第 1 章　ARM 嵌入式技术概论

1.1　ARM 处理器的历史及发展

嵌入式技术是当前微电子技术与计算机技术结合的技术，以嵌入式计算机为核心的嵌入式系统是继 IT 网络技术之后信息技术又一个新的发展方向。嵌入式系统是“量身定做”的“专用计算机应用系统”，由嵌入式硬件和嵌入式软件组成。嵌入式系统硬件的核心是嵌入式微处理器，嵌入式系统软件主要包括嵌入式应用软件和嵌入式操作系统。

ARM 即 Advanced RISC Machines 的缩写，具有多种含义。它既可以代表一个公司的名字，也可以代表微处理器内核，即 ARM 公司设计的知识产权（IP）核——ARM 核，还可以代表一类嵌入式处理器，即使用 ARM 核的嵌入式微处理器——ARM 处理器。当前，ARM 处理器凭借其卓越的性能和显著的优点，已经成为高性能、低功耗、低成本嵌入式处理器的代名词，得到了众多的半导体厂家和整机厂商的大力支持。

1990 年，Advanced RISC Machines Limited 在英国剑桥成立，后来简称为 ARM Limited，即 ARM 公司。ARM 公司是设计公司，专门从事基于 RISC 芯片技术开发，是知识产权（IP）供应商。

ARM 公司本身不直接从事芯片生产，主要出售芯片设计技术的授权。世界各大半导体生产商从 ARM 公司购买其设计的 ARM 微处理器核，再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 微处理器芯片进入市场。目前，全世界有几十家大的半导体公司使用 ARM 公司的授权，因此不仅使得 ARM 技术获得更多的第三方工具、制造、软件的支持，又使整个系统成本降低，使产品更容易进入市场被消费者所接受，更具有竞争力。ARM 公司于 1993 年开发了 ARM7 系列处理器内核，随后相继推出了 ARM9 系列、ARM9E 系列、ARM10E 系列、ARM11 系列、Cortex 系列、SecurCore 系列处理器。

ARM7、ARM9、ARM11 都是经典系列，也就是上一代处理器，其中 ARM9、ARM11 架构被采用得比较多，有不少中端 MID 平板电脑的处理器采用这种架构。新的 ARM Cortex 处理器系列包括了 ARMv7 架构的所有系列，分为 A、R、M 三类，旨在为各种不同的市场提供服务。Cortex－A 系列针对日益增长的且运行 Linux、Windows CE 和 Symbian 操作系统的消费娱乐产品和无线产品；Cortex－R 系列针对需要运行实时操作系统来进行控制应用的系统，包括汽车电子、网络和影音系统；Cortex－M 系列则是为那些对开发费用非常敏感，同时对性能要求不断增加的微控制器应用所设计的。Cortex 系列处理器的发展路线如图 1-1 所示。

试读版本仅供防疫期间线上教学使用

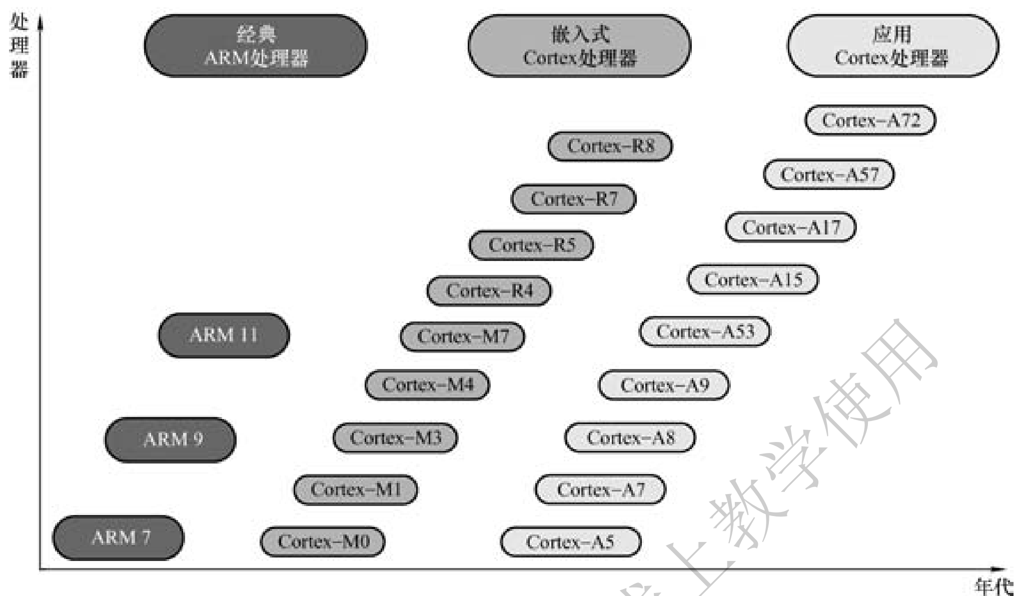

图 1-1 Cortex 系列处理器发展路线

试读版本仅供防疫期间线上教学使用

1.2 ARM 处理器简介

1.2.1 ARM 处理器特征

ARM 内核采用精简指令集计算机（RISC）体系结构。ARM 处理器的主要特征如下：

1）采用大量的寄存器，它们都可以用于多种用途。

2）每条指令都可以有条件执行。

3）能够在单时钟周期执行的单条指令内完成一项普通的移位操作和一项普通的 ALU 操作。

4）通过协处理器指令集来扩展 ARM 指令集，包括在编程模式中增加了新的寄存器和数据类型。

5）ARM 内核增加了一套称为 Thumb 指令的 16 位指令集，使得内核既能够执行 16 位指令，也能够执行 32 位指令，从而增强了 ARM 内核的功能。

1.2.2 ARM 处理器架构

目前，ARM 设计的处理器体系结构已经从 v1 发展到了 v8，ARM 处理器的架构见表 1-1。

表 1-1 ARM 处理器的架构

架构	处理器系列
ARMv1	ARM1
ARMv2	ARM2、ARM3
ARMv3	ARM6、ARM7

（续）

架构	处理器系列
ARMv4	StrongARM、ARM7TDMI、ARM9TDMI
ARMv5	ARM7EJ、ARM9E、ARM10E、XScale
ARMv6	ARM11、ARM Cortex－M
ARMv7	ARM Cortex－A、ARM Cortex－M、ARM Cortex－R
ARMv8	Cortex－A57、Cortex－A72

ARM 早期的几款内核架构已经被后来的新架构所取代，目前主流的架构有 ARMv4、ARMv5、ARMv6、ARMv7 以及最新的 ARMv8 等，基于这 5 种架构的 ARM 微处理器又可分为 ARM7（不包括 v3 架构部分）、ARM9、ARM9E、ARM10E、ARM11 以及 Cortex－A/M/R 等主流系列。

1.2.3　Cortex 处理器架构

Cortex 系列处理器基于 ARMv7 架构和最新的 ARMv8 架构。下面对这两种新的架构进行简单介绍。

新一代 ARMv7 架构是在 ARMv6 架构的基础上诞生的，该架构采用了 Thumb－2 技术，它是在 ARM 的 Thumb 代码压缩技术的基础上发展起来的，并且保持了对现存 ARM 解决方案完整的代码兼容性。Thumb－2 技术比纯 32 位代码少使用 31% 的内存，减小了系统开销，同时比已有的基于 Thumb 技术的解决方案高出 38% 的性能。ARMv7 架构还采用了 NEON 技术，将 DSP 和媒体处理能力提高了近 4 倍，并支持改良的浮点运算，满足下一代 3D 图形、游戏物理应用及传统嵌入式控制应用的需求。

最新的 ARMv8 架构基于 32 位的 ARMv7 架构，包含两个主要的执行状态：AArch64 和 AArch32。AArch64 执行状态针对 64 位处理技术，引入了一个全新的指令集 A64，且支持现有的 ARM 指令集。ARMv8 将 64 位架构支持引入 ARM 架构中，保留了 TrustZone 安全执行环境、虚拟化、NEON（高级 SIMD）等 ARMv7 的关键技术特性，应用于有更高要求的产品领域，如企业应用、高档消费电子产品等。

1.3　ARM 处理器系列

目前的 ARM 处理器主要包括 Classic ARM 处理器、Cortex－A 系列处理器、Cortex－M 系列处理器、Cortex－R 系列处理器以及 SecurCore 系列处理器。Cortex－A 系列处理器是针对尖端的基于虚拟内存的操作系统和用户应用设计的，Cortex－M 系列处理器对微控制器和低成本应用提供优化，Cortex－R 系列处理器是面向实时系统的。

1.3.1　Classic ARM 处理器

虽然 Classic ARM 处理器的推出时间已超过 15 年，但是 ARM7TDMI 仍是市场上销量最高的 32 位处理器，占据目前市场上销售的所有 32 位处理器市场份额的四分之一，所以该类产品仍处于核心地位。

Classic ARM 处理器由三个处理器系列组成：

1）ARM7 系列：ARM7TDMI－S 和 ARM7EJ－S 处理器。

2）ARM9 系列：ARM926EJ－S、ARM946E－S 和 ARM968E－S 处理器。

3）ARM11 系列：ARM1136J（F）－S、ARM1156T2（F）－S、ARM1176JZ（F）－S 和 ARM11MPCore 处理器。

各系列处理器的说明见表 1-2。在新的设计中可以采用更高性能的 Cortex 系列处理器代替相应的 Classic ARM 处理器，具体替代关系也在表 1-2 中给出。

表 1-2　Classic ARM 各系列处理器说明

系列	处理器	说　　明	Cortex 替代产品
ARM11	ARM11MPCore	率先采用了多核技术，并继续为各种不同的应用场合授权，包括手机、导航设备等等	Cortex－A9 Cortex－A5
	ARM1176JZ（F）－S	是 Classic ARM 系列中的最高性能单核处理器，它引入了 TrustZone 技术，从而可以在恶意代码所及范围之外安全执行操作。它在各种不同的应用领域得到广泛授权，可用于当今主流品牌的手机、机顶盒、数字电视、高端相框和其他众多应用领域	Cortex－A9 Cortex－A8 Cortex－A5
	ARM1156T2（F）－S	是最高性能的实时 Classic ARM 处理器，它首次引入了 Thumb－2 指令集架构。该处理器在高性能确定性控制系统（例如汽车、工业控制和机器人解决方案）中很有用	Cortex－R4
	ARM1136J（F）－S	除扩展流水线、频率和性能之外、ARM1136J（F）－S 在许多方面都与 ARM926EJ－S 相似。该处理器还引入了基本单指令多数据（Single Instruction Multiple Data，SIMD）指令来提高编解码器性能，并提供可选浮点支持	Cortex－A5
ARM9	ARM968E－S	面积最小，功耗最低的 ARM9 处理器是众多实时类型应用的理想之选。通过可轻松从标准接口集成的紧密耦合内存，该处理器可高效工作	Cortex－R4
	ARM946E－S	包含可选 cache 接口以及完整的内存保护单元的实时处理器，对于大部分代码位于主存储器中的应用，该处理器非常有用。它按需加载到 cache 中，同时关键的异常处理代码和数据仍本地保留在紧密耦合内存中	Cortex－R4
	ARM926EJ－S	是入门级处理器。可支持完全版操作系统，包括 Linux、Windows CE 和 Symbian。因此，该处理器是众多需要完整图形用户界面应用的理想之选	Cortex－A5
ARM7	ARM7TDMI－S	是出色的重负荷处理器，适用于众多应用领域，该处理器通常用于手机，现在广泛用于移动和非移动应用领域	Cortex－M3 Cortex－M0

试读版本仅供防疫期间线上教学使用

1.3.2　Cortex－A 系列处理器

Cortex－A 系列处理器提供了一系列用于执行复杂计算任务的解决方案，例如它支持多个操作系统（OS）平台以及多个软件应用程序的设备解决方案。另一方面，Cortex－A 系列

处理器在功耗和兼容性方面也拥有显著的优势。

Cortex - A 系列包括高性能的 Cortex - A17、成熟的 Cortex - A15、被广泛运用的 Cortex - A9 和高效率的 Cortex - A7、Cortex - A5 处理器，这些处理器都使用相同的 ARMv7 - A 架构，因此它们对应用程序具有良好的兼容性，包括对传统的 ARM、Thumb 和高性能的 Thumb - 2 指令集的支持。采用 ARMv8 - A 架构的 Cortex - A72、Cortex - A57、Cortex - A53 和 Cortex - A35 处理器都支持 64 位计算。ARMv8 - A 架构还拥有一个专门的执行状态，允许它来处理传统的 ARM32 位应用程序。这提供了对现有的 32 位生态系统升级的较好方法，并确保 64 位的生态系统是向后兼容的。

Cortex - A 系列的处理器众多，本节只对最新的 Cortex - A72、成熟的 Cortex - A15 和被广泛运用的 Cortex - A9 处理器作简单介绍。读者如需要其他处理器的详细信息可以自行到 ARM 公司官网（www. arm. com）查阅相关资料。

1. Cortex - A9 处理器

Cortex - A9 处理器是低功耗、散热良好、成本要求高的设备上的通用选择，例如智能手机、数字电视等，并且消费者和企业也将其应用在实现物联网上。

Cortex - A9 多核处理器是首款结合了 Cortex 应用级架构以及具有可扩展性能的多处理能力的 ARM 处理器，其提供了下列增强的多核技术：

1）加速器一致性端口（ACP），用于提高系统性能和降低系统能耗。

2）先进总线接口单元（Advanced Bus Interface Unit），用于在高带宽设备中实现低延迟时间。

3）多核 TtustZone 技术，结合中断虚拟，允许基于硬件的安全和加强的类虚拟（para-virtualization）解决方案。

4）通用中断控制器（GIC），用于软件移植和优化的多核通信。

Cortex - A9 处理器的基本信息见表 1-3。

表 1-3　Cortex - A9 处理器的基本信息

项　　目	采用的技术
体系结构	ARMv7 - A Cortex
Dhrystone 性能	每个内核 2. 50 DMIPS/MHz
多核	1 ~4 个核，也提供单核版本
ISA 支持	ARM Thumb - 2/Thumb Jazelle DBX 和 RCT DSP 扩展 高级 SIMD NEON 单元（可选） 浮点单元（可选）
内存管理	内存管理单元
调试和追踪	CoreSight DK - A9（单独提供）

2. Cortex - A15 处理器

在 Cortex - A9 双核处理器之后，ARM 推出了一款型号为 Cortex - A15 的多核处理器。

试读版本仅供防疫期间线上教学使用

Cortex－A15 的最快处理速度能达到 2.5GHz，还可以支持超过 4G 的存储。

Cortex－A15 处理器基于 ARMv7－A Cortex 微架构，单个处理器集群拥有 1～4 个 SMP 处理核心，彼此通过 AMBA4 技术互联，支持一系列 ISA，能够在不断下降的功耗和成本预算的基础上提供高度可扩展性的解决方案，广泛适用于智能手机、平板电脑、大屏幕移动计算设备、高端数字家庭娱乐终端、无线基站、企业基础架构产品等。Cortex－A15 处理器的基本信息见表 1-4。

表 1-4　Cortex－A15 处理器的基本信息

项　　目	采用的技术
体系结构	ARMv7－A Cortex
多核	单个处理器群集中可配置 1～4 个 SMP 核心 通过 AMBA4 技术实现多个一致的 SMP 处理器群集
ISA 支持	ARM Thumb－2 TrustZone 安全技术 NEON 高级 SIMD DSP&SIMD 扩展 VFPv4 浮点 Jazelle RCT 硬件虚拟化支持 大物理地址扩展（LPAE）
内存管理	ARMv7 内存管理单元
调试和追踪	CoreSight DK－A15

版本仅供防疫期间线上教学使用

3. Cortex－A72 处理器

于 2015 年年初正式发布的 Cortex－A72 处理器是 ARM 公司性能最高、最先进的处理器之一，其基于 ARMv8－A 架构，并构建于 Cortex－A57 处理器在移动和企业设备领域获得成功的基础之上。在相同的移动设备电池寿命限制下，Cortex－A72 相较基于 Cortex－A15 的设备能提供 3.5 倍的性能表现，展现优异的整体功耗性能。

Cortex－A72 的强化性能和功耗水平为消费者带来超凡的体验，这些高端设备包括高档的智能手机、中型平板电脑、大型平板电脑、翻盖式笔记本电脑等外形规格可变化的移动设备。未来的企业基站和服务器芯片也能受惠于 Cortex－A72 的性能，并在其优异的能效基础上和有限的功耗范围内增加内核数量，提升工作负载量。

Cortex－A72 是目前基于 ARMv8－A 架构的处理器中性能最高的处理器。它再次展现了 ARM 在处理器技术中的领先地位，在提升新的性能标准之余大幅降低了功耗，广泛地扩展应用于移动设备与企业设备。

Cortex－A72 可在芯片上单独实现，也可以搭配 Cortex－A53 处理器与 ARM CoreLink CCI 高速缓存一致性互连（Cache Coherent Interconnect）构成 ARM big.LITTLE 配置，进一步提升能效。Cortex－A72 处理器的基本信息见表 1-5。

表 1-5　Cortex－A72 处理器的基本信息

项　　目	采用的技术
体系结构	ARMv8－A Cortex
多核	单个处理器群集中可配置 1 ~4 个 SMP 核心 通过 AMBA 5 CHI 或 AMBA 4 ACE 技术，可实现多个一致的 SMP 处理器群集
ISA 支持	AArch32 可完全向下兼容 ARMv7 AArch64 提供 64 位支持和全新架构功能 TrustZone 安全技术 NEON 高级 SIMD DSP 和 SIMD 扩展 VFPv4 浮点 硬件虚拟化支持
调试和跟踪	CoreSight DK－A72

1.3.3　Cortex－M 系列处理器

Cortex－M 系列处理器针对那些对成本和功耗敏感的 MCU 和终端应用进行了优化。这些终端应用包括智能测量、人机接口设备、汽车和工业控制系统、大型家用电器、消费性产品和医疗器械等。针对十分广泛的嵌入式应用，每个处理器都提供最佳权衡取舍。Cortex－M 系列处理器的特点如图 1-2 所示。

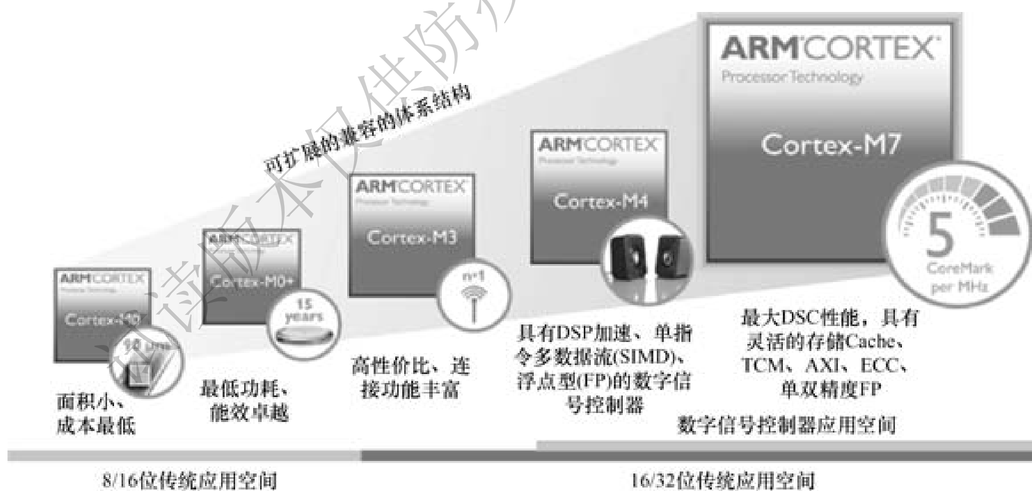

图 1-2　Cortex－M 系列处理器的特点

Cortex－M 系列处理器都是向上兼容的，这使得软件重用以及从一个 Cortex－M 处理器无缝发展到另一个成为可能。

Cortex－M0 处理器是目前最小的 ARM 处理器。该处理器的芯片面积非常小，功耗极低，且编程所需的代码占用量很少，这使得开发人员可以直接跳过 16 位系统，以接近 8 位系统的成本获取 32 位系统的性能。

Cortex－M0＋处理器是能效极高的 ARM 处理器。它以极为成功的 Cortex－M0 处理器为

基础，保留了全部指令集和数据兼容性，同时进一步降低了能耗，提高了性能。它与 Cortex - M0 处理器一样，芯片面积很小，功耗极低，并且所需的代码量极少。

Cortex - M3 处理器是行业领先的 32 位处理器，适用于具有较高确定性的实时应用，经过专门开发，可以针对广泛的设备（包括微控制器、汽车车身系统、工业控制系统以及无线网络和传感器）开发高性能低成本平台。

Cortex - M3 处理器具有出色的计算性能以及对事件的优异响应能力，并且可应对实际应用中对低功率（包括动态和静态）需求的挑战。此处理器配置十分灵活，从而可满足广泛的应用。

Cortex - M4 处理器是 ARM 公司专门开发的最新嵌入式处理器之一，用于需要控制功能和数字信号处理功能相结合的领域。

高效的信号处理功能与 Cortex - M 处理器系列的低功耗、低成本和易于使用的优点相结合，旨在提供专门面向电动机控制、汽车、电源管理、嵌入式音频和工业自动化市场等新兴应用的灵活解决方案。

Cortex - M7 处理器目前是高性能 Cortex - M 系列处理器中具有最高性能的成员之一，它能构建各种复杂的微控制器与嵌入式芯片。Cortex - M7 的设计旨在提供超高性能，并保持 ARMv7 - M 架构卓越的响应性和易用性。它拥有业内领先的高性能和灵活的系统接口，是各种应用领域的理想之选。

1.3.4 Cortex - R 系列处理器

Cortex - R 系列处理器面向深层嵌入式系统以及实时嵌入式市场。它满足汽车安全、存储或无线基带等领域所要求的高性能、实时性、可靠性以及高性价比等方面的需求。

Cortex - R 系列处理器主要包括 Cortex - R4、Cortex - R5、Cortex - R7、Cortex - R8 等处理器。该系列处理器的主要特点如图 1-3 所示。

试读版本仅供防疫期间线上教学使用

图 1-3 Cortex - R 系列处理器主要特点

Cortex－R4 处理器是第一个基于 ARMv7－R 架构的深层嵌入式实时处理器。它专用于大容量深层嵌入式片上系统应用，如硬盘驱动控制器、无线基带处理器、消费性产品、手机 MTK 平台和汽车系统的电子控制单元。

Cortex－R5 处理器为市场上的实时应用提供高性能的解决方案，包括移动基带、汽车、大容量存储、工业和医疗市场。该处理器基于 ARMv7－R 架构，因此提供了一种从 Cortex－R4 处理器上移植到更高性能的 Cortex－R7 处理器的简单途径。

Cortex－R7 处理器为范围广泛的深层嵌入式应用提供了高性能的双核、实时解决方案。Cortex－R7 处理器通过引入新技术（包括无序执行和动态寄存器重命名），并与改进的分支预测、超标量执行功能和用于除法等功能的硬件支持相结合来达到实时的目的。

目前，Cortex－R8 处理器是同系列中即时反应最快的处理器，主要采用四核架构规格，并且着重对应 5G 联网数据机的设计应用以及巨量储存装置设计需求。该处理器通过低延迟、高效能与低功耗等特性满足 5G 连网、物联网与储存应用。同时，其即时性更好，可提高汽车制动性能、增进安全性，在汽车电子领域有很好的应用前景。与两核的 Cortex－R7 处理器相比，该处理器为四核，故可执行多重指令，且表现性能提高两倍，并具有侦错/除错功能。此外，其紧密耦合记忆体（TCM）提高至每核 2MB，以达到更好的即时性。

1.3.5　SecurCore 系列处理器

SecurCore 系列处理器基于行业领先的 ARM 架构，提供功能强大的 32 位安全解决方案。通过用各种安全功能来加强已十分成功的 ARM 处理器，SecurCore 推出了智能卡，使从事安全类工作的 IC 开发人员可以方便地利用 ARM 32 位技术的优点（例如晶片尺寸小、能效高、成本低、代码密度优异且性能十分突出）。SecurCore 系列处理器的性能超越了旧的 8 位或 16 位安全处理器。SecurCore 系列处理器主要包括 SC000、SC100 和 SC300 等。

SecurCore 系列处理器主要是为防篡改智能卡而设计的，由于具有多种安全功能，其非常适合此类应用。有关其安全功能的详细信息，请参阅 ARM 提供的保密协议（NDA）。其主要功能是：

1）用于简易实现的完全可合成设计。

2）广泛的工具支持：通过 RealView 微控制器开发工具包（Keil μVision 环境）这个受业内欢迎的智能卡开发工具来提供全面支持。

3）连接到标准系统 IP：AMBA 互连兼容性可用来通过外设和内存实现快速高效的系统设计。

SecurCore 系列处理器与其他处理器相比在安全领域有诸多优点：

1）可通过极小的可合成处理器实现 32 位处理。

2）更高的系统性能。

3）上市速度更快。

4）大量可用的软件开发工具。

5）开发成本较低。

6）可从行业领先的芯片供应商处得到供货。

7）向 SoC 设计者提供理想的调试支持。

1.4 ARM处理器的芯片选型

近年来嵌入式系统飞速发展，ARM处理器获得了越来越多的关注，同时也在更多领域获得了广泛的应用。ARM处理器的内核结构有数十种，ARM公司的chipless生产模式会使得将来获得授权的生产厂家越来越多（单就目前来说，获得授权的芯片生产厂家就有70多家），而且每个厂家根据需要对内核功能进行的配置又各不相同。所以对于开发人员，完成好芯片选型有一定难度。

芯片选型需要根据项目需求考虑众多的因素，不单是功能和性能，成本往往也是一个重要方面。本节主要从以下几个方面介绍选型需要考虑的技术因素，用户在实际项目中应根据需要从各方面均衡考虑。

1. 功能

考虑处理器本身能够支持的功能，比如USB、网络、串口、液晶显示等。

2. 性能

从处理器的功耗、速度、稳定可靠性等方面考虑。随着便携式产品的快速发展，功耗越来越引起关注，甚至直接关系到便携式产品的销量。

3. 熟悉程度及开发资源

通常公司对产品的开发周期都有严格的要求，选择一款自己熟悉的处理器可以大大降低开发风险。在自己熟悉的处理器都无法满足功能的情况下，可以尽量选择开发资源丰富的处理器。

在开发资源方面需要考虑处理器的自带资源和可扩展资源两个方面。例如主频，有无内置的以太网MAC，I/O口数量，自带哪些接口，是否支持在线仿真，是否支持OS，能支持哪些OS，是否有外部存储接口等。微处理器自带的资源是选型的一个重要考虑因素，自带资源越接近产品需求，产品开发相对就越简单。

可扩展资源主要考虑硬件平台是否支持OS、RAM和ROM。芯片一般需要内置RAM和ROM，但其容量一般都很小，内置512KB就算很大了。若运行OS一般都需要兆级以上容量，这就要求芯片带有可扩展存储器。

4. 封装

常见的微处理器芯片封装主要有QFP、BGA两大类型。BGA类型的封装焊接比较麻烦，一般小公司都不支持，但是BGA封装芯片体积会小很多。如果产品对芯片体积要求不严格，选型时最好选择QFP封装。

5. 操作系统

在选择处理器时，如果最终的程序需要运行在操作系统上，那么还应该考虑处理器对操作系统的支持。

6. 仿真器

仿真器是硬件底层软件调试时要用到的工具，开发初期如果没有它，基本上会寸步难行。选择配套合适的仿真器，将会给开发带来许多便利。对于已经有仿真器的情况，在选型

该版本仅供防疫期间线上教学使用

过程中要考虑它是否支持所选芯片。

7. 升级与维护

很多产品在开发完成后都会面临升级的问题，所以在选择处理器时必须加以考虑。如尽量选择具有相同封装的不同性能等级的处理器，并考虑产品未来可能增加的功能。

8. 价格

通常产品总是希望在完成功能要求的基础上尽量降低成本。在选择处理器时需要考虑处理器的价格以及由处理器衍生出的开发价格，如开发板价格、处理器自身价格、外围芯片价格、开发工具价格、制板价格等。

9. 供货

供货稳定也是选择处理器时的一个重要参考因素，尽量选择大厂家的产品和比较通用的芯片。

本章小结

本章系统地介绍了 ARM 公司及其处理器的发展历史、现状以及未来发展趋势，详细介绍了 ARM 公司的主流处理器架构和特征。ARM 公司的主流处理器架构包括从 ARMv4 到 ARMv8 等主要版本。在 1.2 节深入讲解了各个处理器所应用的版本架构及发展。ARM 公司的主流处理器包括 Classic ARM 处理器、Cortex－A 系列处理器、Cortex－M 系列处理器、Cortex－R 系列处理器以及 SecurCore 系列处理器等。在 1.3 节分别从基本技术特点、应用领域以及市场应用情况对其进行了详细阐述。对于种类繁多的处理器，选型是一个重要且棘手的问题，所以在本章的最后给出了一些选型中应考虑的基本因素。通过本章的学习，学生将对 ARM 公司主流处理器及其架构有一个宏观的了解。

思考题

1. ARM 公司的主流处理器有哪些？
2. Cortex 系列处理器分为哪 3 类，主要应用在哪些领域？
3. ARM 处理器架构有哪些版本？
4. 指出 Cortex 系列处理器主要基于哪两种架构？各有什么特点？
5. 处理器选型需要考虑的技术因素有哪些？

试读版本仅供防疫期间线上教学使用

第 2 章　ARM 处理器体系结构

ARM 处理器体系结构是 ARM 区别于其他类型处理器的最基本特征之一，是学习 ARM 处理器编程的基础。本章主要介绍 ARM 处理器的数据类型、处理器工作模式和处理器寄存器组织等 ARM Cortex－A 系列处理器所共有的体系结构特征，在此基础上，进一步介绍隶属于 ARM Cortex－A 架构的 ARM Cortex－A9 的处理器内核。

2.1　数据类型

ARM 采用的是 32 位架构，ARM 的基本数据类型有以下 3 种：

1）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）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（字必须与 4 字节的边界对准）。

3）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（半字必须与 2 字节的边界对准）。

存储器可以看作线性字节阵列。如图 2-1 所示为 ARM 的基本数据类型，其中每一个字节都有唯一的地址。字节可以占用任一位置，图中给出了几个例子。长度为 1 个字的数据项占用一组 4 字节的位置，该位置开始于 4 的倍数的字节地址（地址最末两位为 0）。长度为半字的数据占有 2 字节的位置，该位置开始于偶数字节地址（地址最末一位为 0）。

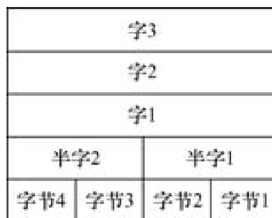

图 2-1　ARM 的基本数据类型

2.2　处理器工作模式

为了提高处理器效能，ARM 处理器通过多种的处理器模式来管理处理器，Cortex－A9 处理器属于 ARMv7－A 架构，共有 8 种工作模式：

1）用户模式（User）：正常程序执行模式，大部分任务执行在这种模式下。

2）快速中断模式（FIQ）：当一个快速中断产生时将会进入这种模式，一般用于高速数

试读版本仅供防疫期间线上教学使用

据传输和通道处理。

3）中断模式（IRQ）：当一个一般中断产生时将会进入这种模式，一般用于通常的中断处理。

4）管理模式（Supervisor）：当复位或软中断指令执行时进入这种模式，是一种供操作系统使用的保护模式。

5）中止模式（Abort）：当数据或指令存取异常时进入这种模式，用于虚拟存储或存储保护。

6）未定义模式（Undef）：当执行未定义指令时进入这种模式，通常用于通过软件来仿真硬件协处理器的工作方式。

7）系统模式（System）：使用和 User 模式相同寄存器集的模式，用于运行特权级的操作系统任务。

8）监控模式（Monitor）：可以在安全模式与非安全模式之间切换。

除用户模式外的其他 7 种处理器模式称为特权模式。在特权模式下，程序可以访问所有的系统资源，可以进行处理器模式切换。除系统模式外的 6 种特权模式又称为异常模式。

处理器模式可以通过软件控制进行切换，也可以通过外部中断或异常处理过程进行切换。大多数的用户程序运行在用户模式下。当处理器工作在用户模式时，应用程序既不能访问受操作系统保护的一些系统资源，也不能直接进行处理器模式切换。当需要时，应用程序可以产生异常处理，在异常处理过程中进行处理器模式切换。

当应用程序发生异常中断时，处理器进入相应的异常模式。在每一种异常模式中都有一组专用寄存器以供相应的异常处理程序使用，通常是用程序状态备份寄存器、堆栈指针寄存器和链接寄存器来保护处理器的工作状态和异常返回地址，这样就可以保证在进入相应异常模式时用户模式下的寄存器不被破坏。

2.3　ARM 处理器的存储系统

2.3.1　存储空间

ARM 体系结构使用 2^{32} 个字节的单一、线性地址空间。这些字节单元的地址是一个无符号的 32 位数值，其取值范围为 $0\sim2^{32}-1$。

ARM 的地址空间可以看作由 2^{30} 个 32 位的字组成。每个字的地址是字对齐的，因此字的地址可被 4 整除。字对齐地址为 A 的字由地址为 A、A+1、A+2 和 A+3 的 4 个字节组成。

ARM 的地址空间也可以看作由 2^{31} 个 16 位的半字组成的。每个半字的地址是半字对齐的，因此半字的地址可被 2 整除。半字对齐地址为 A 的半字由地址为 A 和 A+1 的 2 字节组成。

各存储单元的地址作为 32 位的无符号数，可以进行常规的整数运算。这些运算的结果将进行 2^{32} 取模。也就是说，运算结果发生上溢出和下溢出时，地址将会发生卷绕。

2.3.2　存储格式

ARM 体系结构所支持的最大寻址空间为 4GB（2^{32} 字节），支持两种处理器存储格式：

试读版本仅供防疫期间线上教学使用

1）大端格式（Big Endian）：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。

2）小端格式（Little Endian）：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。

下面的例子显示了使用内存的大/小端的存取格式。

程序执行前

r0 = 0x11223344，r1 = 0x100

执行指令

STR r0，[r1]；将一个 r0 中的字数据存储到 r1 中地址所对应的存储单元，一次传送 4 个字节。

LDRB r2，[r1]；加载 r1 中地址所对应的存储单元中一个字节的数据到 r2。

执行后，小端模式下 r2 = 0x44，大端模式下 r2 = 0x11。

上面的例子表明，在大端模式下，一个字的高地址放的是数据的低位，而在小端模式下，数据的低位放在内存中的低地址。

2.3.3 存储管理单元

存储管理单元（Memory Management Unit，MMU）是处理器用来管理虚拟存储器、物理存储器的控制模块，同时也负责将虚拟地址映射为物理地址，并提供硬件机制的内存访问授权。

对于操作系统来说，MMU 提供的一个关键服务是使各个任务作为各自独立的程序在自己的私有存储空间中运行。在带 MMU 的操作系统的控制下，运行的任务无需知道其他与之无关的任务的存储需求情况，这就简化了各个任务的设计。

MMU 提供了一些资源以允许使用虚拟存储器（将系统物理存储器重新编址，可将其看成一个独立于系统物理存储器的存储空间）。MMU 作为转换器，将程序和数据的虚拟地址（编译时的连接地址）转换成实际的物理地址，即在物理主存中的地址。这个转换过程允许运行的多个程序使用相同的虚拟地址，而这些程序各自存储在物理存储器的不同位置。

在 MMU 的处理下，存储器有两种类型的地址：虚拟地址和物理地址。虚拟地址由编译器和连接器在定位程序时分配；物理地址用来访问实际的硬件存储模块。

2.3.4 高速缓冲存储器

高速缓冲存储器（Cache）是存在于主存与 CPU 之间的一级存储器，由静态存储芯片（SRAM）组成，容量比较小但存取速度比主存高得多，接近于 CPU 的速度。高速缓冲存储器和主存储器之间信息的调度和传送是由硬件自动进行的。

Cache 保存最近用到的存储器数据副本。对于程序员来说，Cache 是透明的，它自动决定保存哪些数据、覆盖哪些数据。Cache 经常与写缓存器（Write Buffer）一起使用。写缓存器是一个非常小的先进先出（FIFO）存储器，位于处理器核与主存之间。使用写缓存的目的是将处理器核和 Cache 从较慢的主存写操作中解脱出来。当 CPU 向主存储器做写入操作时，它先将数据写入到写缓存器中，由于写缓存器的速度很快，这种写入操作的速度也会很快。写缓存器在 CPU 空闲时，以较低的速度将数据写入到主存储器中相应的位置。

试读版本仅供防疫期间线上教学使用

2.3.5　协处理器

ARM 处理器支持 16 个协处理器。在程序执行过程中，每个协处理器忽略属于 ARM 处理器和其他协处理器的指令。当一个协处理器硬件不能执行属于它的协处理器指令时，将产生一个未定义指令的异常中断，在该异常中断处理程序中，可以通过软件模拟该硬件操作。例如，如果系统不包含向量浮点运算器，则可以选择浮点运算软件模拟包来支持向量浮点运算。

CP15 即通常所说的系统控制协处理器（System Control Coprocessor）。在 ARM 处理器中，诸如 Cache 配置、写缓存配置之类的存储系统管理工作均由 CP15 完成。如果处理器核中不存在 CP15，针对 CP15 的操作指令将被视为未定义指令，指令的执行结果不可预知。

CP15 包含 16 个 32 位寄存器，其编号为 0～15。实际上某些编号的寄存器可能对应多个物理寄存器，在指令中指定特定的标志位来区分这些物理寄存器。这种机制有些类似于 ARM 中的寄存器，当处于不同的处理器模式时，某些相同编号的寄存器对应于不同的物理寄存器。

CP15 中的寄存器可能是只读的，也可能是只写的，还有一些是可读可写的。在对协处理器寄存器进行操作时，需要注意以下几个问题：

1）寄存器的访问类型（只读/只写/可读可写）。

2）不同的访问引发不同的功能。

3）相同编号的寄存器是否对应不同的物理寄存器。

4）寄存器的具体作用。

2.4　寄存器组织

ARM 处理器作为 RISC 型处理器之一，其最主要的特性之一就是 ARM 核内部使用了大量的寄存器，用于提升处理器的运算速度，提高处理器的性能。ARM 处理器有 40 个 32 位的寄存器，其中包含 33 个通用寄存器和 7 个状态寄存器。这些寄存器不能被同时访问，处理器所处的工作模式决定哪些处理器可以被编程者使用。本节将在介绍 ARM 处理器模式下寄存器分布的基础上，分别对通用寄存器和状态寄存器的使用特点进行详细说明。

2.4.1　ARM 处理器模式下的寄存器分布

对于 ARM 核的众多寄存器的使用，可以从不同 ARM 处理模式下的寄存器分布来了解。ARM 处理器共有 8 种不同的处理器模式，在每一种处理器模式中都有一组相应的寄存器。ARM 处理器将寄存器分成很多个组，来运行不同模式下的程序，让 CPU 程序运行得更加稳健。表 2-1 是 ARM 寄存器分组：共 User、System、Supervisor、Abort Undefined、IRQ、FIQ 和 Secure Monitor 8 个组。其中 System 和 User 两个组的寄存器完全共用，Secure Monitor 是 ARM Cortex－A9 内核追加的模式。

试读版本仅供防疫期间线上教学使用

表 2-1 ARM 状态下的寄存器分组

用户模式（User）	系统模式（System）	管理模式（Supervisor）	数据访问中止模式（Abort）	未定义指令中止模式（Undefined）	外部中断模式（IRQ）	快速中断模式（FIQ）	安全监控模式（Secure Monitor）
R0	R0	R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7	R7	R7
R8	R8	R8	R8	R8	R8	R8_fiq	R8
R9	R9	R9	R9	R9	R9	R9_fiq	R9
R10	R10	R10	R10	R10	R10	R10_fiq	R10
R11	R11	R11	R11	R11	R11	R11_fiq	R11
R12	R12	R12	R12	R12	R12	R12_fiq	R12
R13	R13	R13_svc	R13_abt	R13_und	R13_irq	R13_fiq	R13_mon
R14	R14	R14_svc	R14_abt	R14_und	R14_irq	R14_fiq	R14_mon
PC	PC	PC	PC	PC	PC	PC	PC
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
		SPSR_svc	SPSR_abt	SPSR_und	SPSR_irq	SPSR_fiq	SPSR_mon

仅供防疫期间线上教学使用

表 2-2 为 8 种处理器模式各自拥有和可访问的寄存器。每种处理模式使用的寄存器包含两大类：一类为公共部分，即所有模式使用的寄存器是相同物理寄存器，这部分寄存器在使用时应作为环境变量，进行使用前保存和使用后还原处理，以避免影响其他模式的运算处理；另一类为私有部分，这部分寄存器为特定模式专用，可按照模式本身的需要进行处理，不需要另外进行环境变量的保存和还原操作，其主要是为加速处理器运行，特别是为减少处理器异常中断的响应时间而存在的。其中 R0 ~ R15 和 CPSR 这 17 个寄存器是各个组共用，特别是 R0 ~ R7 是每个组都共用的。但由于其通用性，在异常中断所引起的处理器模式切换时，使用的是相同的物理寄存器，所以也就很容易使寄存器中的数据被破坏。每种异常模式有自己独立的堆栈指针（SP_mode）、链接寄存器（LR_mode）和程序状态备份寄存器（SPSR_mode）。另外 FIQ 有独立的 R8_fiq ~ R14_fiq，用于高速的数据处理。许多寄存器都有专门用途，并可以使用别名来操作：

1）R9（SB）：静态基地址寄存器。

2）R10（SL）：数据堆栈限制指针。

3）R11（FP）：帧指针 Frame Pointer。

4）R12（IP）：子程序内部调用暂存寄存器 Intra – Procedure – call Scratch Register。

5）R13（SP）：堆栈指针 Stack Pointer。

6）R14（LR）：链接寄存器 linker Register，用于程序跳转后返回（LR 寄存器的值赋给 PC 寄存器）。

7）R15（PC）：程序计数器 Program Counter，用于指向程序执行代码。

表 2-2 8 种处理器模式各自拥有和可访问的寄存器

组 名 称	可见寄存器	
	公共部分	私有部分
User	R0 ~ R15，CPSR，PC	无
System	R0 ~ R15，CPSR，PC	无
Supervisor	R0 ~ R12，CPSR，PC	R13_svc，R14_svc，SPSR_svc
Abort	R0 ~ R12，CPSR，PC	R13_abt，R14_abt，SPSR_abt
Undefined	R0 ~ R12，CPSR，PC	R13_und，R14_und，SPSR_und
IRQ	R0 ~ R12，CPSR，PC	R13_irq，R14virq，SPSR_irq
FIQ	R0 ~ R7，CPSR，PC	R8_fiq – R14_fiq，SPSR_fiq
Secure Monitor	R0 ~ R12，CPSR，PC	R13_mon，R14_mon，SPSR_mon

2.4.2 通用寄存器

通用寄存器（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）。

1）未分组寄存器 R0 ~ R7。对于每一个未分组寄存器来说，在所有的处理器模式下指的都是同一个物理寄存器。在异常中断造成处理器模式切换时，由于不同的处理器模式使用相同的物理寄存器，可能造成未分组寄存器中数据被破坏。任何可采用通用寄存器的应用场合都可以使用未分组寄存器。

2）分组寄存器 R8 ~ R14。分组寄存器是指同一个寄存器名。在 ARM 微处理器内部存在多个独立的物理寄存器，每一个物理寄存器分别与不同的处理器模式相对应。

3）程序计数器 PC（R15）。寄存器 R15 用作程序计数器 PC。在 ARM 状态下，由于 ARM 指令是字对齐的，所以 PC 的第 0 位和第 1 位总为 0；在 Thumb 状态下，PC 的第 0 位是 0。PC 虽然可以作为一般的通用寄存器使用，但是一些指令在使用 R15 时有特殊限制。当违反了这些限制时，该指令执行的结果将是不可预料的。

由于 ARM 体系结构采用了流水线机制（以三级流水线为例），对于 ARM 指令集来说，PC 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。

分组的寄存器 R8 ~ R14 中的 R13 和 R14 作为堆栈指针寄存器和链接寄存器，在程序设计中，特别是汇编指令的使用中，通常是默认使用的。其主要的使用方法和特点如下：

（1）堆栈指针寄存器

寄存器 R13_<mode>又被称为堆栈指针寄存器（Stack Pointer，SP），其中<mode>可以是以下几种之一：usr、svc、abt、und、irp、fiq 及 mon。在 ARM 处理器中，R13 常用做堆栈指针，当然，这只是一种习惯用法，并没有任何指令强制性地使用 R13 作为堆栈指针，

版本仅供防疫期间线上教学使用

用户完全可以使用其他寄存器作为堆栈指针。而在 Thumb 指令集中，有一些指令强制性地将 R13 作为堆栈指针，如堆栈操作指令。每一种异常模式拥有自己的 R13。异常处理程序负责初始化自己的 R13，使其指向该异常模式专用的栈地址。在异常处理程序入口处，将用到的其他寄存器的值保存在堆栈中，返回时，重新将这些值加载到寄存器。通过这种保护程序现场的方法，异常处理程序不会破坏被其中断的程序现场。

（2）链接寄存器

寄存器 R14_<mode>又被称为链接寄存器（Link Register，LR），其中<mode>可以是以下几种之一：usr、svc、abt、und、irp、fiq 及 mon。在 ARM 体系结构中具有下面两种特殊的作用：每一种处理器模式用自己的 R14 存放当前子程序的返回地址，当通过 BL 或 BLX 指令调用子程序时，R14 被设置成该子程序的返回地址；在子程序返回时，把 R14 的值复制到 PC。典型的做法如下：

1）执行下面任何一条指令。

```
MOV PC，LR
BX LR
```

2）在子程序入口处使用下面的指令将 PC 保存到堆栈中。

```
STMFD SP!，{<register>,LR}
```

在子程序返回时，使用如下相应的配套指令返回。

```
LDMFD SP!，{<register>,PC}
```

3）当异常中断发生时，该异常模式特定的物理寄存器 R14 被设置成该异常模式的返回地址。对于有些模式，R14 的值可能与返回地址有一个常数的偏移量（如数据异常，使用 SUB PC，LR，#8 返回）。具体的返回方式与上面的子程序返回方式基本相同，但使用的指令稍微有些不同，以保证当异常出现时正在执行的程序状态被完整保存。R14 也可以用作通用寄存器。

2.4.3 程序状态寄存器

ARM 体系结构包含 1 个当前程序状态寄存器（Current Program Status Register，CPSR）和 6 个备份的程序状态寄存器（Saved Program Status Register，SPSR）。

当前程序状态寄存器可以在任何处理器模式下被访问，它记录和指示出当前程序运行状态，包含下列内容：

1）算术逻辑单元（Arithmetic Logic Unit，ALU）状态标志的备份。

2）当前的处理器模式。

3）中断使能标志。

4）设置处理器的状态。

每一种处理器模式下都有 1 个专用的物理寄存器做备份程序状态寄存器。当特定的异常中断发生时，这个物理寄存器负责存放当前程序状态寄存器的内容。当异常处理程序返回时，再将其内容恢复到当前程序状态寄存器。

CPSR 寄存器（和保存它的 SPSR 寄存器）中的位分配见表 2-3。

表 2-3　CPSR 寄存器（和保存它的 SPSR 寄存器）中的位分配

31	30	29	28	27	26～25	24	23～20	19～16	15～10	9	8	7	6	5	4～0
N	Z	C	V	Q	IT［1：0］	J	保留	GE［3：0］	IT［7：2］	E	A	I	F	T	M［4：0］

（1）条件状态位

N、Z、C 和 V 统称为条件码标志位。其内容可被算术和逻辑运算的结果所改变，由此可以决定某些指令是否被执行。CPSR 中的条件码标志位见表 2-4。

表 2-4　CPSR 中的条件码标志位

条件码标志位	含　　义
N	本位设置成当前指令运算结果的第 31 位值 当两个补码表示的有符号数进行运算时，N＝1，表示运算的结果为负数；N＝0，表示运算的结果为正数
Z	Z＝1 表示运算的结果为零；Z＝0 表示运算的结果不为零 对于 CMP 指令，Z＝1 表示进行比较的两个数大小相等
C	有 4 种方法设置 C 的值： 1）在加法指令中（包括比较指令 CMN），当结果产生进位（无符号数溢出），则 C＝1；否则 C＝0 2）在减法指令中（包括比较指令 CMP），当运算中发生借位（无符号数溢出），则 C＝0；否则 C＝1 3）对于包含移位操作的非加/减运算指令，C 为移出位的最后一位 4）对于其他的非加/减运算指令，C 的值通常不改变
V	对于加/减运算指令，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

在 ARM 指令的使用中，可以使用 2 个字母的助记符来使用 4 个条件状态位，而不用考虑条件状态位的具体值，助记符和程序状态寄存器标志位之间的关系见表 2-5。

表 2-5　助记符和程序状态寄存器标志位之间的关系

指令中的条件编码	助记符	PSR 中标志位	用意
0000	EQ	Z＝1	相等
0001	NE	Z＝0	不相等
0010	CS	C＝1	>＝(无符号数)
0011	CC	C＝1	<(无符号数)
0100	MI	N＝1	负数
0101	PL	N＝0	正数或为零
0110	VS	V＝1	溢出
0111	VC	V＝0	未溢出
1000	HI	C＝1，且 Z＝0	>(无符号数)
1001	LS	C＝0，且 Z＝1	<＝(无符号数)
1010	GE	N＝V	>＝(带符号数)
1011	LT	N!＝V	<(带符号数)
1100	GT	Z＝0，且 Z＝＝V	>(带符号数)
1101	LE	Z＝1，或 N!＝V	<＝(带符号数)
1110	AL	忽略	无条件执行

试读版本仅供防疫期间线上教学使用

（2）饱和计算溢出标志位

Q：Sticky overflow（有符号运算，确保数的极性不会翻转），发生溢出时，该位置 1。

（3）IT 和 GE 状态位

IT［7：2］：用于 IF…ELSE…条件执行状态位。

GE［3：0］：由一些 SIMD 指令使用。

（4）使能控制位

A：当 A =1 时，禁止不确定数据异常。

I：IRQ 中断使能位，当其置 1 时，禁止处理器响应 IRQ 中断；当其置 0 时允许处理器响应 IRQ 中断。

F：FIQ 中断使能位，当其置 1 时，禁止处理器响应 FIQ 中断；当其置 0 时允许处理器响应 FIQ 中断。

（5）指令类别状态位

J：当 T 为 1 时，J =0 表示执行 Thumb 指令状态；J =1 表示执行 ThumbEE 指令状态。

T：当其为 1 时表示执行 Thumb 指令状态（或 ThumbEE，这依赖于 J 标志，J 为 1，表示执行 ThumbEE 指令状态），当其为 0 时表示执行 ARM 指令状态。

指令类别状态位见表 2-6。

表 2-6　指令类别状态位

J	T	指令状态
0	0	ARM
0	1	Thumb
1	0	Jazelle
1	1	ThumbEE

（6）工作模式位

工作模式位用来指示不同组寄存器与 ALU 协同工作。

M［4：0］：工作模式。

B10000（0x10）：用户模式（User）。

B10001（0x11）：快速中断模式（FIQ）。

B10010（0x12）：向量中断模式（IRQ）。

B10011（0x13）：管理模式（Supervisor）。

B10111（0x17）：异常模式（Abort），取数据失败时发生。

B11011（0x1B）：未定义模式（Undefined），指令解码出错时发生。

B11111（0x1F）：系统模式（System）。

B10110（0x16）：安全监视模式（Secure Monitor），主要用于安全交易。

（7）大端和小端存储标志位

E：大端和小端存储标志，当 E =0 时为小端存储，E =1 时为大端存储。因为 ARM 总线都为 32bit（4 个字节）以上的，而数据的存取常规以 8bit 为单位，这样就会出现如何放置数据的问题。规定数据高位放在高字节、低位放在低字节的称为小端存储；反之，称为大端存储。

试读版本仅供防疫期间线上教学使用

2.5　异常处理

在 ARM 体系中通常有以下 3 种方式控制程序的执行流程。

1）在正常程序执行过程中，每执行一条 ARM 指令，程序计数器 PC 的值加 4 个字节；每执行一条 Thumb 指令，程序计数器 PC 的值加 2 个字节。整个过程是按顺序执行的。

2）通过跳转指令，程序可以跳转到特定的地址标号处执行，或者跳转到特定的子程序处执行。其中，B 指令用于执行跳转操作；BL 指令在执行跳转操作的同时，保存子程序的返回地址；BX 指令在执行跳转操作的同时，根据目标地址的最低位可以将程序切换到 Thumb 状态；BLX 指令执行 3 个操作，跳转到目标地址处执行，保存子程序的返回地址，根据目标地址的最低位可以将程序状态切换到 Thumb 状态。

3）当异常中断发生时，系统执行完当前指令后，将跳转到相应的异常中断处理程序处执行。在当异常中断处理程序执行完成后，程序返回到发生中断的指令的下一条指令处执行。在进入异常中断处理程序时，要保存被中断的程序的执行现场。在从异常中断处理程序退出时，要恢复被中断的程序的执行现场。

ARM 处理器的异常机制类似传统处理器的中断处理机制，它反映了一个处理器处理突发事件的实时性能。当某一异常发生时，ARM 处理器打断当前正在执行的程序，处理器自动到该异常相对应的固定地址读取一条 ARM 指令并执行，该 ARM 指令通常为一条跳转指令，处理器通过该跳转指令跳转到程序员设计的异常程序并执行。当异常程序执行完成之后，程序员必须借助相应的处理器寄存器，编写代码返回到被异常打断的代码继续往下执行，并能还原被异常打断之前的处理器模式和通用寄存器值等环境变量。

2.5.1　ARM 处理器异常类型

异常可以通过内部或者外部源产生，并引起处理器处理一个事件，例如外部中断或者试图执行未定义指令都会引起异常。在处理异常之前，处理器的状态必须保存，以便在处理异常完成后，原来的程序能够重新继续执行。

ARM 体系结构支持 7 种异常，见表 2-7。

表 2-7　ARM 体系结构支持的异常类型

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
外部中断请求 IRQ	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
快速中断请求 FIQ	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

试读版本仅供防疫期间线上教学使用

异常模式除了可以通过程序切换进入外，也可以通过特定的异常类型触发，当特定的异常类型出现时，处理器进入相应的异常模式。表 2-8 为异常类型对应的异常向量地址和处理器模式。当异常发生时，处理器会把 PC 设置为一个特定的存储器地址。这一地址放在被称为向量表（vector table）的特定地址范围内。向量表的入口是一些跳转指令，跳转到专门处理某个异常或中断的子程序。存储器映射地址 0x00000000 是为向量表（一组 32 位字）保留的。在有些处理器中，向量表可以选择定位在存储空间的高地址（从偏移量 0xffff0000 开始）。一些嵌入式操作系统，如 Linux 和 Windows CE 就利用了这一特性。

注意：Cortex－A8/A9 系统中支持通过设置 CP15 的 C12 寄存器将异常向量表的首地址设置在 32 字节对齐的任意地址。为了保持继承，下文仍沿用传统的 0x0 低地址方式 和 0xFFFF0000 高地址方式。

表 2-8　异常类型对应的异常向量地址和处理器模式

异常类型	异常模式	低地址异常向量	高地址异常向量
复位	管理	0x00000000	0xFFFF0000
未定义指令	未定义	0x00000004	0xFFFF0004
软件中断（SWI）	管理	0x00000008	0xFFFF0008
预取中止（取指令存储器中止）	中止	0x0000000C	0xFFFF000C
数据中止（数据访问存储器中止）	中止	0x00000010	0xFFFF0010
IRQ（中断）	IRQ	0x00000018	0xFFFF0018
FIQ（快速中断）	FIQ	0x0000001C	0xFFFF001C

2.5.2　ARM 异常处理

1. ARM 内核对异常的响应

当任何一个异常发生并得到响应时，ARM 内核自动完成以下动作：

1）将下一条指令的地址存入相应链接寄存器 LR，以便程序在异常处理完成后能从正确的位置重新开始执行。

2）将 CPSR 的值复制到相应的 SPSR 中。

3）设置适当的 CPSR 位，包括改变处理器状态进入 ARM 状态，改变处理器模式进入相应的异常模式，设置中断禁止位禁止相应中断。

4）设置 PC 使其从相应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。

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            /* 在 ARM 状态执行 */
If <Exception_Mode> == Reset or FIQ then
CPSR[6] = 1            /* 禁止快速中断 */
```

读者版本仅供防疫期间线上教学使用

```
/* 否则 CPSR[6]不变 */
CPSR[7] =1
PC = Exception Vector Address
```

当处理器发生复位异常时，系统进入管理模式，切换到 ARM 状态，同时禁止 FIQ 和 IRQ 中断，然后设置 PC 使其从复位向量地址 0x00000000（或者 0xFFFF0000）取下一条指令执行，伪代码描述如下：

```
R14_svc = UNPREDICTABLE value
SPSR_svc = UNPREDICTABLE value
CPSR[4:0] =0b10011          /* 进入管理模式 */
CPSR[5] =0                  /* 在 ARM 状态执行 */
CPSR[6] =1                  /* 禁止快速中断 */
CPSR[7] =1                  /* 禁止正常中断 */
If high vectors configured then
    PC =0xFFFF0000
else
    PC =0x00000000
```

当处理器发生未定义指令异常时，系统将下一条指令的地址存入 R14_und，同时将 CPSR 的值复制到 SPSR_und 中；然后强制设置 CPSR 的值，使系统进入未定义模式，同时切换到 ARM 状态；设置 CPSR 的 I 位为 1，用来禁止 IRQ 中断；最后设置 PC 使其从未定义向量地址 0x00000004（或者 0xFFFF0004）取下一条指令执行。伪代码描述如下：

```
R14_und = address of next instruction after the undefined instruction
SPSR_und = CPSR
CPSR[4:0] =0b11011          /* 进入未定义模式 */
CPSR[5] =0                  /* 在 ARM 状态执行 */
                            /* CPSR[6]不变 */
CPSR[7] =1                  /* 禁止正常中断 */
If high vectors configured then
    PC =0xFFFF0004
else
    PC =0x00000004
```

当处理器发生软件中断时，系统将下一条指令的地址存入 R14_svc，同时将 CPSR 的值复制到 SPSR_svc 中；然后强制设置 CPSR 的值，使系统进入管理模式，同时切换到 ARM 状态；设置 CPSR 的 I 位为 1，用来禁止 IRQ 中断；最后设置 PC 使其从软件中断向量地址 0x00000008（或者 0xFFFF0008）取下一条指令执行。伪代码描述如下：

```
R14_svc = address of next instruction after the SWI instruction
SPSR_svc = CPSR
CPSR[4:0] =0b10011          /* 进入管理模式 */
```

试读版本仅供防疫期间线上教学使用

```
CPSR[5] = 0                    /* 在 ARM 状态执行 */
                               /* CPSR[6]不变 */
CPSR[7] = 1                    /* 禁止正常中断 */
If high vectors configured then
    PC = 0xFFFF0008
else
    PC = 0x00000008
```

当处理器发生指令预取中止时，系统将下一条指令的地址存入 R14_abt，同时将 CPSR 的值复制到 SPSR_abt 中；然后强制设置 CPSR 的值，使系统进入中止模式，同时切换到 ARM 状态；设置 CPSR 的 I 位为 1，用来禁止 IRQ 中断；最后设置 PC 使其从预取指令中止向量地址 0x0000000C（或者 0xFFFF000C）取下一条指令执行。伪代码描述如下：

```
R14_abt = address of the aborted instruction + 4
SPSR_abt = CPSR
CPSR[4:0] = 0b10111            /* 进入指令预取中止模式 */
CPSR[5] = 0                    /* 在 ARM 状态执行 */
                               /* CPSR[6]不变 */
CPSR[7] = 1                    /* 禁止正常中断 */
If high vectors configured then
    PC = 0xFFFF000C
else
    PC = 0x0000000C
```

当处理器发生数据预取中止时，系统将下一条指令的地址存入 R14_abt，同时将 CPSR 的值复制到 SPSR_abt 中；然后强制设置 CPSR 的值，使系统进入中止模式，同时切换到 ARM 状态；设置 CPSR 的 I 位为 1，用来禁止 IRQ 中断；最后设置 PC 使其从数据中止向量地址 0x00000010（或者 0xFFFF0010）取下一条指令执行。伪代码描述如下：

```
R14_abt = address of the aborted instruction + 8
SPSR_abt = CPSR
CPSR[4:0] = 0b10111            /* 进入中止模式 */
CPSR[5] = 0                    /* 在 ARM 状态执行 */
                               /* CPSR[6]不变 */
CPSR[7] = 1                    /* 禁止正常中断 */
If high vectors configured then
    PC = 0xFFFF0010
else
    PC = 0x00000010
```

当处理器发生 IRQ 异常中断时，系统将下一条指令的地址存入 R14_irq，同时将 CPSR 的值复制到 SPSR_irq 中；然后强制设置 CPSR 的值，使系统进入 IRQ 模式，同时切换到

试读版本仅供防疫期间线上教学使用

ARM 状态；设置 CPSR 的 I 位为 1，用来禁止 IRQ 中断；最后设置 PC 使其从 IRQ 向量地址 0x00000018（或者 0xFFFF0018）取下一条指令执行。伪代码描述如下：

```
R14_irq = address of next instruction to be executed  +  4
SPSR_irq = CPSR
CPSR[4:0] = 0b10010          /* 进入 IRQ 模式 */
CPSR[5] = 0                  /* 在 ARM 状态执行 */
                             /* CPSR[6]不变 */
CPSR[7] = 1                  /* 禁止正常中断 */
If high vectors configured then
    PC = 0xFFFF0018
else
    PC = 0x00000018
```

当处理器发生 FIQ 异常中断时，系统将下一条指令的地址存入 R14_fiq，同时将 CPSR 的值复制到 SPSR_fiq 中；然后强制设置 CPSR 的值，使系统进入 FIQ 模式，同时切换到 ARM 状态；设置 CPSR 的 I 位和 F 位为 1，用来禁止 IRQ 中断和 IRQ 中断；最后设置 PC 使其从 FIQ 向量地址 0x0000001C（或者 0xFFFF001C）取下一条指令执行。伪代码描述如下：

```
R14_fiq = address of next instruction to be executed  +  4
SPSR_fiq = CPSR
CPSR[4:0] = 0b10001          /* 进入 FIQ 模式 */
CPSR[5] = 0                  /* 在 ARM 状态执行 */
CPSR[6] = 1                  /* 禁止快速中断 */
CPSR[7] = 1                  /* 禁止正常中断 */
If high vectors configured then
    PC = 0xFFFF001C
else
    PC = 0x0000001C
```

2. ARM 内核从异常返回

异常处理完毕后，可以通过以下的基本操作完成从异常中断处理程序中返回：

1）将链接寄存器 LR 的值减去相应的偏移量后送到 PC 中。

2）将 SPSR 的值复制回 CPSR 中。

3）清除中断禁止位。

异常返回时非常重要的问题是返回地址的确定。在上面提到进入异常时处理器会有一个保存 LR 的动作，但是该保存值并不一定是正确中断的返回地址。下面以 ARM 状态下三级指令流水线执行示例来对此加以说明，如图 2-2 所示。

在 ARM 体系结构里，PC 值指向当前执行指令的地址加 8 处。也就是说，当执行指令 A（地址 0x8000）时，PC 等于指令 C 的地址（0x8008）。假如指令 A 是“BL”指令，则当执行时，会把 PC（=0x8008）保存到 LR 寄存器里面，但是接下去处理器会马上对 LR 进行一

试读版本仅供防疫期间线上教学使用

图 2-2 ARM 状态下三级指令流水线执行示例

个自动的调整动作：LR = LR－0x4。这样，最终保存在 LR 里面的是 B 指令的地址，所以当从 BL 返回时，LR 里面恰好是正确的返回地址。

同样的调整机制在所有 LR 自动保存操作中都存在，比如进入中断响应时处理器所做的 LR 保存中，也进行了一次自动调整，并且调整动作都是 LR = LR－0x4。

（1）SWI 和未定义指令异常中断返回

SWI 和未定义指令异常中断都是由当前执行的指令自身产生，所以这两种异常返回地址是一致的。下面以 SWI 为例，假设 A（地址 0x8000）为“SWI”指令。当执行指令 A 时，产生 SWI 异常中断，程序计数器 PC 的值还未更新，它指向当前指令后面第 2 条即 C 指令（地址 0x8008），处理器将值（PC－4 = 0x8004）保存到 SWI 模式下 LR 寄存器，正是当前指令的下一条指令 B 处。因此，返回操作可以通过下面的指令来实现：

```
MOV    PC,LR
```

（2）FIQ 和 IRQ 异常中断返回

处理器执行完当前指令后，查询 IRQ 中断引脚及 FIQ 中断引脚，并且查看系统是否允许 IRQ 中断及 FIQ 中断。如果有中断引脚有效，并且系统允许该中断产生，处理器将产生 IRQ 异常中断或 FIQ 异常中断。

假设处理器执行完 A 指令后，查询中断引脚有效，产生 IRQ 或 FIQ 异常中断，当前 PC 值已经更新为 D 指令的地址（0x800C），处理器将值（PC－4 = 0x8008）即 C 指令的地址保存到异常模式下的寄存器 LR 中。要想中断返回时执行 B 指令，可以通过下面的指令来实现：

```
SUBS    PC,LR,#4
```

（3）指令预取中止异常返回

在指令预取时，如果目标地址是非法的，该指令被标记成有问题的指令。这时，流水线上该指令之前的指令继续执行。当执行到该被标记成有问题的指令时，处理器产生指令预取中止异常中断。

当产生指令预取中止异常中断时，程序要返回到该有问题的指令处，重新读取并执行该指令。因此指令预取中止异常中断程序应该返回到产生该指令预取中止异常中断的指令处，而不是像前两种情况下返回到发生异常的指令的下一条指令。

指令预取中止异常中断是由当前执行的指令自身产生的，假设执行指令 A 时，产生指令预取中止异常中断，PC 值还未更新，它指向当前指令后面第 2 条即 C 指令（地址 0x8008），处理器将值（PC－4 = 0x8004）即 B 指令地址，保存到异常模式下的 LR 寄存器。要想中断返回时 PC 指到 A 地址处，可以通过下面的指令来实现：

试读版本仅供防疫期间线上教学使用

```
SUBS    PC,LR,#4
```

（4）数据访问中止异常返回

当发生数据访问中止异常中断时，程序要返回到该有问题的数据访问处，重新访问该数据。因此数据访问中止异常中断程序应该返回到产生该数据访问中止异常中断的指令处。

数据访问中止异常中断是由数据访问指令产生的，假设执行指令 A 时，产生数据访问中止异常中断，PC 的值已经更新为 D 指令的地址（0x800C），处理器将值（PC－4＝0x8008）即 C 指令的地址保存到数据访问中止模式的 LR 中。要想中断返回到产生该数据访问中止的指令 A 处（地址 0x8000），可以通过下面的指令来实现：

```
SUBS PC,LR,#8
```

如果原来的指令执行状态是 Thumb，异常返回地址的分析与此类似，对 LR 的调整正好与 ARM 状态完全一致。

总结 7 种异常返回指令，见表 2-9。

表 2-9　异常返回指令表

异　　常	返 回 指 令
软件中断	MOVS PC，LR　　；R14_svc
未定义指令	MOVS PC，LR　　；R14_und
快速中断 FIQ	SUBS PC，LR，#4　　；R14_fiq
外部中断 IRQ	SUBS PC，LR，#4　　；R14_irq
中止（预取指令）	SUBS PC，LR，#4　　；R14_abt
中止（数据）	SUBS PC，LR，#8　　；R14_abt
复位	NA（不需要返回）

2.5.3　异常优先级

异常可以同时发生，此时处理器按表 2-10 中设置的优先级顺序处理异常。例如，处理器上电时发生复位异常，复位异常的优先级最高，所以当产生复位时，它将优先于其他异常得到处理。同样，当一个数据异常发生时，它将优先于除复位异常外的其他所有异常而得到处理。优先级最低的两种异常是软件中断异常和未定义指令异常。因为正在执行的指令不可能既是一条软中断指令，又是一条未定义指令，所以软中断异常和未定义指令异常享有相同的优先级。

表 2-10　ARM 处理器异常优先级

异 常 类 型	优　先　级
复位	1（最高优先级）
数据中止	2
FIQ	3
IRQ	4
预取中止	5
未定义指令	6
SWI	6（最低优先级）

试读版本仅供防疫期间线上教学使用

2.6 ARM Cortex - A9 内核架构

2.6.1 ARM Cortex - A9 架构简介

ARM Cortex - A9 是性能很高的 ARM 处理器，可实现受到广泛支持的 ARMv7 体系结构的丰富功能。ARM Cortex - A9 的设计旨在打造最先进的、高效率的、长度动态可变的多指令执行超标量体系结构，提供采用乱序猜测方式执行的 8 阶段管道处理器。

ARM Cortex - A9 体系结构既可用于可伸缩的多核处理器（Cortex - A9 MPCore 多核处理器），也可用于更传统的处理器（Cortex - A9 单核处理器）。可伸缩的多核处理器和单核处理器支持 16KB、32KB 或 64KB 4 路关联的 L1 高速缓存配置，对于可选的 L2 高速缓存控制器，最多支持 8MB 的 L2 高速缓存配置，它们具有极高的灵活性，均适用于特定应用领域和市场。

2.6.2 ARM Cortex - A9 单核技术

ARM Cortex - A9 处理器拥有首屈一指的性能和功效，对于要求高性能、低功耗、成本敏感、基于单核处理器的设备，它无疑是理想的解决方案。通过一种方便的综合流程和 IP 交付包，ARM Cortex - A9 为要求更高性能、更高能耗效率的 ARM11 处理器设计提供了一个理想的升级途径，同时还能不增加硅成本和功耗，并保持兼容的软件环境。ARM Cortex - A9 单核处理器为独立指令和数据传输提供两个低延时的 Harvard 64bit AMBA 3 AXITM Master 接口，在通过内存缓存区复制数据时，每 5 个处理器周期能维持 4 次双字写入。ARM Cortex - A9 处理器为包括手机、高端消费类电子和企业产品在内的多种市场应用提供了一种具有可扩展性的解决方案，因为该款处理器满足了以下各项要求：

1）降低功耗、提升功效和性能。

2）提升峰值性能，适应各种要求最为严苛的应用。

3）开发不同设备时可复用软件和工具。

2.6.3 ARM Cortex - A9 多核技术

ARM Cortex - A9 多核处理器是一种设计定制型处理器，以集成缓存一致的方式支持 1 ~ 4 个 CPU。它可单独配置各处理器，设定其缓存大小以及是否支持 FPU、MPE 或 PTM 接口等。此外，无论采用何种配置，处理器都可应用一致性加速口（Accelerator Coherence Port），允许其他无缓冲的系统外设及加速器核与一级处理器缓存保持缓存一致。另外还集成了一种符合 GIC 架构的综合中断及通信系统，该系统配有专用外设，其性能和软件可移植性都更上一层楼。适当配置后，可支持 0 ~ 224 个独立中断资源。这种处理器可支持单个或两个 64bit AMBA 3 AXITM 互联接口。Cortex - A9 MPCore 多核处理器采用了通过硅验证的 ARM MPCore 技术的增强版，实现了可扩展型多核处理。ARM Cortex - A9 多核处理器的系统框图如图 2-3 所示。

ARM Cortex - A9 多核处理器是首款结合了 Cortex 应用级架构以及用于可扩展性能的多处理能力的 ARM 处理器，提供了下列多核技术。

试读版本仅供防疫期间线上教学使用

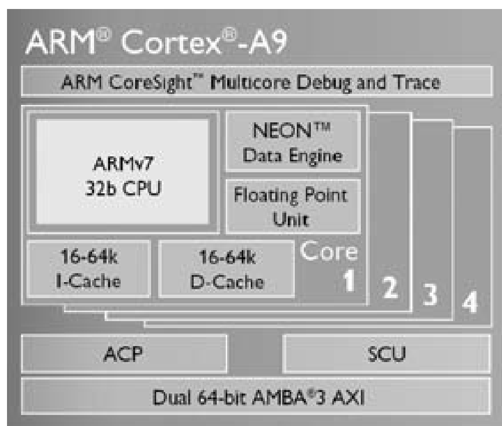

图 2-3　Cortex A9 多核处理器的系统框图

（1）侦测控制单元

侦测控制单元（Snoop Control Unit，SCU），通过 AXI 接口将 1 ~4 个 Cortex – A9 处理器连接到存储系统，相当于 ARM 多核技术的“中央情报局”，负责为支持 MPCore 技术的处理器提供互联、仲裁、通信、缓存间及系统内存传输、缓存一致性及其他多核功能的管理。同时，Cortex – A9 MPCore 处理器还率先向其他系统加速器及无缓冲的 DMA 驱动控制外设开启此类功能，通过处理器缓存层次的共享，有效地提高了性能、减少了整个系统的功耗水平。不仅如此，利用这种系统来维持每个操作系统驱动中的软件一致性，软件复杂性就大大降低了。

（2）一致性加速口

一致性加速口（Accelerator Coherence Port）用于提高系统性能和降低能耗，它和与 AMBA 3 AXI 兼容的 Slave 接口位于 SCU 之上，为多种系统 Master 接口提供了一个互联接口。出于总体系统性能、功耗或软件简化等方面的考虑，最好直接将这些 Master 接口与 Cortex – A9 MPCore 处理器相连。一致性加速口是标准的 AMBA 3 AXI Slave 接口，支持所有标准读写事务，对所接部件无任何附加一致性要求，如图 2-4 所示。

（3）通用中断控制器

通用中断控制器（Generic Interrupt Controller，GIC）用于软件移植和优化的多核通信。它采用了最新标准化和架构，为处理器间通信及系统中断的路由选择及优先级的确定提供了一种丰富而灵活的解决办法。GIC 最多支持 224 个独立中断，通过软件控制，可在整个 CPU 中对每个中断进行分配，确定其硬件优先级并在操作系统与信任区软件管理层之间进行路由。这种路由灵活性加上对中断虚拟进入操作系统的支持，是进一步提升基于半虚拟化管理器解决方案功能的关键因素之一。

（4）先进的总线接口单元

ARM Cortex – A9 MPCore 处理器提供了先进的总线接口单元（Advanced Bus Interface Unit，ABIU），增强了处理器与系统互联之间的接口性能，为各种系统集成芯片设计理念创造了更大的灵活性，并可在高带宽设备中实现低延迟时间。

这种处理器支持单个或两个 64bit AMBA 3 AXI Master 接口的设计配置，可以按 CPU 的

版本仅供防疫期间线上教学使用

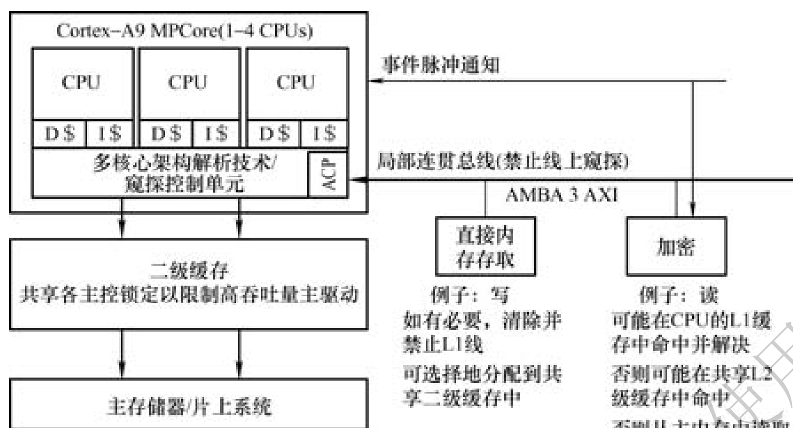

图 2-4　一致性加速口

速度全负荷地将事务传送至系统互联之中，最高速度可达 12GB/s 以上。另外，第二接口也可定义某种事务过滤，只处理全局地址空间的一部分。也就是说，可在处理器内部直接对地址空间进行切分。而且每个接口还支持不同的 CPU –总线频率比（包括同步半时钟比），不但提高了设计灵活性，而且为需要考虑 DVFS 或高速集成内存的设计增加了系统带宽。同时为完整的 ARM 智能能源管理（IEM）提供了良好的支持。

ARM 二级缓存控制器 PrimeCell PL310 与 Cortex – A9 系列处理器同步设计，旨在提供一种能匹配 Cortex – A9 处理器，性能和吞吐能力优化的二级缓存控制器。PL310 最多可为每个接口提供多项 Outstanding AXI 事务支持，支持按 Master 接口进行锁定。这样一来，通过将 PL310 用作加速器与处理器之间的缓冲器，充分利用一致性加速口，实现多个 CPU 或组件之间的可控共享，既提升了系统性能，也降低了相关功耗。

另外，PL310 不但具有 ARM Cortex – A9 先进总线接口单元的各项功能，支持同步 1/2 时钟比，有助于减少高速处理器设计中的延时现象，而且能够对第二 Master AXI 接口设置地址过滤，为分割地址和频率域以及集成片上内存的快速存取提供了支持。PL310 最高可支持 8 MB 的 4 ~ 16 路关联二级缓存，可以奇偶校验及支持 ECC 的 RAM 集成，而且运行速率能够与处理器保持一致。而先进的锁定技术也提供了必要的机制，将缓存用作相关性加速器和处理器之间的传输 RAM。

（5）多核 TrustZone 技术

TrustZone 是 ARM 针对消费电子设备安全所提出的一种架构。对于这种设备的安全威胁，可以有以下几种形态的安全解决方案。

1）外部的硬件安全模块，比如设备上的 SIM 卡。这种方式的优点是 SIM 卡具有特定的软硬件安全特性，能够保护卡内的密钥等资源，而且要攻破其防护所付出的代价很高。缺点是与设备的接口通信速度低，而且不能保护用户界面的安全，即与用户交互的数据的安全，所以在交易支付方面该方案还不能提供好的保护。

2）内部的硬件安全模块，即把类似于智能卡的功能直接放到 SoC 里面。这种方式也只能保护诸如密钥之类的资源，不能保护用户交互数据。在 SoC 里面有两个核：一个普通的 app 核和一个安全核，两个核之间的通信速度也会比较低。

3）软件虚拟化。虚拟化技术如果要保护用户界面的安全，就需要在 GPU 的控制上加入很多的验证，这对于图形处理的性能会产生较大影响。同时，调试端口也仍然是一个问题。

TrustZone 的硬件架构是整个系统设计过程中的安全体系的扩展，目标是防范设备可能遭受到的多种特定威胁（注意这种威胁除了来自恶意软件、黑作坊，还有可能来自设备的持有人）。系统的安全，是通过将 SoC 的硬件和软件资源划分到两个相对独立的部分来获得的，安全子系统对应的是安全空间，其他子系统对应的是普通空间。AMBA3 AXI 总线系统能确保安全空间的资源不会被普通空间所访问。而在 ARM 处理器核也有相应的扩展，来让两个空间的代码能分时运行在同一个核上，这样实际上节省了一个核。另一方面也是扩展了调试体系，使得安全空间的调试有相应的访问控制。

本章小结

本章系统地介绍了 ARM 处理器体系结构的主要内容和 ARM Cortex - A9 处理器核的主要技术。熟练掌握本章内容，是进行 ARM 编程和设计的基础。

思考题

1. ARM 的处理器工作模式有哪些？ARM 如何进行处理器模式切换？
2. ARM 的特权模式有哪些？分别写出每个特权模式对应的寄存器。
3. ARM 的异常类型有哪些？简述 FIQ 异常的进入和退出流程。
4. ARM Cortex - A9 的多核如何协调工作？

试读版本仅供防疫期间线上教学使用

第3章　ARM处理器指令系统

本章主要介绍ARM指令中最常用、最基本的部分，即ARMv4T所包含的ARM指令集，并以此来介绍ARM指令集的特点和使用方法。ARMv4T版本之后指令集所增加的指令，主要是针对各ARM处理核所对应的特色模块，其使用方法和最基本的ARM指令集是一样的。

3.1　ARM指令集概述

ARM处理器是基于精简指令集计算机（RISC）原理设计的，与基于复杂指令集原理设计的处理器相比，指令集和相关译码机制较为简单。ARM指令有32位ARM指令集和16位Thumb指令集。ARM指令集效率高，但是代码密度低；而Thumb指令集具有较高的代码密度，却仍然保持ARM的大多数性能上的优势，它是ARM指令集的子集。

所有的ARM指令都是可以有条件执行的，而Thumb指令仅有一条指令具备条件执行功能。ARM程序和Thumb程序可相互调用，相互之间的状态切换开销几乎为零。

ARM微处理器的指令集是加载/存储型的，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指令集从ARMv4版本开始成熟，并广泛运用于各种等级的ARM处理器核，其主要的ARM指令集版本如下。

（1）ARMv4T

ARMv4T是当前应用最广泛的ARM指令集版本。T表示支持16位的Thumb指令集。ARM7TDMI、ARM720T、ARM9TDMI、ARM940T、ARM920T、Intel的StrongARM等都是基于ARMv4T版本。

（2）ARMv5

ARM9E-S、ARM966E-S、ARM1020E、ARM 1022E以及XScale是基于ARMv5TE版本的；ARM9EJ-S、ARM926EJ-S、ARM7EJ-S、ARM1026EJ-S是基于ARMv5EJ版本的；ARM10采用ARMv5，后缀E表示增强型DSP指令集，包括全部算法和16位乘法操作，J表示支持新的Java。

（3）ARMv6

采用ARMv6指令集的ARM处理器核是ARM11系列，其中：ARM1136J（F）-S基于ARMv6，主要采用的代表性技术有SIMD、Thumb、Jazelle、DBX、（VFP）、MMU；ARM1156T2（F）-S基于ARMv6T2，主要采用的代表性技术有SIMD、Thumb-2、（VFP）、MPU；ARM1176JZ（F）-S基于ARMv6KZ，在ARM1136EJ（F）-S基础上增加MMU、TrustZone；ARM11 MPCore基于ARMv6K，在ARM1136EJ（F）-S基础上可以包括1~4核SMP、MMU。

（4）ARMv7-A

Cortex-A9所隶属的ARMv7-A增加的指令主要包括：NEON单指令多数据（SIMD）

试读版本仅供防疫期间线上教学使用

单元、ARM trustZone 安全扩展以及 Thumb2 指令集。

3.2　ARM 指令的基本格式

ARM 指令集是 32 位的，单条指令具有条件执行功能和丰富的第二操作数选项，能够为编写高效的 ARM 处理器程序提供众多便利的特性。

3.2.1　ARM 指令集编码

不同的 ARM 体系结构版本支持的指令是不同的，但是新的版本一般都兼容以前的版本。ARM 指令集是以 32 位二进制编码的方式给出的，大部分的指令编码中定义了第一操作数、第二操作数、目的操作数、条件标志影响位。每条 32 位 ARM 指令对应不同的二进制编码方式，实现不同的指令功能。ARM 指令集编码如图 3-1 所示。

31 30 29 28	27	26	25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8	7	6	5	4	3 2 1 0
cond	0	0	0	0	0	0	A	S	Rd	Rn	Rs	1	0	0	1	Rm
cond	0	0	0	0	1	U	A	S	RdHi	RdLo	Rs	1	0	0	1	Rm
cond	0	0	0	1	0	B	0	0	Rn	Rd	0 0 0 0	1	0	0	1	Rm
cond	0	0	0	1	0	0	1	0	1 1 1 1	1 1 1 1	1 1 1 1	0	0	0	1	Rm
cond	0	0	0	P	U	0	W	L	Rn	Rd	0 0 0 0	1	S	H	1	Rm
cond	0	0	0	P	U	1	W	L	Rn	Rd	offset	1	S	H	1	offset
cond	0	0	1	opcode				S	Rn	Rd	operand2					
cond	0	1	1	P	U	B	W	L	Rn	Rd	offset					
cond	1	0	0	P	U	S	W	L	Rn	register list						
cond	1	0	1	L	offset											
cond	1	1	0	P	U	N	W	L	Rn	CRd	cp_num	offset				
cond	1	1	1	0	op1				CRn	CRd	cp_num	op2			0	CRm
cond	1	1	1	0	op1			L	CRn	Rd	cp_num	op2			1	CRm
cond	1	1	1	1	swi number											

图 3-1　ARM 指令集编码

3.2.2　ARM 指令基本语法格式

ARM 指令基本语法格式如下：

<opcode> {<cond>} {S}　　<Rd>，<Rn>{，<operand2>}

其中 < > 号内的项是必需的，{} 号内的项是可选的。各项的说明如下：

opcode：指令助记符；

cond：执行条件；

S：是否影响 CPSR 寄存器的值；

Rd：目标寄存器；

Rn：第 1 个操作数的寄存器；

试读版本仅供防疫期间线上教学使用

operand2：第 2 个操作数。

一个最基本的 ARM 指令组成示例见表 3-1。

表 3-1　基本 ARM 指令组成

指令语法	目标寄存器（Rd）	源寄存器 1（Rn）	源寄存器 2（Rm）
ADD r3，r1，r2	r3	r1	r2

通常一个 ARM 指令由指令助记符、目的寄存器、源寄存器 1 和源寄存器 2（第 2 个操作数的方式之一）组成，指令的功能通常是将源寄存器 1 和源寄存器 2 进行运算，然后将运算的结果存放在目的寄存器中。

指令格式举例如下：

```
LDR       R0,[R1]           ;读取 R1 地址上的存储器单元内容,无条件执行 BEQ
ADDS      R1,R1,#1          ;加法指令,R1 = R1 + 1,影响 CPSR 寄存器的标志位
SUBNES    R1,R1,#0x10       ;条件执行减法运算(NE),R1 - 0x10 => R1,影响 CPSR 寄
                             存器的标志位
```

ARM 指令区别于传统的单片机指令的两个主要特点为：①ARM 指令具有灵活的第二个操作数；②ARM 指令可以条件执行，即通过指令的条件码来控制指令的执行。

（1）第 2 个操作数

灵活地使用第 2 个操作数“operand2”能够提高代码效率。它有如下的形式：

1）#immed_ 8r——常数表达式。

第 2 个操作数可以是常数，常数应用举例如下：

```
SUB R1,R1,#1                ; R1 = R1 - 1
```

2）Rm——寄存器方式。

在寄存器方式下，操作数即为寄存器的数值。寄存器方式应用举例如下：

```
SUB      R1,R1,R2           ;R1 = R1 - R2
MOV      PC,R0              ;PC = R0,程序跳转到指定地址
LDR      R0,[R1],-R2        ;R1 所指存储器单元内容存入 R0,且 R1 = R1 - R2
```

3）Rm，shift——寄存器移位方式。

将寄存器的移位结果作为操作数，但 Rm 值保持不变，移位方法见表 3-2。

表 3-2　寄存器的移位说明

操作码	说　　明	操作码	说　　明
ASR #n	算术右移 n 位	ROR #n	循环右移 n 位
LSL #n	逻辑左移 n 位	RRX	带扩展的循环右移 1 位
LSR #n	逻辑右移 n 位	Type Rs	Type 为移位的一种类型，Rs 为偏移量寄存器，低 8 位有效

寄存器移位方式应用举例：

```
MOV R1,R1,LSR #3; R1 = R1/8
ADD R0,R1,R2, LSL #3; R0 = R1 + R2 * 8
```

（2）条件码

ARM 指令中条件码“cond”可以实现高效的逻辑操作，提高代码效率。所有的 ARM 指令都可以条件执行，而 Thumb 指令只有 B（跳转）指令具有条件执行功能。如果指令不标明条件代码，将默认为无条件（AL）执行。16 种指令的条件助记符及对应的含义见表 3-3。

表 3-3 指令的条件助记符及对应的含义

操作码	条件助记符	标志	含义
0000	EQ	Z = 1	相等
0001	NE	Z = 0	不相等
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	有符号数大于或等于
1011	LT	N! = V	有符号数小于
1100	GT	Z = 0，N = V	有符号数大于
1101	LE	Z = 1，N! = V	有符号数小于或等于
1110	AL	任何	无条件执行（指令默认条件）
1111	NV	任何	从不执行（不要使用）

通过条件码的灵活使用可以编写出简洁高效的 ARM 代码。例如，常用的分支代码可利用 ARM 指令的条件执行来编写，示例如下：

C 代码：

```
If(a > b)
    a ++;
Else
    b ++;
```

对应的汇编代码为：

```
CMP R0,R1           ;R0(a)与 R1(b)比较
ADDHI R0,R0,#1      ;若 R0 > R1,则 R0 = R0 + 1
ADDLS R1,R1,#1      ;若 R0≤R1,则 R1 = R1 + 1
```

3.3 ARM 指令的寻址方式

寻址方式是根据指令中给出的地址码字段来实现寻找真实操作数地址的方式。ARM 处理器具有 9 种基本寻址方式：

试读版本仅供防疫期间线上教学使用

立即寻址，寄存器寻址，寄存器偏移寻址，寄存器间接寻址，基址寻址，多寄存器寻址，堆栈寻址，块拷贝寻址，相对寻址。

（1）立即寻址

立即寻址指令中的操作码字段后面的地址码部分即是操作数本身，也就是说，数据就包含在指令当中，取出指令也就取出了可以立即使用的操作数。

立即寻址指令举例如下：

```
SUBS R0,R0,#1            ;R0 减 1,结果放入 R0,并且影响标志位
MOV R0,#0xFF000          ;将立即数 0xFF000 装入 R0 寄存器
```

立即数要求以“#”为前缀，对于以十六进制表示的立即数，要求在“#”后加上“0x”或“&”符号；对于二进制数要在“#”后加上“0b”；对于十进制数要在“#”后加上“0d”或什么也不加。

这里值得注意的是有效立即数问题。

ARM 的 32 位指令编码中，如果立即数是 8 位的，那么可以在 32 位编码中直接表示。但是，立即数也可能是 32 位的。如何在 32 位 ARM 指令编码中存放 32 位立即数？ARM 指令采用一种间接的方法存放 32 位立即数。

在 ARM 数据处理指令中，当参与操作的第 2 操作数为立即数时，这个立即数就采用一个 8 位的常数循环右移偶数位而间接得到。可以用下面公式表示：

<immediate> = immed_8 循环右移(rotate_imm * 2)

其中：

<immediate>	表示有效立即数；
immed_8	表示 8 位常数；
rotate_imm	表示 4 位的循环右移值；
rotate_imm * 2	表示循环右移的位数是一个 4 位二进制数 rotate_imm 的两倍。

例如，0x3F0 用这种编码方式可以表示为：

immed_8 = 0x3F, rotate_imm = 0xE

采用这种间接表示方法，一个 32 位立即数在 32 位指令编码中就可以用 12 位编码来表示，即 4 位 rotate_ imm，8 位 immed_ 8。这种表示方法的问题是，不是所有 32 位立即数都是有效的立即数，只有可以通过上面公式得到的才是有效的立即数，因此在使用立即数时应引起注意。

有效的立即数：

0xFF,0x104,0xFF0,0xFF00,0xFF000,0xFF000000,0xF000000F；

无效的立即数：

0x101,0x102,0xFF1,0xFF04,0xFF003,0xFFFFFFFF,0xF000001F。

（2）寄存器寻址

操作数的值在寄存器中，指令中的地址码字段指出的是寄存器编号，指令执行时直接取

试读版本仅供防疫期间线上教学使用

出寄存器值来操作。

寄存器寻址指令举例如下：

```
MOV    R1,R2          ; R1 = R2
SUB    R0,R1,R2       ;将 R1 的值减去 R2 的值,结果保存到 R0,即 R0 = R1 - R2。
```

(3) 寄存器移位寻址

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

寄存器移位寻址指令举例如下：

```
MOV    R0,R2,LSL #3     ;R2 的值左移 3 位,结果放入 R0,即 R0 = R2 × 8
ANDS   R1,R1,R2,LSL R3  ;R2 的值左移 R3 位,然后和 R1 相加,结果存放在 R1 中
ADD    R3,R2,R1,LSL #3;R1 的值左移 3 位,然后和 R2 相加,结果存放在 R3 中,即 R3 =
R2 + 8 * R1
```

对一个 32 位的寄存器进行移位操作，有逻辑左移、逻辑右移、算术右移、循环右移和扩展为 1 的循环右移，其对应的指令符号和操作如下：

LSL 逻辑左移(Logical Shift Left)
LSR 逻辑右移(Logical Shift Right)
ASR 算术右移(Arithmetic Shift Right)
ROR 循环右移(Rotate Right)
RRX 扩展的循环右移(Rotate Right eXtended by 1 place)

这些移位操作过程如图 3-2 所示。

试读版本仅供防疫期间线上教学使用

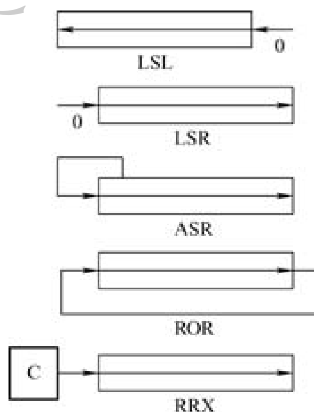

图 3-2　移动操作过程

(4) 寄存器间接寻址

寄存器间接寻址指令中的地址码给出的是一个通用寄存器的编号，所需的操作数保存在寄存器指定地址的存储单元中，即寄存器为操作数的地址指针。寄存器间接寻址指令举例如下：

```
LDR R1,[R2]                 ;将 R2 指向的存储单元的数据读出保存在 R1 中
SWP     R1,R1,[R2]          ;将寄存器 R1 的值和 R2 指定的存储单元的内容交换
```

（5）基址寻址

基址寻址就是将基址寄存器的内容与指令中给出的偏移量相加，形成操作数的有效地址。基址寻址指令举例如下：

```
LDR   R2,[R3,#0x0C]     ;读取 R3+0x0C 地址上的存储单元的内容,放入 R2
STR   R1,[R0,#-4]!      ;先 R0=R0-4,然后把 R1 的值保存到 R0 指定的存储单元
```

基址寻址常用于访问基地址附近的存储单元，包括前索引寻址、带自动索引的前索引寻址、后索引寻址和基址加索引寻址。

1）基址加偏移——前索引寻址

LDR R0，[R1，#4]；将 R1+4 指向的存储单元的数据读出保存在 R0 中，即 R0←[R1+4]。

2）基址加偏移——带自动索引的前索引寻址

LDR R0，[R1，#4]!；将 R1+4 指向的存储单元的数据读出保存在 R0 中，并将地址 R1 寄存器加 4（加"!"，通常表示要根据指令处理结果更新地址寄存器），即 R0←[R1+4]，R1=R1+4。

3）基址加偏移——后索引寻址

LDR R0，[R1]，#4；将 R1 指向的存储单元的数据读出保存在 R0 中，然后将地址 R1 寄存器加 4，即 R0←[R1]，R1=R1+4。

4）基址加索引寻址

LDR R0，[R1，R2]；将 R2+R1 指向的存储单元的数据读出保存在 R0 中，即 R0←[R1+R2]。

（6）多寄存器寻址

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

多寄存器寻址指令举例如下：

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

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

（7）堆栈寻址

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

1）向上生长：向高地址方向生长，称为递增堆栈；

2）向下生长：向低地址方向生长，称为递减堆栈。

堆栈指针指向最后压入的堆栈的有效数据项，称为满堆栈；堆栈指针指向下一个待压入数据的空位置，称为空堆栈。所以可以组合出 4 种类型的堆栈方式：

试读版本仅供防疫期间线上教学使用

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

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

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

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

注意：不论压栈过程还是出栈过程，存储器中的高地址的数据都对应高编号寄存器，并且与大括号中寄存器的排放顺序无关。

（8）块复制寻址

寄存器传送指令用于将一块数据从存储器的某一位置复制到另一位置。如：

```
STMIA    R0!,{R1 - R7}    ;将 R1 ~ R7 的数据保存到存储器中
                          ;存储指针在保存第一个值之后增加
                          ;增长方向为向上增长
STMIB    R0!,{R1 - R7}    ;将 R1 ~ R7 的数据保存到存储器中
                          ;存储指针在保存第一个值之前增加
                          ;增长方向为向上增长
```

根据地址增加的先后顺序，可分为 4 种指令：

1）地址增加在先（IB）：STMIB，LDMIB

2）地址增加在后（IA）：STMIA，LDMIA

3）地址减少在先（DB）：STMDB，LDMDB

4）地址减少在后（DA）：STMDA，LDMDA

（9）相对寻址

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

相对寻址指令举例如下：

```
    BL   SUBR1 ;通过 BL 指令调用到 SUBR1 子程序
               ;程序跳转地址为 PC 提供基准地址加上相对于 SUBRl 的偏移量
         …
SUBR1
         …
    MOV PC,R14;返回
```

3.4　ARM 存储器访问指令

ARM 处理器是典型的 RISC 处理器，对存储器的访问只能使用加载和存储指令实现。ARM 处理器是冯·诺依曼存储结构，程序空间、RAM 空间及 I/O 映射空间统一编址，对外

围 I/O、程序数据的访问均要通过加载/存储指令进行。

存储器访问指令分为单寄存器操作指令和多寄存器操作指令以及寄存器交换指令。

1. 单寄存器操作指令

单寄存器操作指令，见表 3-4。

表 3-4 单寄存器操作指令

助 记 符	说 明	操 作	条件码位置
LDR Rd，addressing	加载字数据	Rd←[addressing]，addressing 索引	LDR {cond}
LDRB Rd，addressing	加载无符号字节数据	Rd←[addressing]，addressing 索引	LDR {cond} B
LDRT Rd，addressing	以用户模式加载字数据	Rd←[addressing]，addressing 索引	LDR {cond} T
LDRBT Rd，addressing	以用户模式加载无符号字节数据	Rd←[addressing]，addressing 索引	LDR {cond} BT
LDRH Rd，addressing	加载无符号半字数据	Rd←[addressing]，addressing 索引	LDR {cond} H
LDRSB Rd，addressing	加载有符号字节数据	Rd←[addressing]，addressing 索引	LDR {cond} SB
LDRSH Rd，addressing	加载有符号半字数据	Rd←[addressing]，addressing 索引	LDR {cond} SH

单寄存器存储指令见表 3-5。

表 3-5 单寄存器存储指令

助 记 符	说 明	操 作	条件码位置
STR Rd，addressing	存储字数据	[addressing]←Rd，addressing 索引	STR {cond}
STRB Rd，addressing	存储字节数据	[addressing]←Rd，addressing 索引	STR {cond} B
STRT Rd，addressing	以用户模式存储字数据	[addressing]←Rd，addressing 索引	STR {cond} T
STRBT Rd，addressing	以用户模式存储字节数据	[addressing]←Rd，addressing 索引	STR {cond} BT
STRH Rd，addressing	存储半字数据	[addressing]←Rd，addressing 索引	STR {cond} H

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

LDR/STR 指令寻址非常灵活，它由两部分组成，其中一部分为一个基址寄存器，可以为任一个通用寄存器；另一部分为一个地址偏移量。地址偏移量有以下 3 种格式：

（1） 立即数

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

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

（2） 寄存器

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

```
LDR      R1,[R0,R2]    ;将 R0 + R2 地址处的数据读出,保存到 R1 中
LDR      R1,[R0,-R2]   ;将 R0 - R2 地址处的数据读出,保存到 R1 中
```

（3）寄存器及移位常数

寄存器移位后的值可以加到基址寄存器，也可以从基址寄存器中减去这个数值。指令举例如下：

```
LDR   R1,[R0,R2,LSL #2]     ;将 R0 + R2 × 4 地址处的数据读出,保存到 R1 中(R0、R2 的值不变)
LDR   R1,[R0,-R2,LSL #2]    ;将 R0 - R2 × 4 地址处的数据读出,保存到 R1 中(R0、R2 的值不变)
```

LDR/STR 指令可加载有符号半字或字节，也可加载/存储无符号半字或字节。有符号位半字/字节加载是指用符号位加载扩展到 32 位，无符号半字/字节加载是指用零扩展到 32 位。

如存储地址 0x40000000 存放一个字数据 -8，补码表示为 0XFFFFFFF8，存储格式为小端模式，R0 寄存器的内容为 0x40000000，则分别执行以下操作后，可以得到 R2 寄存器的内容。

```
LDR      R2,[R0]; 字数据读取,R2 = 0XFFFFFFF8,为 -8 的补码表示
LDRH     R2,[R0]; 无符号半字数据读取,R2 = 0XFFF8
LDRB     R2,[R0]; 无符号字节数据读取,R2 = 0XF8
LDRSH    R2,[R0]; 有符号半字数据读取,R2 = 0XFFFFFFF8
LDRSB    R2,[R0]; 有符号字节数据读取,R2 = 0XFFFFFFF8
```

通过上面的指令例子可知，LDR、LDRSH 和 LDRSB 执行后，R2 从存储器中读到的是有符号数 -8 的 32 位补码表示 0XFFFFFFF8。

2. 多寄存器操作指令

LDM 和 STM 为多寄存器操作指令，可以实现在一组寄存器和一块连续的内存单元之间传输数据。LDM 为加载多个寄存器；STM 为存储多个寄存器。多寄存器操作指令允许一条指令传送 16 个寄存器的任何子集或所有寄存器，指令格式如下：

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

LDM 和 STM 的主要用途是现场保护、数据复制、常数传递等。

多寄存器加载/存储指令的 8 种模式见表 3-6，右边四种为堆栈操作、左边四种为数据传送操作。

表 3-6　多寄存器加载/存储指令的 8 种模式

模式	说　明	模式	说　明
IA	每次传送后地址加 4	FD	满递减堆栈
IB	每次传送前地址加 4	ED	空递减堆栈
DA	每次传送后地址减 4	FA	满递增堆栈
DB	每次传送前地址减 4	EA	空递增堆栈
数据块传送操作		堆栈操作	

进行数据复制时，先设置好源数据指针和目标指针，然后使用块复制寻址指令 LDMIA/STMIA、LDMIB/STMIB、LDMDA/STMDA、LDMDB/STMDB 进行读取和存储。

进行堆栈操作时，要先设置堆栈指针（SP），然后使用堆栈寻址指令 STMFD/LDMFD 、STMED/LDMED、STMFA/LDMFA 和 STMEA/LDMEA 实现堆栈操作。指令格式中，寄存器 Rn 为基址寄存器，装有传送数据的初始地址，Rn 不允许为 R15。后缀“!”表示最后的地址写回到 Rn 中。寄存器列表 reglist 可包含多于一个寄存器或包含寄存器范围，使用“,”分开，如 {R1，R2，R6－R9}，寄存器按由小到大排列。后缀“^”不允许在用户模式或系统模式下使用。若在 LDM 指令且寄存器列表中包含有 PC 时使用，那么除了正常的多寄存器传送外，将 SPSR 也复制到 CPSR 中，这可用于异常处理返回。使用后缀“^”进行数据传送且寄存器列表不包含 PC 时，加载/存储的是用户模式的寄存器，而不是当前模式的寄存器。当 Rn 在寄存器列表中且使用后缀“!”时，对于 STM 指令，若 Rn 为寄存器列表中的最低数字的寄存器，则会将 Rn 的初值保存；其他情况下 Rn 的加载值和存储值不可预知。举例如下：

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

3. 寄存器和存储器交换指令

SWP 指令用于将一个内存单元（该单元地址放在寄存器 Rn 中）的内容读取到一个寄存器 Rd 中，同时将另一个寄存器 Rm 的内容写入到该内存单元中。使用 SWP 可实现信号量操作，指令格式如下：

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

其中，B 为可选后缀，若有 B，则交换字节，否则交换 32 位字；Rd 用于保存从存储器中读入的数据；Rm 的数据用于存储到存储器中，若 Rm 与 Rd 相同，则为寄存器与存储器内容进行交换；Rn 为要进行数据交换的存储器地址，Rn 不能与 Rd 和 Rm 相同。

SWP 指令应用示例：

```
SWP R1,R1,[R0]   ;将 R1 的内容与 R0 指向的存储单元的内容进行交换
SWPB R1,R2,[R0]  ;将 R0 指向的存储单元内容读取 1 字节数据到 R1 中,
                 ;高 24 位清零,并将 R2 的内容写入到该内存单元中,
                 ;最低字节有效
```

3.5 ARM 数据处理指令

数据处理指令主要包含数据传送指令、算术运算指令、逻辑运算指令、比较指令和乘法指令。

数据处理指令只能对寄存器的内容进行操作，而不能对内存中的数据进行操作。所有 ARM 数据处理指令均可选择使用后缀 S，并影响状态标志。比较指令 CMP、CMN、TST 和

试读版本仅供防疫期间线上教学使用

TEQ 不需要后缀 S，它们会直接影响状态标志。

在数据处理指令中，除了比较指令以外，其他的指令如果带有 S 后缀，同时又以 pc 为目标寄存器进行操作，则操作的同时把 SPSR 的内容恢复到 CPSR 中。比如：

```
MOVS PC, #0XFF                ; CPSR = SPSR; PC = 0XFF
ADDS PC, R1, #0XFFFFFF00      ; CPSR = SPSR; PC = R1 + 0XFFFFFF00
ANDS PC, R1, R2               ; CPSR = SPSR; PC = R1 & R2
```

如果在 user 或者 system 模式下使用带有后缀 S 的数据处理指令，同时以 PC 为目标寄存器，那么会产生不可预料的结果。因为 user 和 system 模式下没有 SPSR。

利用数据处理指令 S 后缀，结合 ARM 指令的条件执行功能，可为程序员设计高效 ARM 代码提供便利。如程序设计中常用的循环代码可用 SUBS 和 BNE 进行设计。示例如下：

C 代码：

```
for(i = 100;i! = 0;i - - )
```

对应的汇编代码：

```
    MOV     R0,#100
LOOP
    SUBS    R0,R0,#1    ;R0 减 1,影响标志位
    BNE     LOOP        ;不等于 0,运行下一循环
```

需要注意的是：除比较指令 CMP、CMN、TST 和 TEQ 外的数据处理指令必须加 S 后缀才能影响 CPSR 中状态标志，而后的条件指令才能利用数据处理运算结果进行程序控制，如将之上的循环代码中的“SUBS R0，R0，#1”改成“SUB R0，R0，#1”，BNE 指令执行所依据的条件将不是指令“SUB R0，R0，#1”的运行结果，而是沿用指令执行之前的 CPSR 状态，这样可能陷入死循环（如果之前代码的‘NE’是成立的）。

1. 数据传送指令

数据传送指令见表 3-7。

表 3-7　数据传送指令

助记符	说　明	操　作	条件码位置
MOV　Rd，operand2	数据传送	Rd→operand2	MOV ｛cond｝ ｛S｝
MVN　Rd，operand2	数据非传送	Rd→（~operand2）	MVN ｛cond｝ ｛S｝

（1）数据传送指令 MOV

MOV 指令将立即数或寄存器传送到目标寄存器 Rd，第 2 操作数 operand2 可移位运算等操作。指令格式如下：

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

MOV 指令举例如下：

```
MOVS   R3,R1,LSL #2     ;R3 = R1 << 2,并影响标志位
```

试读版本仅供防疫期间线上教学使用

```
MOV    PC,LR                ;PC = LR,子程序返回
```

（2）数据非传送指令 MVN

MVN 指令将立即数或寄存器（第 2 操作数 operand2）按位取反后传送到目标寄存器 Rd。指令格式如下：

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

MVN 指令举例如下：

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

2. 算术运算指令

算术运算指令见表 3-8。

表 3-8　算术运算指令

助记符	说　明	操　作	条件码位置
ADD　Rd, Rn, operand2	加法运算指令	Rd←Rn + operand2	ADD {cond} {S}
SUB　Rd, Rn, operand2	减法运算指令	Rd←Rn − operand2	SUB {cond} {S}
RSB　Rd, Rn, operand2	逆向减法指令	Rd←operand2 − Rn	RSB {cond} {S}
ADC　Rd, Rn, operand2	带进位加法	Rd←Rn + operand2 + Carry	ADC {cond} {S}
SBC　Rd, Rn, operand2	带进位减法指令	Rd←Rn − operand2 − （NOT） Carry	SBC {cond} {S}
RSC　Rd, Rn, operand2	带进位逆向减法指令	Rd←operand2 − Rn − （NOT） Carry	RSC {cond} {S}

（1）加法运算指令 ADD

ADD 指令将 operand2 的值与 Rn 的值相加，结果保存到寄存器 Rd。指令格式如下：

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

应用示例：

```
ADDS     R1,R1,#1              ;R1 = R1 + 1,并影响标志位
ADDS     R3,R1,R2,LSL #2       ; R3 = R1 + R2 << 2
```

（2）减法运算指令 SUB

SUB 指令用寄存器 Rn 减去 operand2，结果保存到 Rd 中。指令格式如下：

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

应用示例：

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

（3）逆向减法运算指令 RSB

RSB 指令将 operand2 的值减去 Rn，结果保存到 Rd 中。指令格式如下：

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

应用示例：

试读版本仅供防疫期间线上教学使用

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

（4）带进位加法指令 ADC

ADC 将 operand2 的值与 Rn 的值相加，再加上 CPSR 中的 C 条件标志位，结果保存到 Rd 寄存器。指令格式如下：

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

应用示例：

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

（5）带进位减法指令 SBC

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

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

应用示例：

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

（6）带进位逆向减法指令 RSC

RSC 指令用寄存器 operand2 减去 Rn，再减去 CPSR 中的 C 条件标志位，结果保存到 Rd 中。指令格式如下：

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

应用示例：

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

3. 逻辑运算指令

逻辑运算指令见表 3-9。

表 3-9　逻辑运算指令

助记符	说　明	操　作	条件码位置
AND　Rd，Rn，operand2	逻辑与操作指令	Rd←Rn & operand2	AND {cond} {S}
ORR　Rd，Rn，operand2	逻辑或操作指令	Rd←Rn | operand2	ORR {cond} {S}
EOR　Rd，Rn，operand2	逻辑异或操作指令	Rd←Rn ^ operand2	EOR {cond} {S}
BIC　Rd，Rn，operand2	位清除指令	Rd←Rn & (~operand2)	BIC {cond} {S}

（1）逻辑与操作指令 AND

AND 指令将 operand2 的值与寄存器 Rn 的值按位作逻辑“与”操作，结果保存到 Rd 中。指令格式如下：

试读版本仅供防疫期间线上教学使用

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

应用示例：

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

（2）逻辑或操作指令 ORR

ORR 指令将 operand2 的值与寄存器 Rn 的值按位作逻辑“或”操作，结果保存到 Rd 中。指令格式如下：

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

应用示例：

```
ORR     R0,R0,#0x0F        ;将 R0 的低 4 位置 1
```

（3）逻辑异或操作指令 EOR

EOR 指令将 operand2 的值与寄存器 Rn 的值按位作逻辑“异或”操作，结果保存到 Rd 中。指令格式如下：

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

应用示例：

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

（4）位清除指令 BIC

BIC 指令将寄存器 Rn 的值与 operand2 的值的反码按位作逻辑“与”操作，结果保存到 Rd 中。指令格式如下：

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

应用示例：

```
BIC     R1,R1,#0x0F        ;将 R1 的低 4 位清零,其他位不变
```

4. 比较指令

比较指令见表 3-10。

表 3-10　比较指令

助记符	说　明	操　作	条件码位置
CMP　Rn，operand2	比较指令	标志 N、Z、C、V←Rn − operand2	CMP {cond}
CMN　Rn，operand2	负数比较指令	标志 N、Z、C、V←Rn + operand2	CMN {cond}
TST　Rn，operand2	位测试指令	标志 N、Z、C、V←Rn&operand2	TST {cond}
TEQ　Rn，operand2	相等测试指令	标志 N、Z、C、V←Rn^operand2	TEQ {cond}

试读版本仅供防疫期间线上教学使用

（1） 比较指令 CMP

CMP 指令将寄存器 Rn 的值减去 operand2 的值，根据操作的结果更新 CPSR 中的相应条件标志位，以便后面的指令根据相应的条件标志来判断是否执行。指令格式如下：

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

应用示例：

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

（2） 负数比较指令 CMN

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

应用示例：

```
CMN     R0,#1       ;R0 + 1,判断 R0 是否为 1 的补码。若是,则 CPSR 中 Z 标志位置 1
```

（3） 位测试指令 TST

TST 指令将寄存器 Rn 的值与 operand2 的值按位作逻辑“与”操作，根据操作的结果更新 CPSR 中的相应条件标志位，以便后面的指令根据相应的条件标志来判断是否执行。指令格式如下：

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

应用示例：

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

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

（4） 相等测试指令 TEQ

TEQ 指令将寄存器 Rn 的值与 operand2 的值按位作逻辑“异或”操作，根据操作的结果更新 CPSR 中的相应条件标志位，以便后面的指令根据相应的条件标志来判断是否执行。指令格式如下：

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

应用示例：

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

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

5. 乘法指令

ARM7TDMI 具有 3 种乘法指令，分别为 32 × 32 位乘法指令，32 × 32 位乘加指令，32 × 32 位结果为 64 位的乘/乘加指令。乘法指令见表 3-11。

武汉版本仅供防疫期间线上教学使用

表 3-11 乘法指令

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

（1）32 位乘法指令 MUL

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

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

应用示例：

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

（2）32 位乘加指令 MLA

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

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

应用示例：

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

（3）64 位无符号乘法指令 UMULL

UMULL 指令将 Rm 和 Rs 中的值作无符号数相乘，结果的低 32 位保存到 RdLo 中，而高 32 位保存到 RdHi 中。指令格式如下：

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

应用示例：

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

（4）64 位无符号乘加指令 UMLAL

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

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

此版本仅供防疫期间线上教学使用

应用示例：

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

（5）64 位有符号乘法指令 SMULL

SMULL 指令将 Rm 和 Rs 中的值作有符号数相乘，结果的低 32 位保存到 RdLo 中，而高 32 位保存到 RdHi 中。指令格式如下：

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

应用示例：

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

（6）64 位有符号乘加指令 SMLAL

SMLAL 指令将 Rm 和 Rs 中的值做有符号数相乘，64 位乘积与 RdHi、RdLo 相加，结果的低 32 位保存到 RdLo 中，而高 32 位保存到 RdHi 中。指令格式如下：

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

应用示例：

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

3.6 ARM 分支指令

在 ARM 中有两种方式可以实现程序的跳转：一种是使用分支转移指令直接跳转；另一种则是直接向 PC 寄存器赋值来实现跳转。ARM 的分支转移指令，可以从当前指令向前或向后的 32MB 的地址空间跳转，根据完成的功能分支指令可以分为以下 4 种：

1）B 指令：分支指令。指令格式如下：

```
B{cond}    label
```

B 指令跳转到指定的地址执行程序。指令举例如下：

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

分支指令 B 限制在当前指令的 ±32 MB 的范围内。

2）BL 指令：带链接的分支指令。指令格式如下：

```
BL{cond}    label
```

BL 指令先将下一条指令的地址复制到 LR 链接寄存器中，然后跳转到指定地址运行程序。指令举例如下：

```
    BL    SUB1          ;下一条指令地址存入 LR
                        ;跳转至子程序 SUB1 处
      …
```

试读版本仅供防疫期间线上教学使用

```
SUB1    …
      MOV     PC,   LR       ;子程序返回
```

注意：跳转地址限制在当前指令的 ±32 MB 的范围内。BL 指令通常用于子程序调用。

3）BX 指令：带状态切换的分支指令。指令格式如下：

```
BX{cond}      Rm
```

BX 指令跳转到 Rm 指定的地址执行程序。若 Rm 的位［0］为 1，则跳转时自动将 CPSR 中的标志 T 置位，即把目标地址的代码解释为 Thumb 代码；若 Rm 的位［0］为 0，则跳转时自动将 CPSR 中的标志 T 复位，即把目标地址的代码解释为 ARM 代码。

以下代码使用 CODE32 和 CODE16 来指明其后的代码分别为 ARM 和 Thumb 代码，然后使用 BX 实现 ARM 和 Thumb 代码之间的程序跳转。

```
      CODE32                          ;ARM 状态下的代码
      LDR R0, =Into_Thumb +1          ;产生跳转地址并且设置最低位
      BX R0                           ;分支跳转,进入 Thumb 状态
        …
      CODE16                          ;Thumb 状态下的代码
Into_Thumb
        …
      LDR R3, =Back_to_ARM            ;产生字对齐的跳转地址,最低位被清除
      BX R3                           ;分支跳转,返回到 ARM 状态
      CODE32                          ;ARM 状态下的代码
Back_to_ARM
      …
```

4）BLX 指令：带链接和状态切换的分支指令。指令格式如下：

```
BLX      <target address>
```

BLX 指令先将下一条指令的地址复制到 R14（即 LR）链接寄存器中，然后跳转到指定地址处执行程序（只有 ARM 指令集 v5T 及以上指令集版本支持 BLX），转移地址限制在当前指令的 ±32MB 的范围内。

3.7 协处理器指令

ARM 体系结构允许通过增加协处理器来扩展指令集。最常用的协处理器是用于控制片上功能的系统协处理器，例如，控制 Cache 和存储管理单元的 cp15 协处理器。此外，还有用于浮点运算的浮点 ARM 协处理器，各生产商还可以根据需要开发自己的专用协处理器。

ARM 协处理器指令可分为以下 3 类，表 3-12 列出了所有协处理器处理指令。

试读版本仅供防疫期间线上教学使用

表 3-12 协处理器处理指令

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

1. 协处理器数据操作指令

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

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

应用示例：

```
CDP    p7,0,c0,c2,c3,0    ;对协处理器 7 操作,操作码为 0
                          ;可选操作码为 0
CDP    p6,1,c3,c4,c5      ;对协处理器 6 操作,操作码为 1
```

2. 协处理器数据存取指令

协处理器数据存取指令 LDC/STC 可以将某一连续内存单元的数据读取到协处理器的寄存器中，或者将协处理器的寄存器数据写入到某一连续的内存单元中，传送的字数由协处理器来控制。若协处理器不能成功地执行该操作，将产生未定义指令异常中断。

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

应用示例：

```
LDC p5,c2,[R2,#4];将 [R2+4]存储器单元字数据加载到协处理器 p5 的 c2 中
LDC p6,c2,[R1]; 将 [R1]存储器单元字数据加载到协处理器 p6 的 c2 中
STC p5,c1,[R0];将协处理器 p6 的 c1 的内容存储到[R0]存储器单元
STC p5,c1,[R0,#-0x04];将协处理器 p5 的 c1 的内容存储到[R0-4]存储器单元
```

3. 协处理器寄存器传送指令

如果需要在 ARM 处理器中的寄存器与协处理器中的寄存器之间进行数据传送，那么可以使用 MCR/MRC 指令。MCR 指令用于将 ARM 处理器的寄存器中的数据传送到协处理器的寄存器。MRC 指令用于将协处理器的寄存器中的数据传送到 ARM 处理器的寄存器中。若协处理器不能成功地执行该操作，将产生未定义指令异常中断。

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

应用示例：

```
MCR p6,2,R7,c1,c2         ;ARM 处理器的寄存器 R7 送到协处理器的寄存器 c2 中。
                          ;操作码为 2
MRC p15,5,r4,c0,c2,3      ; 协处理器源寄存器为 p15 的 c0 和 c2,目的寄存器为 ARM 寄
                            存器 r4,操作码为 5,可选操作码为 3。
```

3.8 杂项指令

ARM 指令集中有 3 条指令作为杂项指令，实际上这 3 条指令非常重要，见表 3-13。

表 3-13 杂项指令

助记符	说　明	操　作	条件码位置
SWI immed_ 24	软中断指令	产生软中断，处理器进入管理模式	SWI {cond}
MRS Rd，psr	读状态寄存器指令	Rd←psr，psr 为 CPSR 或 SPSR	MRS {cond}
MSR psr_ fields，Rd/#immed_ 8r	写状态寄存器指令	psr_ fields←Rd/#immed_ 8r，psr 为 CPSR 或 SPSR	MSR {cond}

1. 读状态寄存器指令

在 ARM 处理器中，只有 MRS 指令可以对状态寄存器 CPSR 和 SPSR 进行读操作。通过读 CPSR 可以了解当前处理器的工作状态。读 SPSR 寄存器可以了解到进入异常前的处理器状态。指令格式如下：

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

其中，Rd 为目标寄存器，Rd 不允许为 R15；psr 为 CPSR 或 SPSR。

指令举例如下：

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

当进程切换或允许异常中断嵌套时，也需要使用 MRS 指令来读取 SPSR 状态值，并保存起来。

2. 写状态寄存器指令

在 ARM 处理器中，只有 MSR 指令可以对状态寄存器 CPSR 和 SPSR 进行写操作。指令格式如下。

```
MSR{cond}    psr_fields,#immed_8r
MSR{cond}    psr_fields,Rm
```

其中，psr 是指 CPSR 或 SPSR。fields 设置状态寄存器中需要操作的位。状态寄存器的

试读版本仅供防疫期间线上教学使用

32 位可以分为 4 个 8 位的域，bits[31：24] 为条件标志位域，用 f 表示；bits [23：16] 为状态位域，用 s 表示；bits [15：8] 为扩展位域，用 x 表示；bits [7：0] 为控制位域，用 c 表示。immed_ 8r 为要传送到状态寄存器指定域的立即数（传输低 8 位）。Rm 为要传送到状态寄存器指定域的数据源寄存器。

指令举例如下：

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

只有在特权模式下才能修改状态寄存器。程序中不能通过由 MSR 指令直接修改 CPSR 中的 T 控制位来实现 ARM 状态/Thumb 状态的切换，必须使用 BX 指令完成处理器状态的切换。MRS 与 MSR 配合使用，实现 CPSR 或 SPSR 寄存器的读–修改–写操作，可用来进行处理器模式切换、允许/禁止 IRQ/FIQ 中断等设置。

程序状态寄存器读写指令的应用举例如下：

（1）使能 IRQ 中断

```
MRS R0,CPSR;读当前程序状态寄存器内容到 R0
BIC R0,R0,#0x80;修改 IRQ 中断允许位,但不改变其他位数据,该位清零,表示允许 IRQ
                中断
MSR CPSR_c,R0;将 R0 的低 8 位设置给 CPSR_c
```

（2）禁止 IRQ 中断

```
MRS R0,CPSR; 读当前程序状态寄存器内容到 R0
ORR R0,R0,#0x80; 修改 IRQ 中断允许位,但不改变其他位数据,该位置 1,表示禁止 IRQ
                 中断
MSR CPSR_c,R0; 将 R0 的低 8 位设置给 CPSR_c
```

（3）设置中断模式堆栈地址为 0x40000000

```
MSR CPSR_c,#0xD2;修改处理器工作模式为中断模式,该条指令必须在特权模式下执行
MOV SP, #0x40000000;设置堆栈地址为 0x40000000
```

试读版本仅供防疫期间线上教学使用

3. 软中断指令 SWI

软中断指令 SWI 用于产生软中断，从而实现从用户模式变换到管理模式，并且将 CPSR 保存到管理模式的 SPSR 中，然后程序跳转到 SWI 异常入口。在其他模式下也可使用 SWI 指令，处理器同样地切换到管理模式。指令格式如下：

```
SWI{cond} immed_24
```

指令举例如下：

```
SWI    0          ;软中断,中断立即数为 0
SWI    0xl23456   ;软中断,中断立即数为 0xl23456
```

使用 SWI 指令时,通常使用以下两种方法进行传递参数。

1）指令中的 24 位立即数指定了用户请求的服务类型，参数通过通用寄存器传递。指

令举例如下：

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

2）指令中的 24 位立即数被忽略，用户请求的服务类型由寄存器 R0 的值决定，参数通过其他的通用寄存器传递。指令举例如下：

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

SWI 异常中断处理程序要通过读取引起软中断的 SWI 指令，以取得 24 位立即数。

3.9 其他指令介绍

1. 特殊指令

fmxr /fmrx 指令是 NEON 下的扩展指令，在做浮点运算的时候，要先打开 vfp，因此需要用到 fmxr 指令。

fmxr：由 ARM 寄存器将数据转移到协处理器中；
fmrx：由协处理器将数据转移到 ARM 寄存器中。

NEON 下浮点异常寄存器 FPEXC 是一个可控制 SIMD 及 VFP 的全局使能寄存器，并指定了这些扩展技术是如何记录的。FPEXC 的位定义见表 3-14。

表 3-14 FPEXC 的位定义

位	域	功能描述
[31]	EX	异常位，该位指定了有多少信息需要存储记录 SIMD/VFP 协处理器的状态
[30]	EN	NEON/VFP 使能位，设置 EN 位为 1，则开启 NEON/VFP 协处理器，复位会将 EN 置 0
[29：0]		保留

如果要打开 VFP 协处理器的话，可以用以下指令：

```
mov r0, #0x40000000
fmxr fpexc, r0 ;使能 NEON and VFP 协处理器。
```

2. CLZ 指令

CLZ 指令用于计算最高符号位与第一个 1 之间的 0 的个数。当一些操作数需要规范化（使其最高位为 1）时，该指令用于计算操作数需要左移的位数。指令格式如下：

```
CLZ {cond} Rd,Rm
```

其中，cond 是一个可选的条件代码；Rd 是目标寄存器；Rm 是操作数寄存器。CLZ 指令对 Rm 中的值的前导零进行计数，并将结果返回到 Rd 中，如果未在操作数寄存器中设置任何位，则该结果值为 32；如果 Rm 操作数的最高位为 1，则结果值为 0。该指令不会影响

试读版本仅供防疫期间线上教学使用

CPSR 中的标志位。ARM 指令集必须是 ARMv5 版本以上。指令举例如下：

CLZ R1,R0;如果 R0 = 0X00ff00ff,则运行之后,R1 = 0x8。

3. 饱和指令

饱和指令是用来设计饱和算法的一组指令，如图 3-3 所示，饱和指令在运算结果出现饱和时，其运算结果为饱和的边界值，而非饱和指令的运算结果将出现跳变。

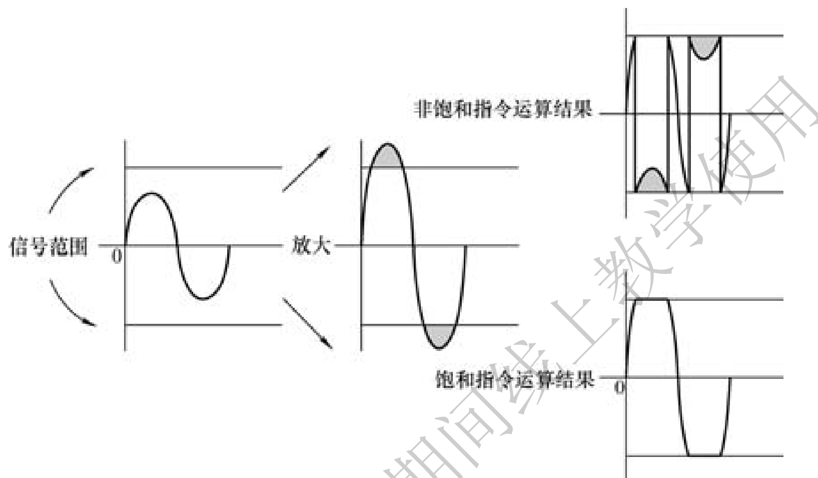

图 3-3　饱和指令使用示意图

指令运行结果出现下列 3 种情况之一，就称为饱和：

1）对于有符号饱和运算，如果指令运算结果小于 -2^n，则返回结果将为 -2^n。

2）对于无符号饱和运算，如果指令运算结果是负值，那么返回的结果将为 0。

3）对于结果大于 2^n-1 的情况，则返回结果将为 2^n-1。

这时，饱和指令执行饱和运算，并设置程序状态寄存器的 Q 标记为“1”。ARM 饱和指令主要有以下 4 条：

QADD：带符号饱和加法指令。

QSUB：带符号饱和减法指令。

QDADD：带符号饱和加倍加法指令。

QDSUB：带符号饱和加倍减法指令。

这 4 条饱和指令，将指令的运算结果饱和导入有效范围（$-2^{31} \leq x \leq 2^{31}-1$）内。

语法格式：

```
op{cond} {Rd} ,Rm,Rn
```

其中，op 是 QADD、QSUB、QDADD 和 QDSUB 之一，cond 是一个可选的条件代码，Rd 是目标寄存器，Rm，Rn 是存放操作数的寄存器（注：不要将 R15 用作 Rd、Rm 或 Rn）。

用法为：QADD 指令可将 Rm 和 Rn 中的值相加；QSUB 指令可从 Rm 中的值减去 Rn 中的值；QDADD/QDSUB 指令涉及并行指令，因此这里不多做讨论。

如果饱和指令执行结果发生饱和，则这些指令设置程序状态寄存器中 Q 标记，若要读取 Q 标记的状态，需要使用 MRS 指令。该指令可用于 v5T－E 及 v6 或者更高版本的 ARM

试读版本仅供防疫期间线上教学使用

指令集体系中。

指令举例如下：

QADD R0，R1，R9 ；将 R1 和 R9 进行饱和相加，结果存入 R0 中。

本 章 小 结

本章先介绍了 ARM 指令集中最基本的部分，这部分指令被 ARMv4T 之后所有的 ARM 指令集版本兼容，也是 ARM 汇编程序设计最主要使用的指令；然后介绍了 ARMv4T 之后 ARM 指令集版本新增加的功能，主要和 ARM 协处理器有关，读者在使用时可参阅具体的技术手册。

思 考 题

1. 存储器存储格式为小端模式，地址 0x3000000 存储的数为 –1，R0 = 0x3000000，分别执行以下指令，寄存器 R2 的内容是什么？

1）LDR R2，[R0]；

2）LDRH R2，[R0]；

3）LDRB R2，[R0]；

4）LDRSH R2，[R0]；

5）LDRSB R2，[R0]。

2. 编写汇编程序实现禁止和使能 FIQ 中断，禁止和使能 IRQ 中断。

3. 当前处理器处于管理模式，编写程序将 FIQ 模式的堆栈指针设置为 0x30000000。

4. R1，R2 中分别存一个 32 位的数，分别编写汇编程序实现以下功能：

1）如果 R1 的第 8 位为 1，则将 R2 的第 12 位取反。

2）如果 R1 的第 12 位为 1，则将 R2 的高 8 位，即 25 到 32 位设置为 1。

3）如果 R1 的第 18 位为 1，则将 R2 的次高位，即 17 位到 24 位设置为 0，其他位保持不变。

5. 分别编写汇编程序实现以下复制功能的 C 语言代码，其中函数体采用 1)、2）和 3）对应的编程方法。

```
void datacopy(int * src,int * dest){
1) for(i=0;i<10;i++)
        dest[i]=src[i];
2) for(i=0;i<10;i++)
       *(dest++)=*(src++);
3) *dest=*src;
    for(i=0;i<9;i++)
        *(++dest)=*(++src);
}
```

试读版本仅供防疫期间线上教学使用