ;将寄存器 R7 的内容送协处理器 P6 寄存器 C1、C2,操作码是 2
MCR   P7,0,R1,C3,C2,1    ;将寄存器 R1 的内容送协处理器 P7 寄存器 C3、C2,操作码是 0(1)
MRC   P5,2,R2,C1,C2      ;将协处理器 P5 寄存器 C1、C2 的内容送寄存器 R2,操作码是 2
MRC   P4,0,R0,C3,C2,2    ;将协处理器 P4 寄存器 C3、C2 的内容送寄存器 R0,操作码是 0(2)
```

3.3.7 ARM 伪指令与应用示例

ARM 伪指令不是 ARM 指令集中的指令，只是为了编程方便定义的指令，使用时可以像其他 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 ADR，ADRL，LDR，NOP 共 4 条。

1. ADR 指令

小范围的地址读取伪指令 ADR 将基于 PC 相对偏移的地址值加载到寄存器中。指令格式如下：

```
ADR{cond}   register,expre
```

其中，register 为加载的目标寄存器，expre 为地址表达式。当地址值是非字对齐地址时，取值范围为-255~255 字节；当地址是字对齐地址时，取值范围为-1020~1020 字节。

ADR 伪指令在 ARM 汇编时始终被汇编成一条指令。在编程时程序员并不需要关心相对于 PC 的偏移量，也不需要计算偏移量，由汇编器自动完成。汇编器会试图产生一条 ADD 指令或 SUB 指令来加载地址，若地址加载不能汇编成一条指令，则产生错误，汇编失败。若标号是程序相对偏移量，则它的值必须与 ADR 伪指令在同一代码存储区域。

应用示例：

```
Label      MOV   R0,   #20
           ADR   R1,   Label
           ADR   R2,   Data_Tab        ;将字数据表的标号送给 R2,作为查表的基址
           LDR   R3,   [R2,R4]         ;R4 为字索引号,依次是 0,4,8,12,…,R3 是查表的结果
Data_Tab DCD 0x01,0x02,0x03            ;定义字数据
```

上述第二条伪指令将被汇编成 SUB　R1，PC，0x0C 指令。注意此时的 PC 值是当前指令地址+8，因此当前的 PC 值减 12（0x0C），就是 Label 的值。执行上述第 3 条伪指令 ADR 时，当前的 PC 指针也是指令地址+8，即 Data_Tab 标号处，所以此指令将汇编成 ADD R2，PC，#0x00。

2. ADRL 指令

中等范围的地址读取伪指令 ADRL 将程序相对偏移或寄存器相对偏移地址加载到寄存器中。在汇编编译源程序时，ADRL 伪指令被编译器替换成两条合适的指令。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。指令格式如下：

```
ADRL{cond}  register,expre
```

其中，register 为加载的目标寄存器，expre 为地址表达式。当地址值是非字对齐地址时，取值范围为-64K~64K 字节；当地址值是字对齐地址时，取值范围为-256K~256K 字节。

ADRL 伪指令始终被汇编成两条指令。即使地址加载可以用一条指令完成，汇编器也会生成两条冗余指令。在编程时程序员并不需要关心相对于 PC 的偏移量是多少，也不需要计算偏移量，由汇编器自动完成。若标号是程序相对偏移，则它的值必须与 ADRL 伪指令在同一代码存储区域。

应用示例：

```
Label   MOV   R0,   #100
ADRL    R1,Label+6000
```

汇编器将第 2 条伪指令生成 ADD R1，PC，#0xE800 和 ADD R1，R1，#0x254 两条指令。注意当前的指令 PC 值是 PC+8，即等价到第 1 条指令地址是 Label 标号地址+12。现在程序要以 PC 为基址，所以相对于 Label+6000 的地址值，就是相对于当前的 PC 值+6000-12=PC 值+5988(0xE800+0x254)。

3. LDR 指令

大范围的地址读取伪指令 LDR 用于加载 32 位的立即数或一个地址值到指定寄存器。在汇编器编译源程序时，LDR 伪指令被汇编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，否则汇编器将产生文字常量放入文字池，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{cond}  register,[=expre | label_expre]
```

其中，register 为加载的目标寄存器，expre 为 32 位立即数，labe1_expre 为程序相对偏移或外部表达式。

以下说明 LDR 伪指令的解释过程。举例如下：

```
LDR   R2,=0xff ;此立即数可以用 12 位来表示,所以汇编器将它汇编成 MOV   R2,#0ff
LDR   R5,=0xfff;此立即数不能用 12 位来表示,所以汇编器将它汇编成 LDR   R5,[PC,offset_pool]
LDR   R7,=data ;若此立即数不能用 12 位来表示,汇编器将它汇编成 LDR   R7,[PC,offset_pool]
```

上述的 offset_pool 是文字池中由系统自动定义的字单元数据相对于 pool 的位置偏移量，程序员只需在其后适当的地方（具体参见 4.1.3 节）写一条文字池声明伪指令 LTORG 即可。自动定义的内容是：pool DCD 0xfff，data。

这条 ARM 伪指令与寄存器装载指令有着相同的操作码，但是它的操作数前需要使用“=”符号。LDR 指令是使用频率最高的指令，也最为方便，程序员不必关心赋值范围是否越限。

应用示例：

(1) 作为 32 位的立即数程序

```
LDR    R0,=0x87654321          ;给 R0 赋立即数,可以是任意的 32 位立即数
LDR    R1,=0x12345677          ;给 R1 赋立即数
ADD    R2,R0,R1                ;R2←R0+R1
…
LTORG                          ;伪指令,声明文字池。当程序中的立即数不能用 MOV 指令后的
                                12 位立即数表示时,ARM 可以自动生成这些立即数并存储在
                                此,这时 ARM 可以使用其他指令进行操作,从而完成 LDR 伪指
                                令的功能
```

(2) 作为 32 位的地址值程序

```
GPBCON   EQU   0x56000010      ;使用伪指令 EQU 定义 B 端口控制寄存器地址
…
LDR   R0,=0xff000111           ;定义控制字送 R0
LDR   R1,=GPBCON               ;B 端口控制寄存器地址值送 R1
STR     R0,  [R1]              ;将控制字写入端口寄存器
…
LTORG
```

注意：ARM 伪指令是假的伪指令，是 ARM 编译器定义的伪指令，会生成相应的机器码。而一般意义下的伪指令只供汇编器使用，不产生机器码。

另外，由于 ARM 都是单字指令，无法将任意一个 32 位的二进制数赋值给它的寄存器，只能将 8 位的二进制数乘以整数 2 * (0~15)中的偶数通过 MOV 指令传递给寄存器（原理前已讲述），这样造成了不能将任意的 32 位立即数送入系统作为操作数或有效的地址单元。为了弥补这一点，便引入了 ARM 伪指令。

4. NOP 指令

空操作指令 NOP 在汇编时将会被代替成 ARM 中的空操作，如 MOV，R0，R0 指令等。NOP 可用于延时操作。NOP 伪指令格式如下：

```
NOP
```

应用示例：

```
          MOV   R1,0xFF00
DELAY1    NOP
          SUBS   R1,R1,#1
          BNE   DELAY1
```

Thumb 指令集如果读者需要，可以参考其他书籍学习，这里不再赘述。

习题

3-1　简述 ARM 指令的特点。

3-2　简述 ARM 指令格式及各项的含义。ARM 指令中的第 2 操作数 operand2 有哪些具体形式？

3-3　ARM 指令条件码有哪些？取决于哪个寄存器？

3-4　ARM 处理器有哪几种基本寻址方式？

3-5　在 ARM 的基址+变址寻址方式中，变址寻址方式有哪几种？举例说明。

3-6　在多寄存器寻址方式中，修改地址的方式有哪些？

3-7　存储器生长堆栈可分为哪几种？各有什么特点？

3-8　ARM 微处理器支持哪几种类型的堆栈工作方式？各有什么特点？

3-9　举例说明块复制寻址的操作过程。

3-10　举例说明变址寻址的操作过程。

3-11　ARM 指令集包含哪些类型的指令？

3-12　ARM 指令集分为哪几大类？

3-13　举例说明 LSL、LSR、ASR、ROR、RRX 的移位操作过程。

3-14　ARM 数据处理指令分为几类？

3-15　ARM 的比较指令与一般的数据处理指令有什么不同？

3-16　ARM 的寄存器装载与存储的基本指令是什么？由它派生出了几种同类的指令？分别是什么？

3-17　简述 ARM 跳转指令的条数及其功能。

3-18　简述 ARM 杂项指令及其功能。

3-19　ARM 协处理器指令作用是什么？简述 5 条指令各完成的功能。

3-20　简述 ARM 伪指令的功能，举例说明操作过程。

3-21　存储器从 0x30040000 开始的 100 个单元中存放着 ASCII 码，编写汇编程序，将其所有的小写字母转换为大写字母，其他保持不变。

3-22　编写程序，比较存储器中 0x30040000 和 0x30040004 两无符号字数据的大小，并且将比较结果存于 0x30040008 的字单元中，若两者相等结果记为 0，若前者大于后者结果记为 1，若前者小于后者结果记为-1（以补码的形式存储）。

3-23　将存储器中 0x30080000 开始的 200 字节数据复制到 0x30086000 开始的区域。

3-24　编写一简单 ARM 汇编程序，实现 1+2+…+100 的运算。

3-25　要实现多个寄存器的内容的压栈和弹栈，举例说明使用什么汇编指令。