中国科学技术大学
University of Science and Technology of China
信息科学技术学院

计算机原理与嵌入式系统

第6章 ARM处理器的指令系统

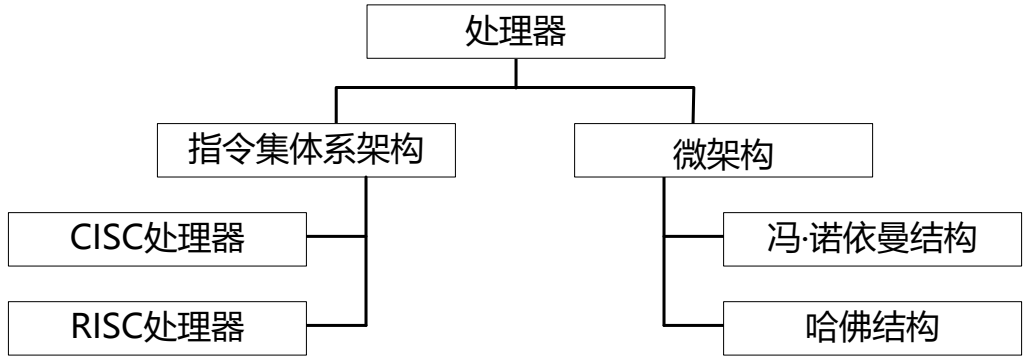
目录

指令系统概念

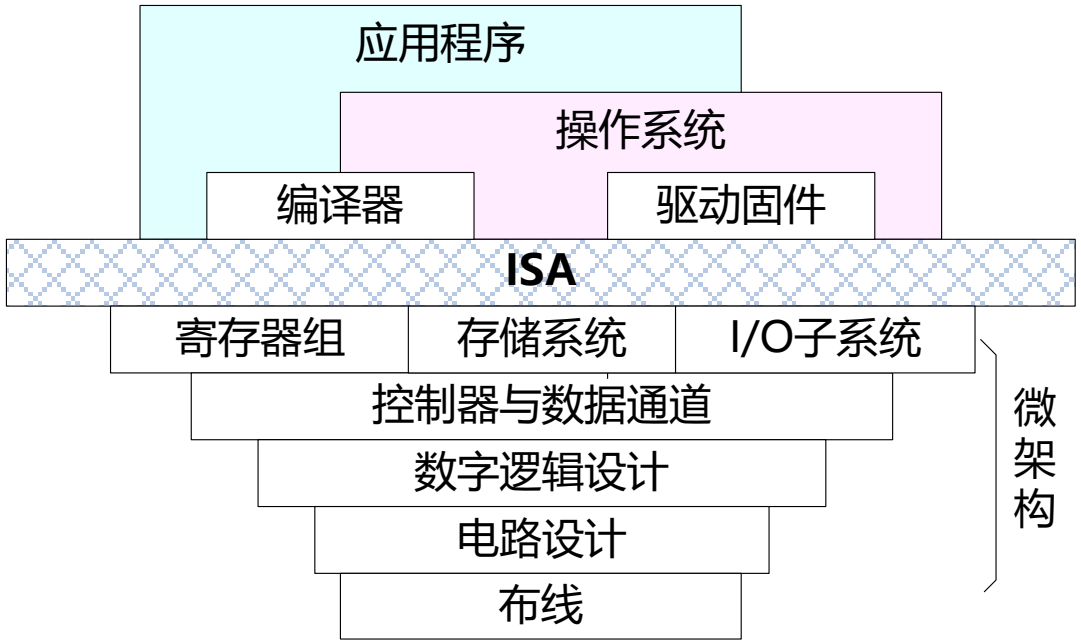
- 指令系统：计算机系统中所有机器指令的集合
- 机器指令是用一个比特串来表示的
  - 表示一条指令的二进制代码（即比特串，或称位串）称为指令字，简称指令
  - 指令字可以是固定长度的，也可以是可变长度的
  - 指令字需要包含操作码、操作数（可能有多个）和操作数地址等字段
- 所有指令字的相关信息在指令集中予以规定。

第5章相关概念：描述处理器指令及其功能、组织方式的规范称为指令集体系架构（ISA）

ARM处理器在ISA方面属于RISC处理器，但有多个体系结构版本。在微架构方面，ARM处理器有基于冯诺依曼结构和哈佛结构两种不同实现方式。



目录



第5章：ARM公司在多个体系结构版本的基础上还定义了若干增强型版本，同时在微架构方面，增加或者选配了相应的功能部件，例如Cache或TCM、DSP、VFP、Java加速器、支持SIMD的多媒体处理单元以及各种跟踪调试部件等，形成了多种具有不同增强功能的处理器产品。

ARM处理器的指令集
Arm Instruction Set Architecture

- 不同时期的ARM处理器指令集存在较大差异
- 目前，ARM公司将其不同系列处理器所支持的指令集架构（ISA，Instruction Set Architecture）统一为三个：A64、A32和T32
  - A64的指令长度固定为32比特
    - ARMv8-A中引入，实现对64比特处理器的支持
  - A32的指令长度固定为32比特
    - 过去称为ARM指令集
  - T32的指令长度既有16比特，也有32比特
    - 是一种混合长度的指令集
    - 过去称为Thumb2指令集

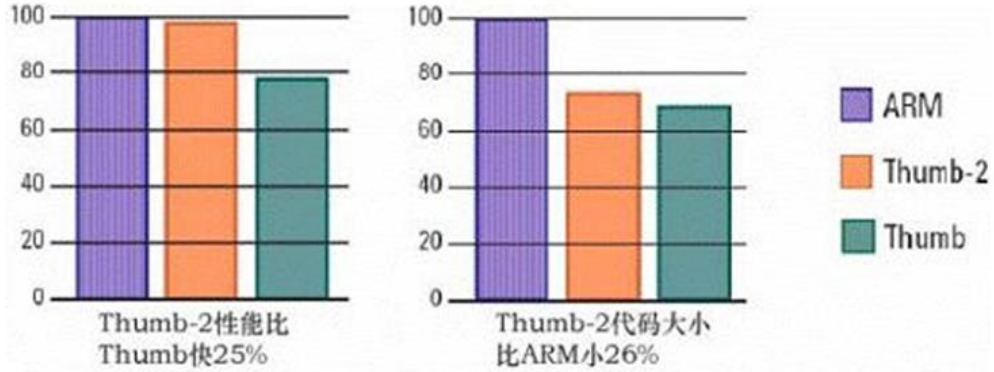
https://developer.arm.com/architectures/instruction-sets

ARM的不同指令集

- ARM指令集，32位 → A32
- ARMv4T起，引入了16 位 Thumb® 指令集
- ARMv6T2起，引入了32 位 Thumb® 指令集
- 2003年引入Thumb®-2技术，16位/32位混合

T32

2019年12月发布的ARMv8中已统一描述为T32



- ARMv7起，引入了ThumbEE(称为Jazelle RCT技术)
- ARMv8起，引入了32位的A64指令集，用于64位微处理器架构

ARM各版本支持的指令集体系结构

- T32是ARM不同版本处理器均支持的指令集
- 不同处理器设计增加了不同的指令集扩展（也称扩展指令）

微架构	支持的指令集架构			重要特性
	A64	A32	T32	
ARMv8.1-M			是	支持矢量扩展（MVE）
ARMv8-A	是	是	是	支持虚拟内存管理（VMSA）
ARMv8-R		是	是	支持内存保护管理（PMSA）
ARMv8-M			是	可选支持内存保护管理（PMSA）
ARMv7-A		是	是	支持虚拟内存管理（VMSA）
ARMv7-R		是	是	支持内存保护管理（PMSA）
ARMv7-M			是	支持16比特和32比特的Thumb2指令
ARMv6-M			是	支持ARMv7-M指令集的子集
ARMv5				支持16比特Thumb指令

指令集扩展
Arm Architecture Extensions

- CPU都有一个基本的指令集，为了提高CPU在某些方面的性能，就必须增加一些特殊的指令满足需求，这些新增的指令就构成了扩展指令集。
- DSP extensions → 数字信号处理
- Floating-point Extension →浮点数运算
- Neon: 单指令多数据（SIMD）
- Helium: Armv8.1-M引入的M-Profile Vector Extension，矢量处理，用于Cortex-M系列
- ……

https://developer.arm.com/architectures/instruction-sets

题外话：Intel的指令集扩展

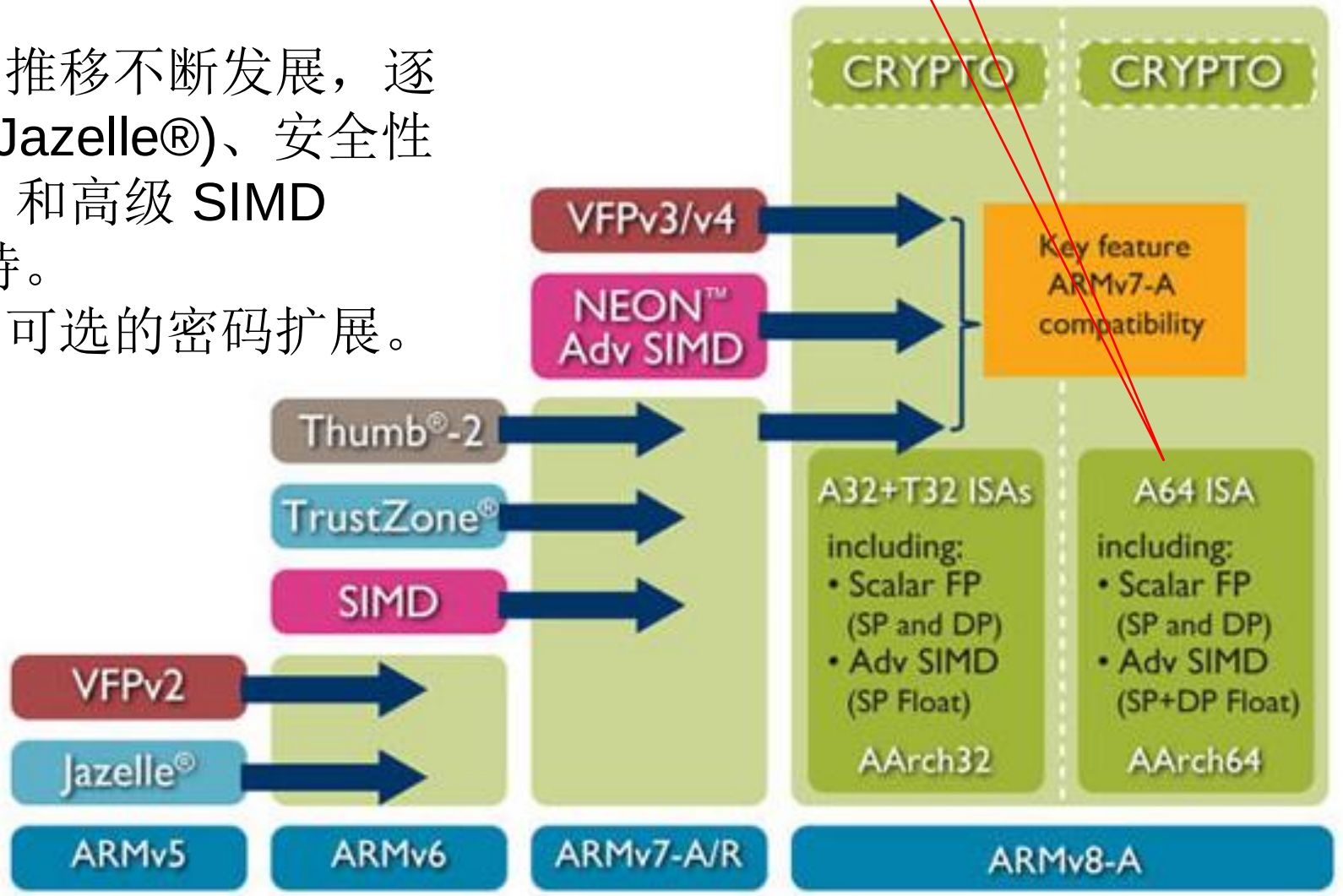
- 常见的Intel处理器指令集扩展
  - Single Instruction Multiple Data (SIMD)
  - Intel® **S**treaming **S**IMD **E**xtensions (Intel® SSE, Intel® SSE2, Intel® SSE3, and Intel® SSE4)
  - Intel® **A**dvanced **V**ector **E**xtensions (Intel® AVX,Intel® AVX2, and Intel® AVX-512)
  - Intel® **SHA** Extension Definitions (**S**ecure **H**ash **A**lgorithm)
  - ….

Instruction	Op 1	Op 2	Op 3	Opcode
SHA1 New Instructions				
SHA1RNDS4	xmm (rw)	xmm/m128 (r)	imm8	0F 3A CC /r ib
SHA1NEXTE	xmm (rw)	xmm/m128 (r)	NA	0F 38 C8 /r
SHA1MSG1	xmm (rw)	xmm/m128 (r)	NA	0F 38 C9 /r
SHA1MSG2	xmm (rw)	xmm/m128 (r)	NA	0F 38 CA /r
SHA256 New Instructions				
SHA256RNDS2	xmm (rw)	xmm/m128 (r)	<xmm0> (implicit)	0F 38 CB /r
SHA256MSG1	xmm (rw)	xmm/m128 (r)	NA	0F 38 CC /r
SHA256MSG2	xmm (rw)	xmm/m128 (r)	NA	0F 38 CD /r

Intel® Instruction Set Extensions Technology
https://www.intel.com/content/www/us/en/support/articles/000005779/processors.html
Intel® SHA Extension Definitions
https://software.intel.com/en-us/articles/intel-sha-extensions

ARM的指令集及扩展指令集

- ARM 架构随着时间的推移不断发展，逐步提供了 Java 加速 (Jazelle®)、安全性 (TrustZone®)、SIMD 和高级 SIMD (NEON™) 技术的支持。
- ARMv8-A 架构增加了可选的密码扩展。



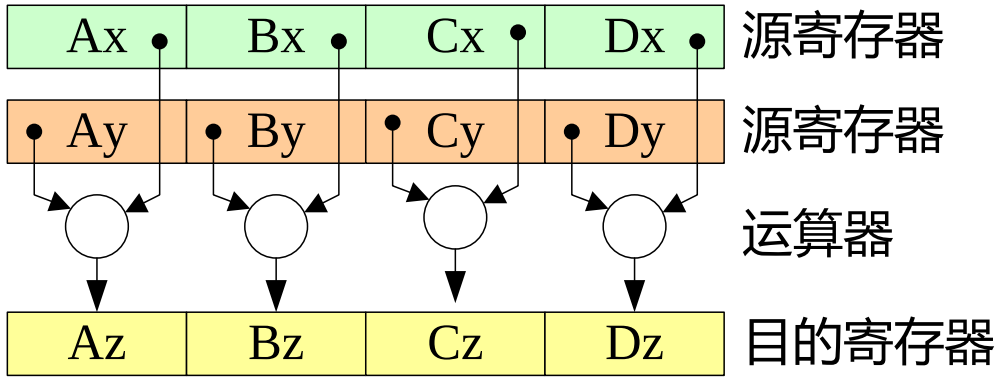
http://www.arm.com/zh/products/processors/instruction-set-architectures/index.php

ARM的指令集扩展

- **DSP 扩展**
  - 单周期 16x16 和 32x16 MAC 实现
  - 零开销饱和扩展支持
  - 用于加载和存储寄存器对的新指令，包含增强的寻址模式
  - CLZ 指令，改善归一化以及除法运算性能
- **浮点数运算扩展**
  - ARM 浮点体系结构 (VFP) 为半精度、单精度和双精度浮点运算中的浮点操作提供硬件支持。
  - 运算：Add、Sub、Mult、Neg-Mult、Negate、Abs Value、Compare、Div、Square Root
  - FMAC：Multiply-Add、Multiply-Subtract、Neg-Multiply-Add、Neg-Multiply-Subtract、类型转换、加载/存储标量和矢量，64 位/周期

ARM的指令集扩展

- 针对多媒体的 **SIMD** 扩展
  - 同时计算 2x16 位或 4x8 位操作数
  - 小数运算
  - 用户可定义的饱和模式（任意字宽）
  - 双 16x16 乘加/减 32x32 小数 MAC
  - 同时 8/16 位选择操作



- **NEON 技术是 ARM Cortex™-A 系列处理器的 128 位 SIMD体系结构扩展。它具有 32 个寄存器，64 位宽（双倍视图为 16 个寄存器，128 位宽）。NEON 指令可执行“打包的 SIMD”处理：**
  - 寄存器被视为同一数据类型的元素的矢量
  - 数据类型：8 位、16 位、32 位、64 位单精度浮点
  - 指令在所有通道中执行同一操作

ARM指令集扩展

- 安全计算的指令扩展
  - ARM Jazelle 包括在任何现有 JVM 和 Java 平台中支持 Jazelle 硬件的技术。
  - ARM TrustZone® 技术是系统范围的安全方法，针对高性能计算平台上的大量应用，包括安全支付、数字版权管理 (DRM) 和基于 Web 的服务。
- 虚拟机有关的指令扩展
  - ARM架构的虚拟化扩展（ Virtualization Extensions ）提供了在ARM处理器基础上建立虚拟机（ virtual machines ）的基本支持

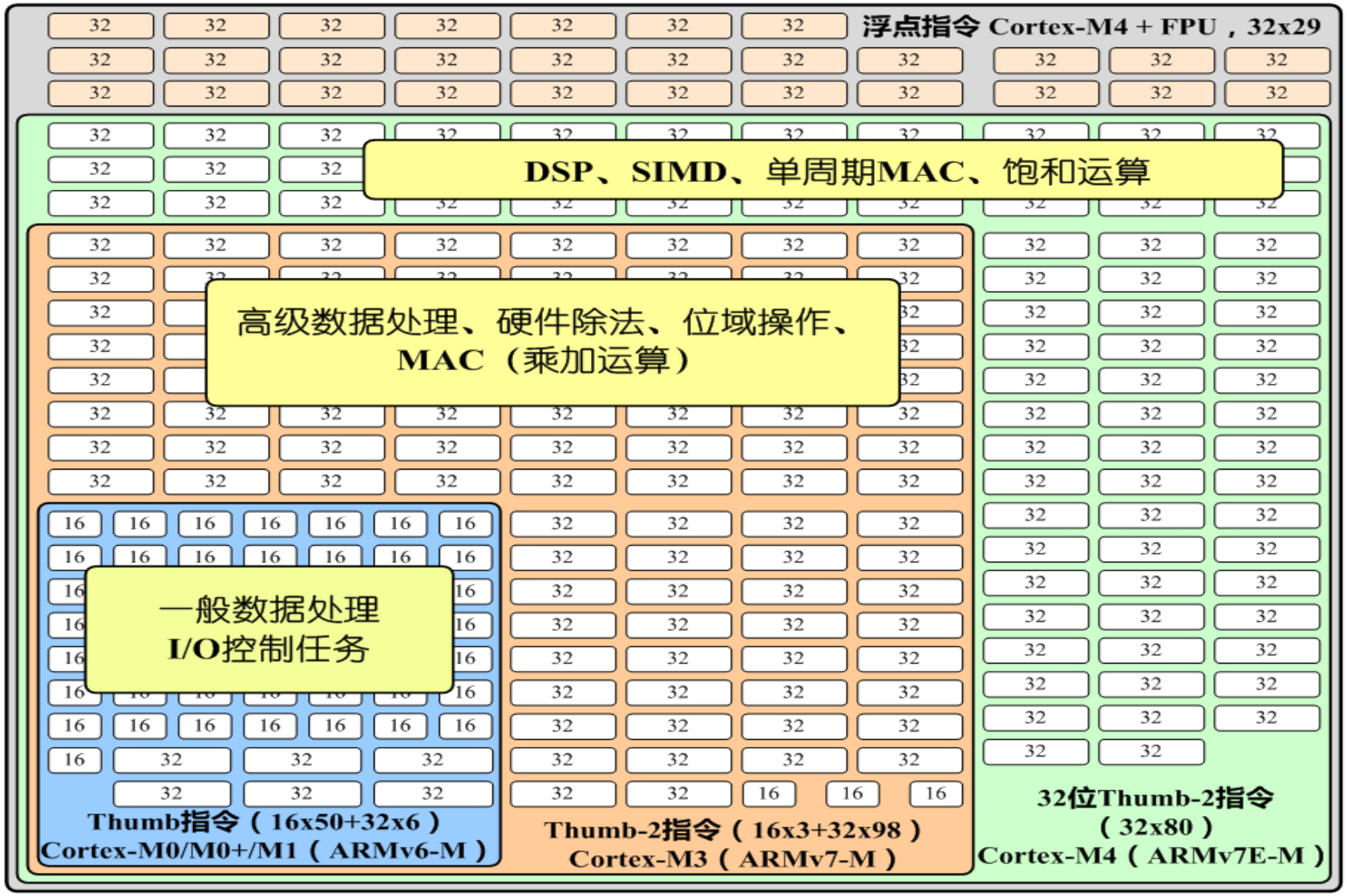
Cortex-M系列处理器支持的指令集和扩展指令

- Y表示支持，N表示不支持，P表示部分支持，O表示可选支持

指令集	指令长度(bits)	指令	m0	m0+	m1	m3	m4	m7	m23	m33
Thumb	16	51条基本指令	Y	Y	Y	Y	Y	Y	Y	Y
		CBNZ, CBZ	N	N	N	Y	Y	Y	Y	Y
		IT	N	N	N	Y	Y	Y	N	Y
Thumb2	32	BL, DMB, DSB, ISB, MRS, MSR	Y	Y	Y	Y	Y	Y	Y	Y
		94条基本指令	N	N	N	P	Y	Y	N	Y
		SDIV, UDIV	N	N	N	Y	Y	Y	Y	Y
数字信号处理	32	80条基本指令	N	N	N	N	Y	Y	N	O
单精度浮点	32	25条基本指令	N	N	N	N	O	O	N	O
双精度浮点	32	14条基本指令	N	N	N	N	N	O	N	N
TrustZone	16	BLXNS, BXNS	N	N	N	N	N	N	O	O
	32	SG, TT, TTT, TTA, TTAT	N	N	N	N	N	N	O	O

Cortex-M3/M4指令功能区别

此图在5.2 Cortex-M3/M4处理器结构部分出现过



讨论：Cortex-M处理器的兼容性

- 由上图可见，在Cortex-M0/M0+/M1上编译过的代码可在Cortex-M3/M4上直接运行
- Cortex-M3/M4不支持ARM指令，不能向后兼容传统ARM处理器（如ARM7TDMI），亦即Cortex-M处理器不能直接运行ARM7TDMI的二进制代码
- 但是，Cortex-M3/M4的Thumb-2指令是Thumb指令的超集， 大部分ARM7TDMI的指令可以移植为等价的32位Thumb-2指令，因此传统ARM处理器上的应用软件可以重新编译后再移植到Cortex-M3/M4上
- 虽然指令有16位和32位之分，对于同一个操作，不同长度的指令执行时间相同

讨论：Cortex-M处理器的适用场景

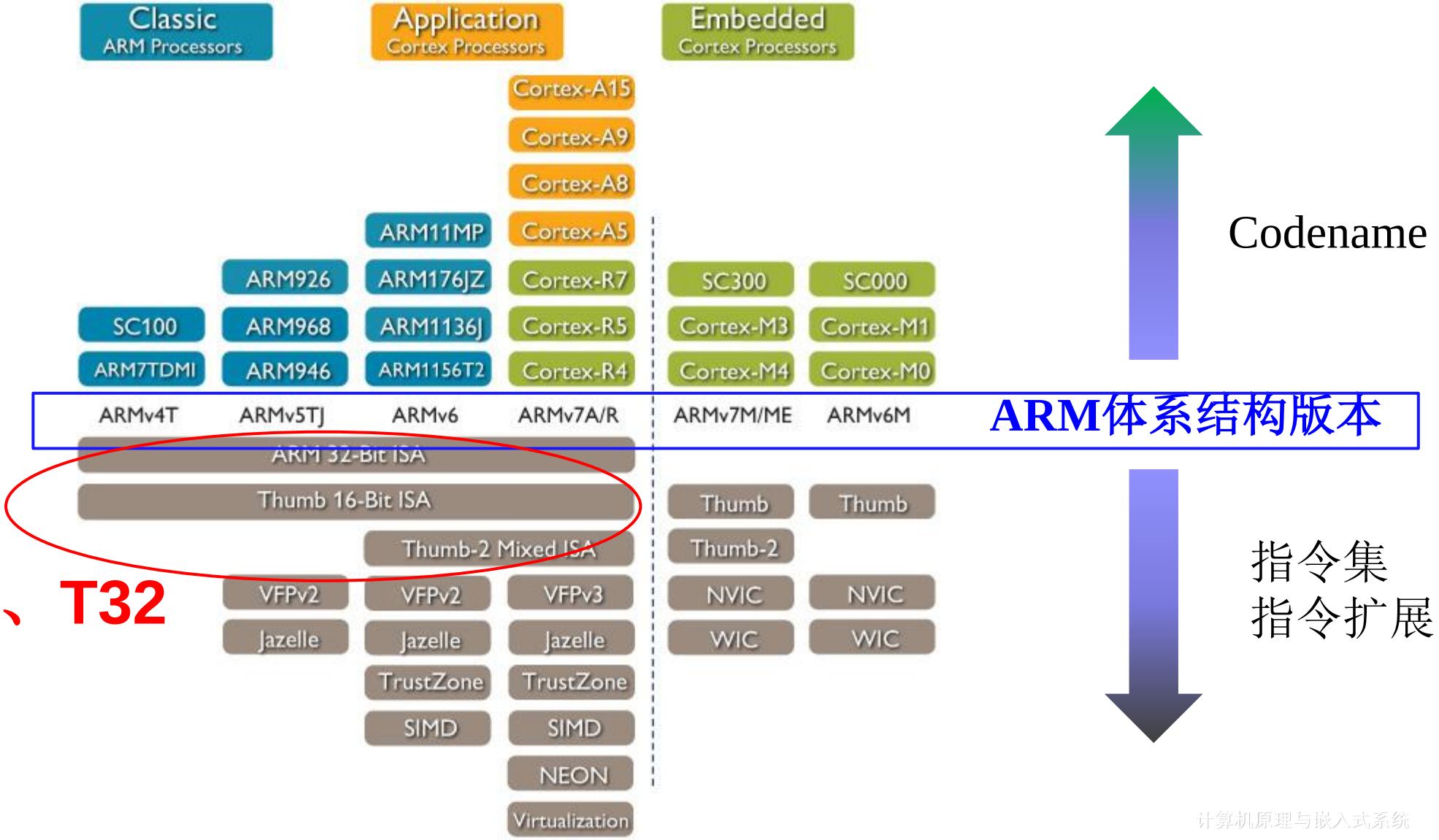
- 对于一般的数据处理和I/O控制，Cortex-M0/M0+完全可以胜任（性能达2.15 Core Mark/MHz）←稍后回顾评测标准
- 如果需要进行复杂的数据处理和快速乘加运算，就应该升级到Cortex-M3/M4处理器
- 如果需要DSP功能，则应该选择Cortex-M4
- 如果还需要计算浮点数，Cortex-M4还应该选配FPU
- 需要特别指出的是：
  - 虽然Cortex-M3/M4支持的指令较多，但是C编译器也能生成高质量的代码，CMSIS-DSP库提供了较丰富的应用函数，现代大多数嵌入式软件都使用C或C++语言开发，一般的应用开发中无需过于关注汇编指令的细节！

计算机性能评测

此页PPT在 2.8 计算机性能评测 部分出现过

- 定量描述指标：速度，如各种“跑分”
- 早期指标：MIPS，对CISC和RISC不能客观评价
- 现在指标：基准测试（Benchmark Test）结果，如：
  - Whetstone和Dhrystone：前者包括浮点，后者只有整数，使用FORTRAN编写，代码量太小，与编译器有关
  - **CoreMark**：2009年由EEMBC（Embedded Microprocessor Benchmark Consortium）提出，用C语言编写，包含**嵌入式系统**常见的4种计算（矩阵、查找和排序、状态机和CRC），已成为嵌入式内核性能评测的事实标准
  - SPEC（Standard Performance Evaluation Corporation）测试，通用计算机使用最多的基准测试

小结：ARM处理器/ISA



A64、A32、T32

目录

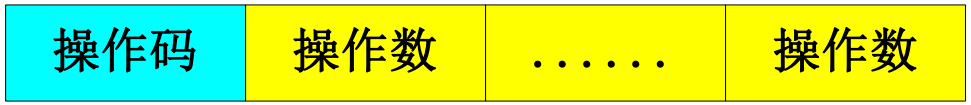
指令的概念

- 每种计算机都有一组**指令集**供用户使用，这组指令集就称为计算机的**指令系统**。
  - 机器指令：计算机能识别和执行的指令二进制代码，如：**0100 0110 1000**
  - 汇编指令：用助记符表示机器指令的操作码和操作数，如：**MOV R0, R1**

助记符	指令功能英文描述	指令功能中文描述
ADD	Add	加法
LDR	Load Register with word	将32比特存入寄存器
MOV	Move	数据移动
STR	Store Register word	将一个寄存器的数值存入存储器
SUB	Subtract	减法

指令组成

- 指令由操作码字段和操作数字段两部分组成



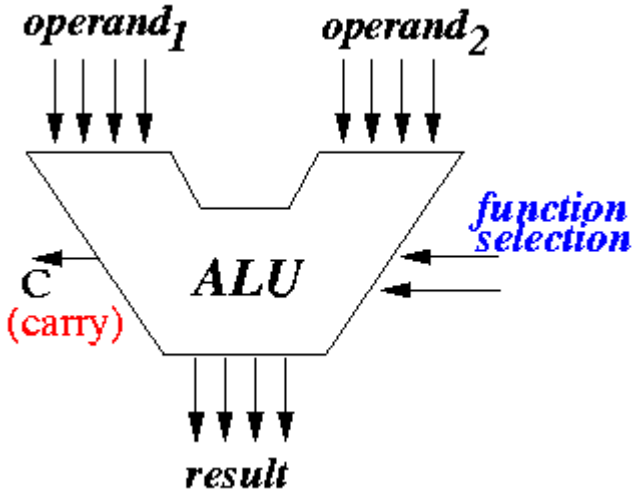
  - 操作码字段-----指示计算机要执行是什么操作
  - 操作数字段-----指令执行操作过程中所需要的操作数；该字段可以是：
  - 1）操作数本身
  - 2）操作数地址或是地址的一部分；
  - 3）指向操作数地址的指针；
  - 4）或其他有关操作数的信息（如寄存器名称）。

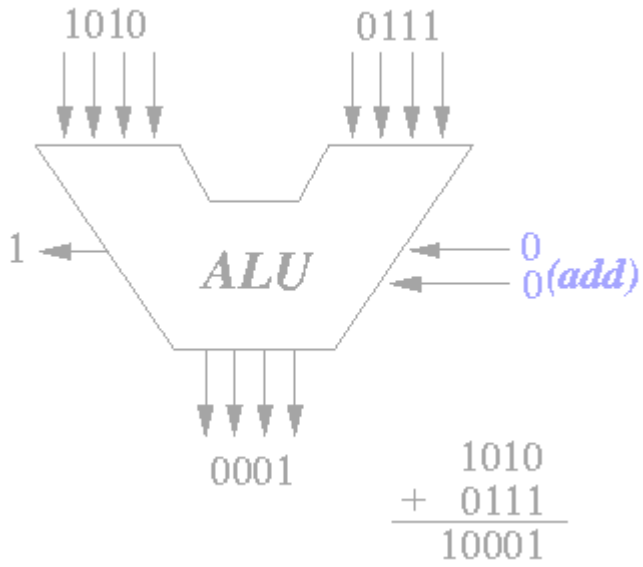
- **ARM大多数指令包括1～2个操作数，少数隐含多个操作数（如LDM指令）；还有若干指令无操作数。**

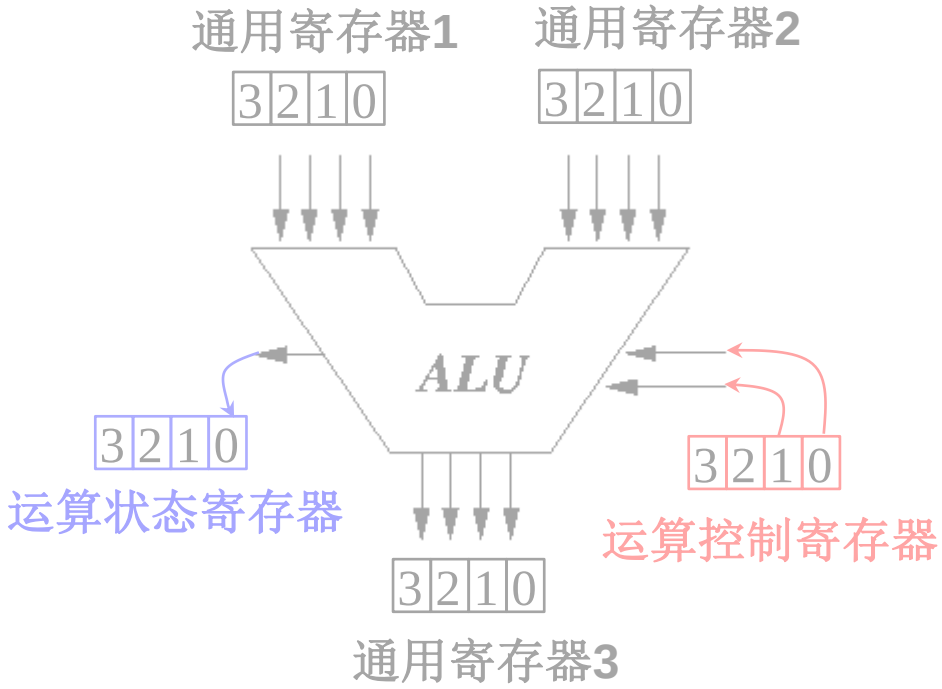
题外话：操作码需要多少位？
题外话：为什么需要寄存器？

此页PPT在 5.3 Cortex-M3/M4 的编程模型 部分出现过

- 若用D触发器构成1个寄存器的比特，D连接在哪？Q连接在哪？







T32的指令编码方式

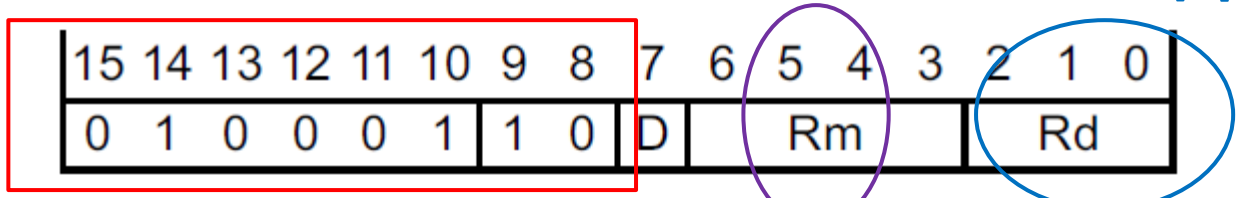
MOV Rd，Rm ;寄存器Rm内容传送给Rd

思考：既然16比特指令就可以实现MOV Rd，Rm，为什么还需要32比特的T3编码格式？

- ARMv7-A/R/M中有多种二进制编码格式
  - T1、T2、T3、T4、A1、A2
  - Encoding T1，16比特

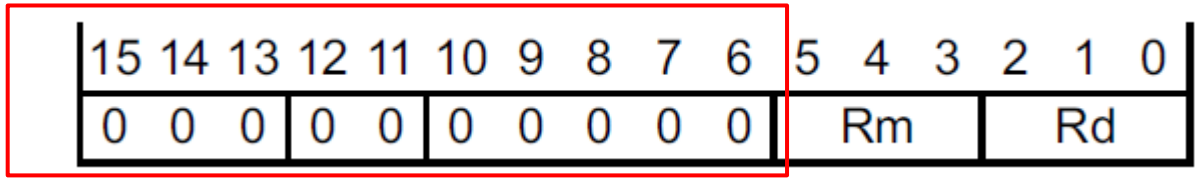
操作码

目标寄存器

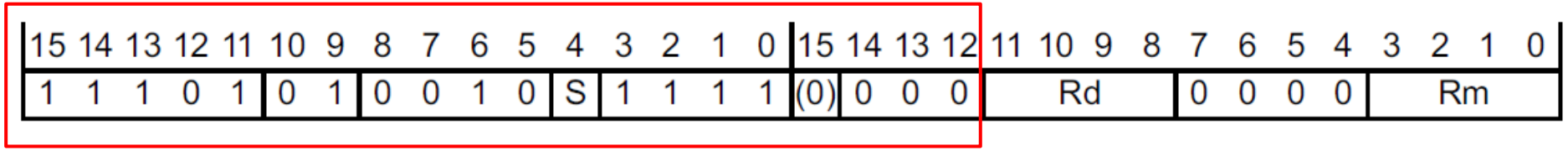


源操作数

  - Encoding T3，16比特



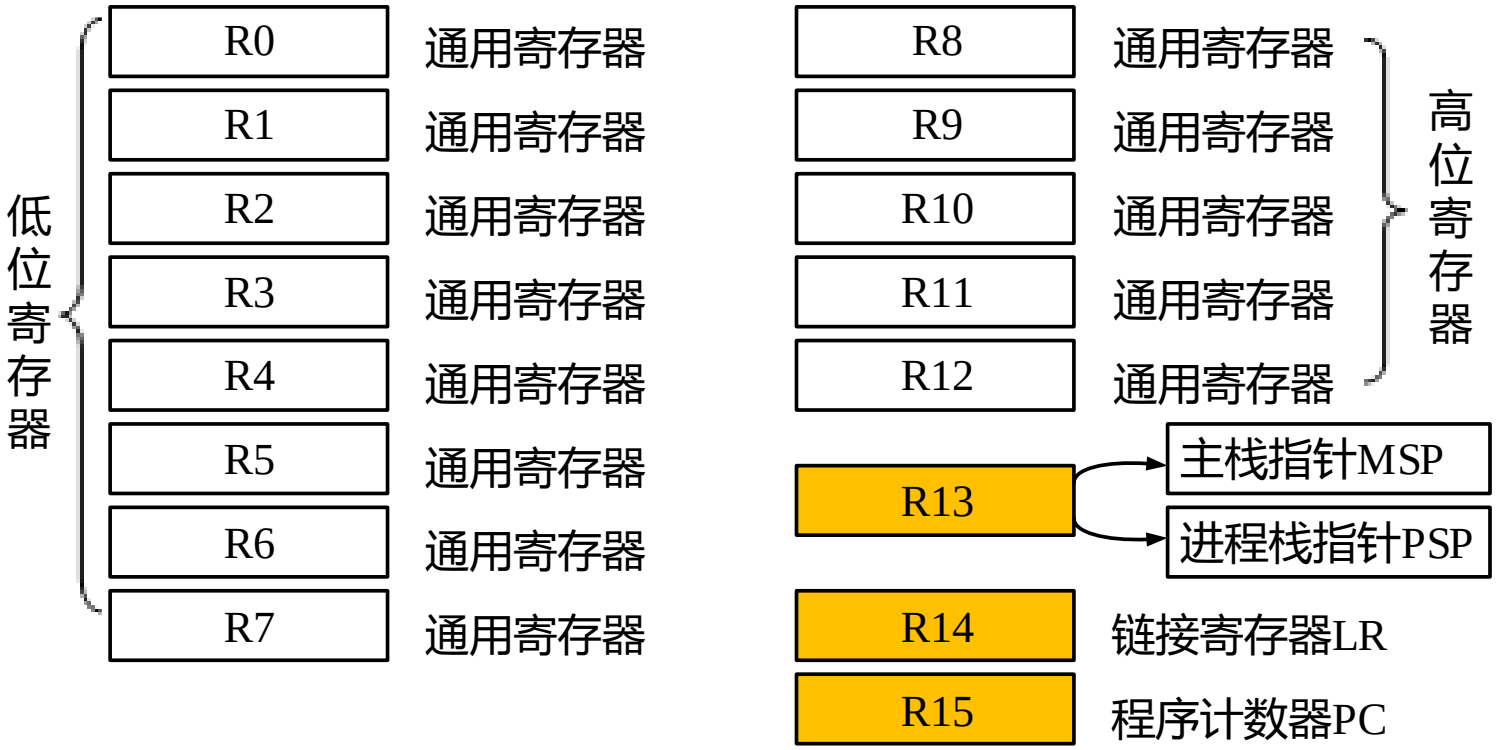
  - Encoding T3，32比特



Cortex-M处理器的常规寄存器

此页PPT在 5.3 Cortex-M3/M4的编程模型 部分出现过

Cortex-M3/M4处理器中常规寄存器的分组

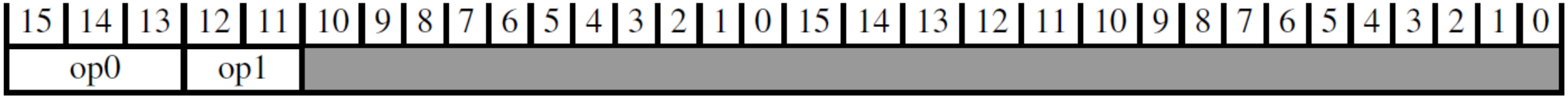


R0~R7，8个低位寄存器，许多16位Thumb指令只能访问低位寄存器

R8~R12，5个高位寄存器，可用于32位指令和少数几个16位指令

如何区分16位和32位的指令？

- T32的指令由半字对齐（halfword-aligned）的序列构成。若为16比特Thumb的指令，则指令中含有一个半字；若为32比特Thumb指令，则指令包含两个半字。通过半字中最高五个比特来区分是16比特指令还是32比特指令。

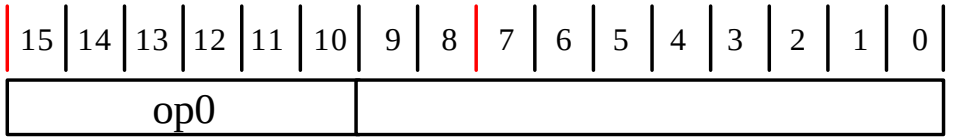


op0	op1	指令类型（subgroup）
!= 111	-	16比特的T32指令编码
111	!= 00	32比特的T32指令编码

三种情况下该半字是一个32比特指令的第一个半字：①0b11101，②0b11110，③0b11111。其他情况的半字均为16比特指令。

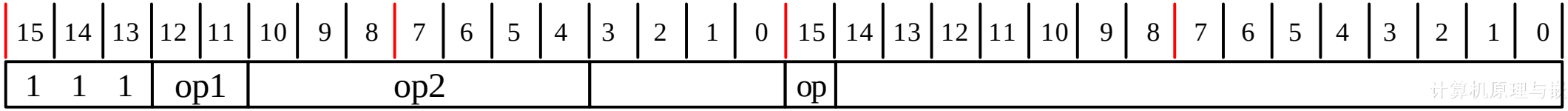
16位Thumb指令操作码格式

操作码	指令类别
00xxxx	基于立即数的指令如移位、加法、减法、比较和数据移动
010000	常规数据处理指令如位运算、移位、加法、减法、比较
010001	特殊的数据处理指令
01001x	基于PC的载入指令
0101xx	载入/保存（Load/store）单个数据的指令
011xxx	
100xxx	
10100x	产生基于PC（PC-relative）的地址
10101x	产生基于SP（SP-relative）的地址
1011xx	一些不容易分类的杂类指令
11000x	保存多个寄存器的数到存储器区域
11001x	从存储器区域载入多个数到多个寄存器
1101xx	条件跳转指令
11100x	无条件跳转指令



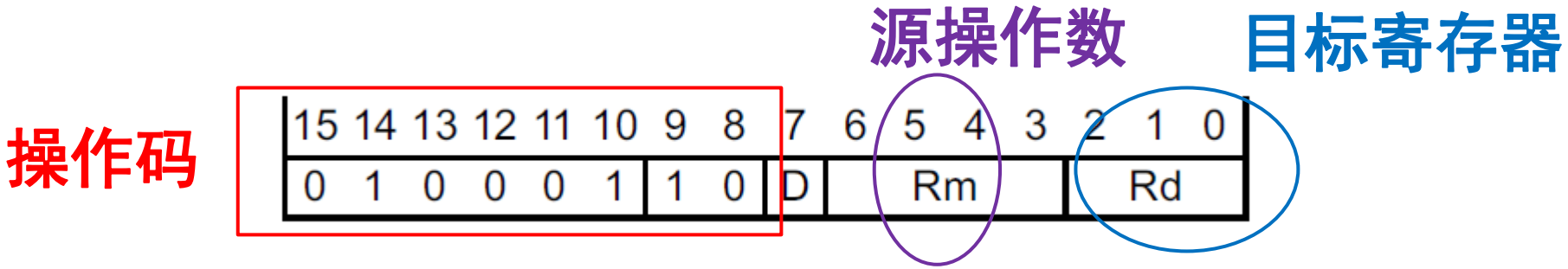
32位Thumb指令操作码格式

op1	op2	op	指令类别
01	00xx0xx	x	载入或存储多个数
01	00xx1xx	x	载入或存储双字，或者以独占方式执行载入或存储
01	01xxxxx	x	常规数据处理如位运算、移位、加法、减法、比较
01	1xxxxxx	x	协处理器指令
10	x0xxxxx	0	使用经过移位处理的立即数的数据处理指令
10	x1xxxxx	0	使用未经过移位处理的立即数的数据处理指令
10	xxxxxxx	1	分支控制指令和杂类指令
11	000xxx0	x	存储单个数据到存储器区域
11	00xx001	x	载入字节指令
11	00xx011	x	载入半字指令
11	00xx101	x	载入字指令
11	00xx111	x	未定义
11	010xxxx	x	寄存器处理指令
11	0110xxx	x	产生32比特结果的乘法、乘并累加指令
11	0111xxx	x	产生64比特结果的乘法、乘并累加指令
11	1xxxxxx	x	协处理器指令



小结：T32的指令编码方式

- 指令系统：指令的集合
- 指令由操作码字段和操作数字段两部分组成



- 机器指令：0100 0110 0000 1000
- 汇编指令：助记符，MOV R0, R1
- ARMv7-A/R/M中有多种二进制编码格式
  - T1、T2、T3、T4、A1、A2
  - 按照二进制机器指令比特串的前缀区分不同编码格式

目录

二进制机器码 vs. 汇编指令

- 直接用比特串表示的机器指令可读性很差，习惯上用汇编语言来表示机器指令。
  - 汇编语言是一种用于电子计算机、微处理器、微控制器或其他可编程器件的低级语言，亦称为符号语言。
- 在汇编语言中，用助记符代替机器指令的操作码，用地址符号或标号代替指令或操作数的地址。
- 不同的设备中，汇编语言对应着不同的机器语言指令集，通过汇编过程转换成机器指令。特定的汇编语言和特定的机器语言指令集是一一对应的，不同平台之间不可直接移植。
- 统一汇编语言
  - ARMv7之后，为了增强兼容性，提出了采用统一汇编语言（UAL，Unified Assembly Language）进行机器指令的描述。使用UAL后，16比特的汇编指令和32比特的汇编指令可以无缝的出现在同一份代码中，只是其编码格式不同。

T32指令的汇编语法

- 机器指令的要素：指令的功能、源操作数、目的操作数、操作数地址。
- ARM处理器汇编指令的通用格式如下：

<opcode>*[cond]*[q]*[S]*_<Rd>,_<Rn>*[,_Oprand2]*

  - 其中，<>内的参数是必选参数，而[]内参数是可选参数。
  - opcode，操作码，也称为助记符。
  - cond：condition，条件码（可选后缀），描述指令的执行条件。
  - q，可选后缀，指令宽度选择。
  - S，可选后缀，含S代表指令执行会更新APSR。
  - Rd，目标操作数，总是一个寄存器。
  - Rn，存放第一源操作数寄存器，该操作数必须是寄存器。
  - Oprand2，第二源操作数，不仅可以是寄存器，还能是立即数，而且能用经过偏移量计算的寄存器和立即数。

操作码的助记符示例

汇编语言所描述的机器指令的形式

标号 助记符 操作数1，操作数2，…;注释

MOV R0, R1 ;寄存器R1中的数值装载到寄存器R0

助记符	指令功能英文描述	指令功能中文描述
ADD	Add	加法
LDM	Load Multiple registers	将多个32位数存入多个寄存器
LDR	Load Register with word	将32位数存入寄存器
MOV	Move	数据移动
STM	Store Multiple registers	将多个寄存器的数值存入特定存储器区域
STR	Store Register word	将寄存器中32位数存入存储器
SUB	Subtract	减法

操作码的助记符示例

- 汇编语言所描述的机器指令的形式

  标号 助记符 操作数1，操作数2，…;注释

  例如：

  label MOV R0, R1 ; R1中的数值装载到R0

  - **标号“label”**是指令的符号地址，属于伪指令内容
  - 不同汇编工具的语法有稍许区别，但助记符和汇编指令相同，伪指令、标号和注释语法可能会有差异
  - 例如，对于GNU工具链，上例的语法为：

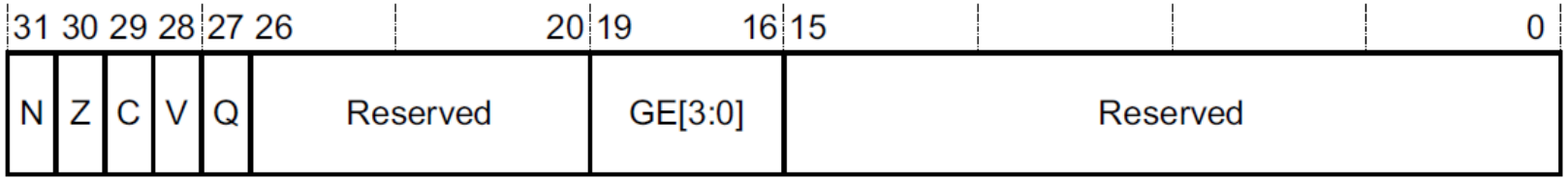
    label: MOV R0, R1 /* 注释 */
  - 例如，对于gcc，插入注释的语法为：

    label: MOV R0, R1 @ 注释

T32的条件执行指令

ARMv7开始，采用PSR整合了更早版本中APSR、EPSR和IPSR三个状态寄存器的位域，详见第5章。本章沿用ARMv7-M技术手册描述，不对PSR或APSR做描述上的区分。

- T32中多数Thumb指令可以根据应用程序状态寄存器APSR中的标志位决定当前指令是否被执行。
- APSR (Application Program Status Register)
  - N bit[31] 负数标志位。N==1表示上一次运算结果为负数。
  - Z bit[30] 零标志位。Z==1表示上一次运算结果为零。
  - C bit[29] 进位标志位。C==1表示上一次运算产生了进位。
  - V bit[28] 溢出标志位。V==1表示上一次运算产生了溢出。
  - Q bit[27] 饱和标志位。Q==1表示上一次运算有饱和操作。
  - GE[3:0] bits[19:16]，DSP扩展指令中SIMD类指令指示。



条件码(Condition codes)含义示例

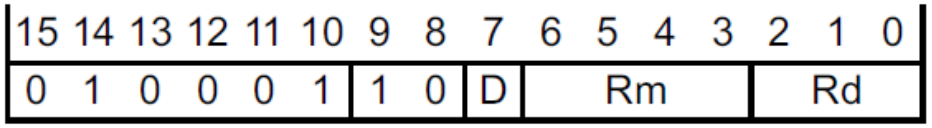
条件码助记符对应汇编语法中的后缀

条件码	助记符	含义	标志位取值
0000	EQ	等于（Equal）	Z == 1
0001	NE	不等于（Not equal）	Z == 0
0010	CS	产生了进位（Carry set）	C == 1
0011	CC	进位为零（Carry clear）	C == 0
0100	MI	负数（Minus, negative）	N == 1
0101	PL	整数或零（Plus, positive or zero）	N == 0
0110	VS	溢出（Overflow）	V == 1
0111	VC	没有溢出（No overflow）	V == 0
……		……	

MOVEQ R0, R1 ;Z==1时把寄存器R1中的数值装载到寄存器R0

不更新APSR的MOV指令

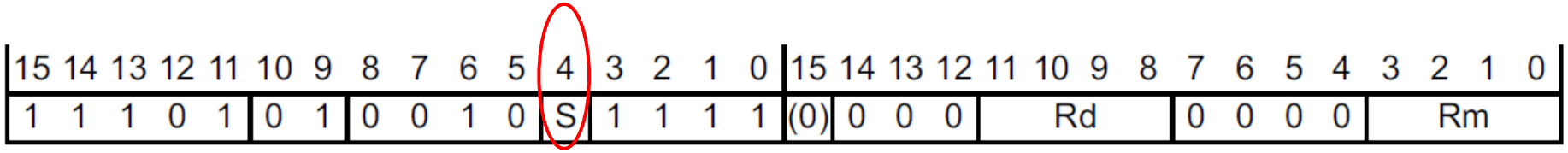
MOV R0, R1 ;寄存器R1中的数值装载到寄存器R0



T1格式默认不更新APSR（不支持）

T1格式(16bits)“0100 0110 0000 1000”

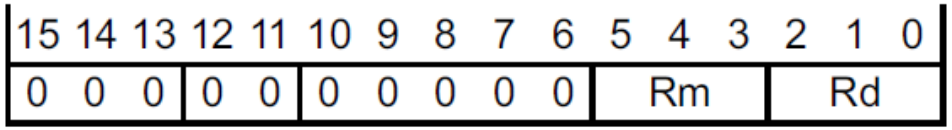
T3格式可设置是否更新APSR



T3格式(32bits)“1110 1010 0100 1111 0000 0000 0000 0001”

更新APSR的MOV指令

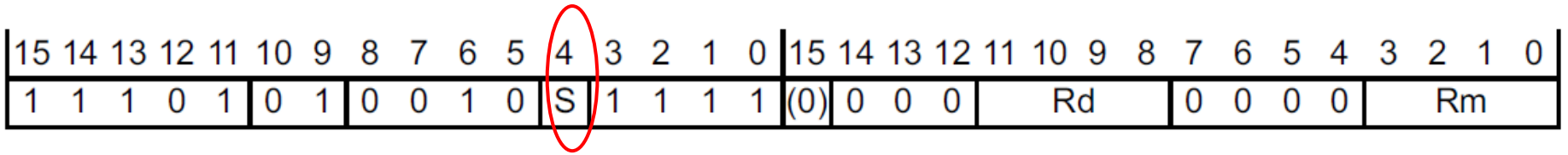
MOVS R0, R1 ;寄存器R1中的数值装载到寄存器R0，更新APSR



T2格式默认更新APSR

T2格式“0000 0000 0000 1000”

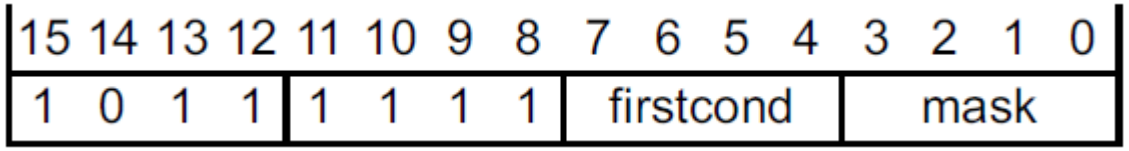
T3格式可设置是否更新APSR



T3格式“1110 1010 0101 1111 0000 0000 0000 0001”

含有条件码(后缀)的MOV指令

含有条件码(后缀)的MOV指令需要和IT指令配合使用



IT指令只有T1格式

IT EQ的T1格式编码为“1011 1111 0000 1000”

```
IT EQ          ;随后的一条指令是条件执行的
MOVEQ R0, R1   ; Z==1时执行，否则不执行
```

IT NE的T1格式编码为“1011 1111 0001 1000”

```
IT NE          ;随后的一条指令是条件执行的
MOVNE R0, R1   ; Z==0时执行，否则不执行
```

IT 指令的汇编语法

语法格式： **IT{x{y{z}}} cond**

- cond 为IT块中第一条指令使用的条件码；
- x 指定IT块中第二条指令的是否执行的开关；
- y 指定IT块中第三条指令的是否执行的开关；
- z 指定IT块中第四条指令的是否执行的开关。

几种加了后缀的IT指令（还有其他形式如ITEEE）

IT，“If-Then”，下一条指令是条件执行的；

ITT，“If-Then-Then”，下两条指令是条件执行的；

ITE，“If-Then-Else”，下两条指令是条件执行的；

ITTE，“If-Then-Then-Else”，下三条指令条件执行；

ITTEE，“If-Then-Then-Else-Else”，下四条指令条件执行。

条件执行的MOV指令示意


```
IT EQ              ;随后的一条指令是条件执行的
MOVEQ R0, R1       ; Z==1时执行，否则不执行

ITT EQ             ;随后的两条指令是条件执行的
MOVEQ R0, R1       ; “Z == 1”时执行，否则不执行
MOVEQ R2, R3       ; “Z == 1”时执行，否则不执行

ITTEE EQ           ;随后的四条指令是条件执行的
MOVEQ R0, R1       ; “Z == 1”时执行，否则不执行
MOVEQ R4, R5       ; “Z == 1”时执行，否则不执行
MOVNE R2, R3       ; “Z == 0”时执行，否则不执行
MOVNE R6, R7       ; “Z == 0”时执行，否则不执行
```

小结：T32指令的汇编语法

- 机器指令的要素：指令功能、源操作数、目的操作数、操作数地址
- ARM处理器汇编指令的通用格式如下：

<opcode>[cond][q][S]_<Rd>,_<Rn>[,_Oprand2]

操作码[条件码][可选后缀，指令宽度][可选后缀，更新PSR]_<目标操作数>,_<第一源操作数>[,_第二源操作数]

MOV R0, R1 ;寄存器R1中的数值装载到寄存器R0

MOVS R0, R1 ;寄存器R1中的数值装载到寄存器R0，更新APSR

MOVEQ R0, R1 ;Z==1时把寄存器R1中的数值装载到寄存器R0

```
ITT EQ              ;随后的两条指令是条件执行的
MOVEQ R0, R1        ; “Z == 1”时执行，否则不执行
MOVEQ R2, R3        ; “Z == 1”时执行，否则不执行
```

目录

寻址和寻址方式

- 寻址——根据指令内容确定操作数地址的过程。
- 寻址方式——如何寻找操作数的方法。不同寻址方式实质上是构成操作数地址的方法不同。
- 寻址包括两种情形：
- ①确定当前指令中的操作数地址，称作操作数寻址或简称数据寻址；
- ②确定下一条待执行指令的地址，常称作指令寻址。

- **32位或64位的处理器**
  - 用32/64位宽的寄存器存放地址，可寻址范围达2^{32}或2^{64}。
- **早期16位的处理器**
  - 用16比特寄存器来存放地址，该地址不够表述整个存储器空间，故采用两个16位寄存器组合生成一个操作数的地址。

操作数的存放

- 操作数的存放不外乎四种情况：

1）操作数包含在指令中

- 这种操作数称为立即数，例：MOV R0 , #0x08

2）操作数包含在CPU的内部寄存器中

- 例： ADD Rd, Rn, Rm ;Rd= Rn+ Rm
- 操作数存放在Rn, Rm寄存器中

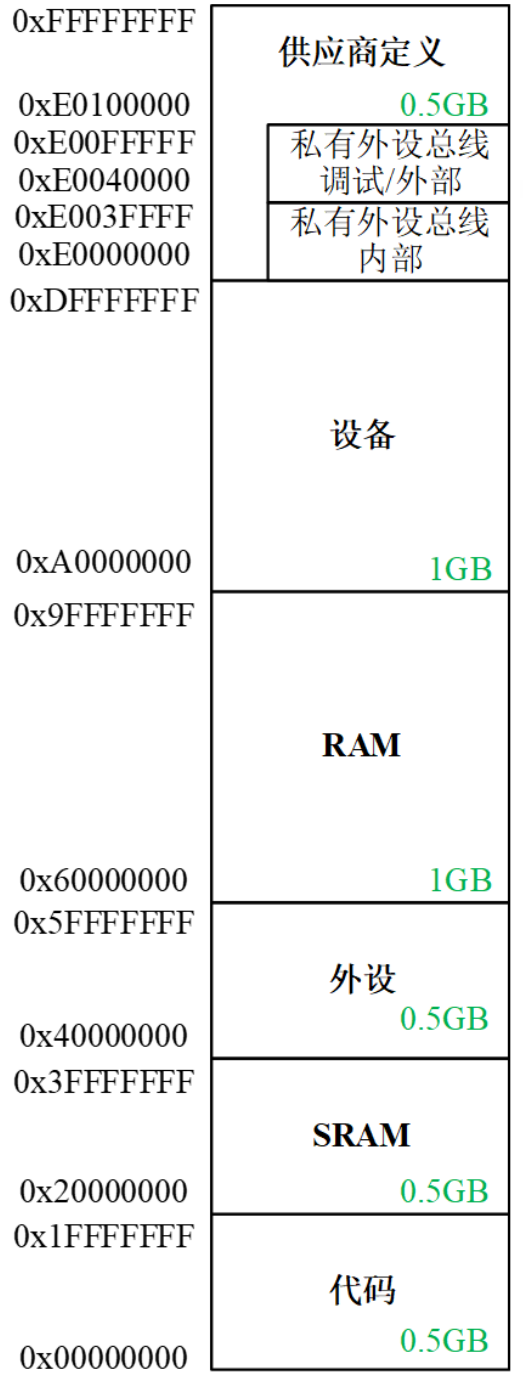
3）操作数在存储器数据区

- 操作数字段包含着此操作数的地址信息

4）操作数在存储器代码区

- 操作数字段包含着此操作数的地址信息

第5章：ARM处理器代码区和数据区是分开的



寻址方式分类（Addressing modes）

操作数位置	可能的寻址方式	英文原称
内含在指令中	1) 立即数寻址	Immediate addressing
存放在寄存器中	2) 寄存器寻址	Register addressing
	3) 寄存器移位寻址	Shifted register addressing
在存储器数据区	4) 寄存器间接寻址	Register addressing
	5) 寄存器偏移寻址	Offset addressing
	6) 前变址寻址	Pre-indexed addressing
	7) 后变址寻址	Post-indexed addressing
	8) 多寄存器寻址	multiple registers
	9) 堆栈寻址	SP addressing
在存储器代码区	10) PC相对寻址	PC-related addressing (Literal)

1)立即数寻址

- 立即数寻址也叫立即寻址，是一种特殊的寻址方式，操作数本身包含在指令中，只要取出指令也就取到了操作数。

```
MOV R0,#66              ;将立即数66传送到寄存器R0
ADD R0, R0, #66  ;R0+66→R0
SUB R0, R0, #0x33       ;R0-0x33 → R0
```

- Cortex-M3或M4支持的T32指令集中，指令长度要么是16比特，要么是32比特，指令长度有限→立即数取值只能在一定范围内

立即数的使用限制

ARM. (2018, 2019-12-28). *ARM®v7-M Architecture Reference Manual Reference Manual*. Available: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0204j/Cihcdbca.html

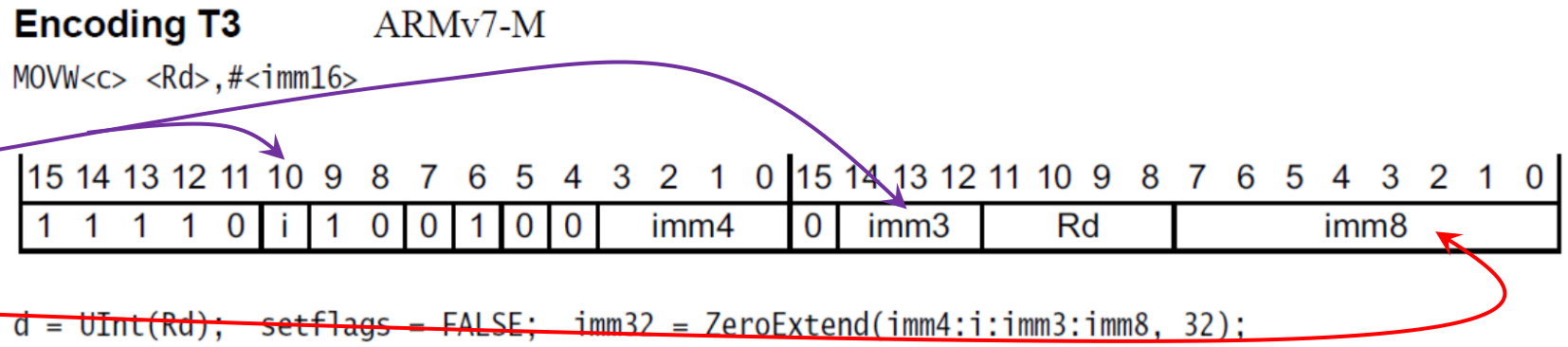
T32指令集中合法的立即数需要满足一定的规则。

①一个可以由八比特数经过循环移位得到的数值是合法的立即数。

- 如，所有八比特的立即数都是合法的。
- 0x01FC是合法的，把它写成二进制形式为：0001 1111 1100，可看出使用0xFE（二进制形式为：1111 1110）这个八位的常数在16位寄存器中循环左移一位就可以得到0X01FC。
- 0x07FC就是非法的立即数，因为它无法通过对一个八比特二进制数经过循环移位得到。

②满足格式“0x00XY00XY”或“0xXY00XY00”或“0xXYXYXYXY”的数，其中X和Y为16进制数字。

以MOV为例，含立即数时，T3编码后的二进制比特串中除了8比特的数，还有4比特用来指示用什么方式了生成最终的立即数

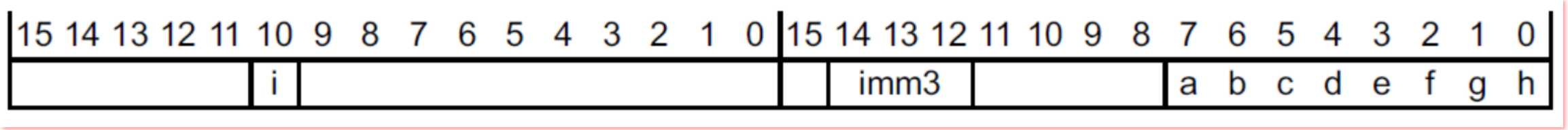


立即数的可能形式

i:imm3:a	<const> [a]
0000x	00000000 00000000 00000000 abcdefgh
0001x	00000000 abcdefgh 00000000 abcdefgh [b]
0010x	abcdefgh 00000000 abcdefgh 00000000 [b]
0011x	abcdefgh abcdefgh abcdefgh abcdefgh [b]
01000	1bcdefgh 00000000 00000000 00000000
01001	01bcdefg h0000000 00000000 00000000
01010	001bcdef gh000000 00000000 00000000
01011	0001bcde fgh00000 00000000 00000000
.	
.	
.	
11101	00000000 00000000 000001bc defgh000
11110	00000000 00000000 0000001b cdefgh00
11111	00000000 00000000 00000001 bcdefgh0

ARM. (2018, 2019-12-28). *ARM®v7-M Architecture Reference Manual Reference Manual*. Available: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0204j/Cihcdbca.html

a. In this table, the immediate constant value is shown in binary form, to relate abcdefgh to the encoding diagram. In assembly syntax, the immediate value is specified in the usual way (a decimal number by default).
b. UNPREDICTABLE if abcdefgh == 00000000.



2) 寄存器寻址

- 寄存器寻址就是利用寄存器中的数值作为操作数，也称为寄存器直接寻址。这种寻址方式在各类微处理器经常被采用，执行效率高。

```
ADD R0, R1, R2
;将R1和R2相加结果送入R0
;常常记为：R1+R2→R0
```

- 寄存器限定于通用寄存器，不可以是PC。

3) 寄存器间接寻址

- 寄存器间接寻址就是把寄存器中存放的数值作为操作数地址，通过这个地址去取得操作数，操作数本身存放在存储器中。如下两条指令均用到了寄存器间接寻址。

```
LDR R0, [R1]
; 以寄存器R1的值作操作数的地址，取得操作数后传送到R0
; 常常记为 [R1]→R0

ADD R0, R1, [R2]
; 基于寄存器R2间接寻址取得操作数后与R1相加，结果存入R0
; 常常记为 R1+[R2]→R0
```

4) 寄存器移位寻址

- 寄存器移位寻址（Shifts applied to a register）是ARM指令集特有的寻址方式。其寻址方式为：先由寄存器寻址得到操作数，对该操作数再进行移位操作后得到最终的操作数。

```
MOV R0, R2, LSL #3    ;R2<<3，→R0
MOV R0, R2, LSL R1    ;R2<<R1，→R0
```

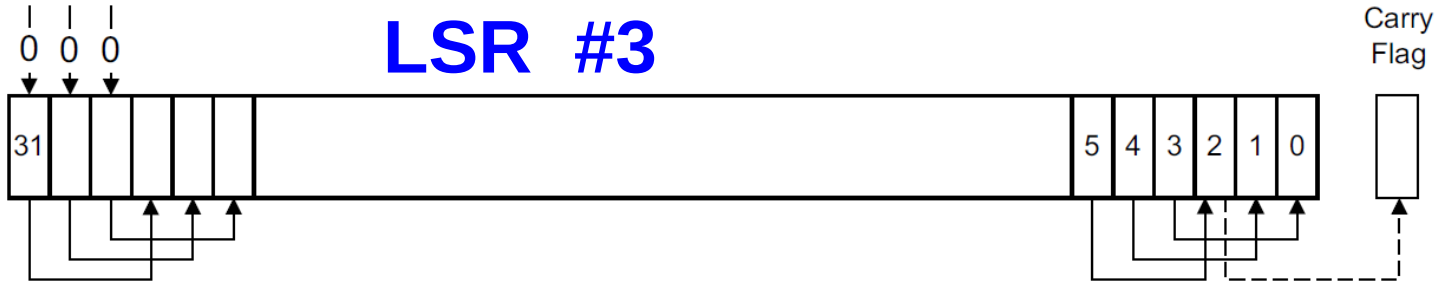
支持的移位方式（稍后解释为何没有算数左移）

- 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：带扩展的循环右移（Rotate Right eXtended）

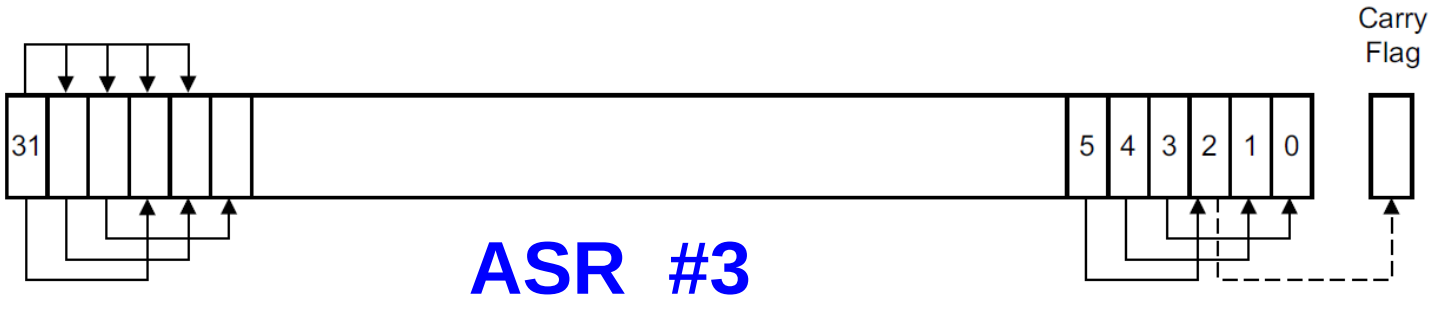
移位运算示例

ASR：算术右移（Arithmetic Shift Right
LSR：逻辑右移（Logical Shift Right）

- ARM处理器支持多种移位运算，因为在处理器电路中内置了桶形移位器。寄存器移位寻址也是建立在桶形移位器电路基础之上的。



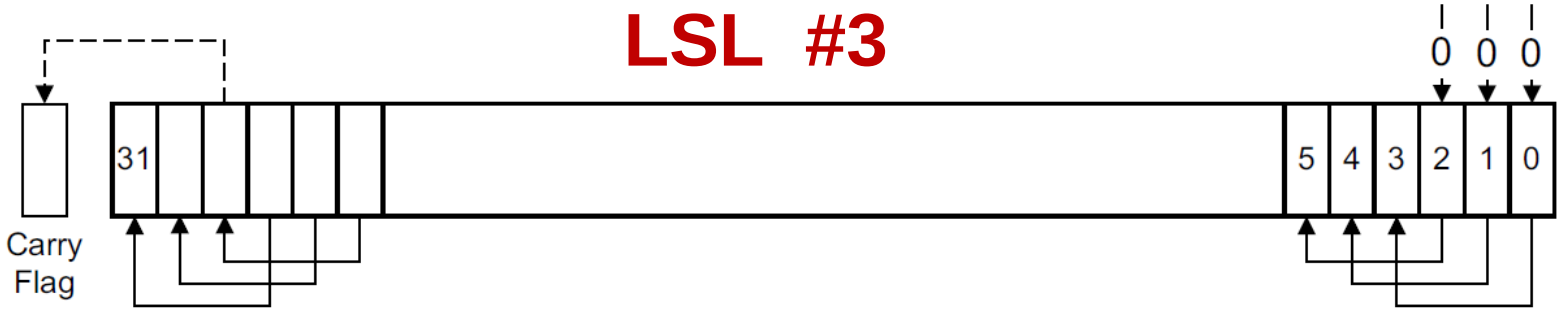
对于无符号型值，算术移位等同逻辑移位
对于有符号数，算术左移等同逻辑左移，算术右移补的是符号位，正数补0，负数补1



移位运算示例

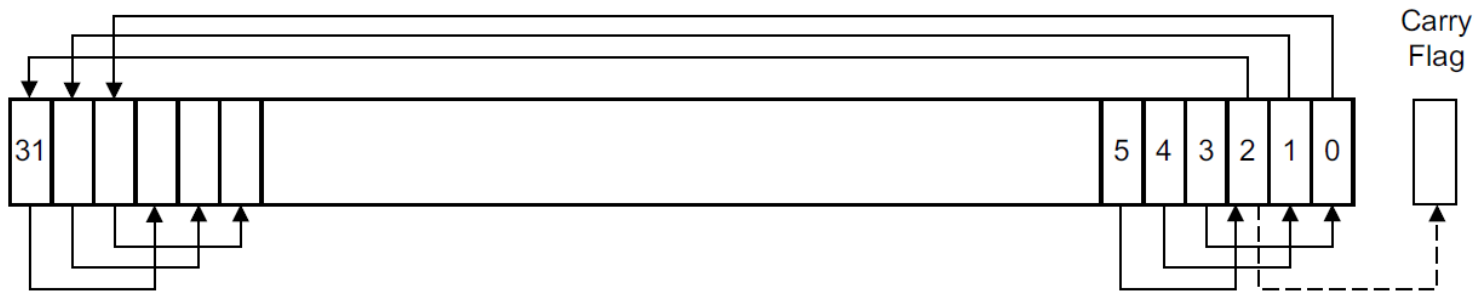
LSL：逻辑左移（Logical Shift Left）
ROR：循环右移（Rotate Right）
RRX：带扩展的循环右移（Rotate Right eXtended）

LSL #3

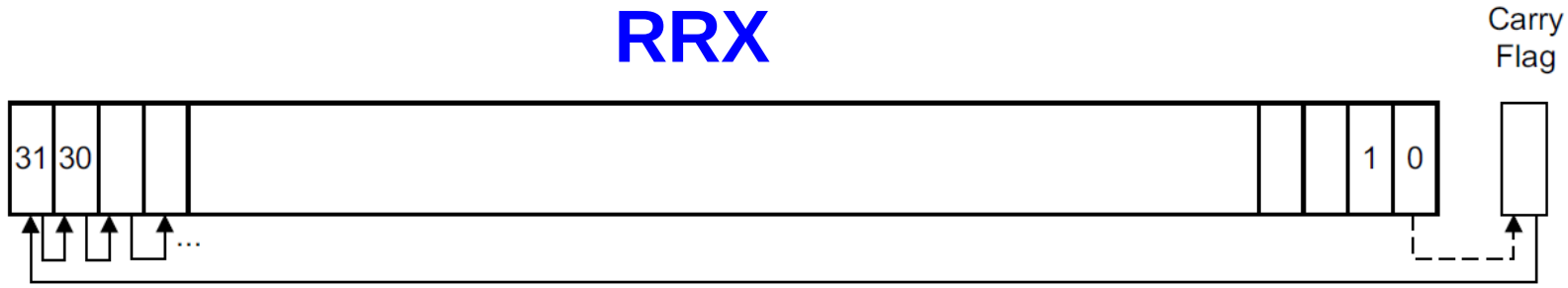


逻辑左移=算数左移

ROR #3



RRX



5) 寄存器偏移寻址

- 可以对前述寄存器间接寻址进行拓展，操作数地址由一个寄存器中存放的数值与指令中给出的地址偏移量相加得到。
- 其汇编语法为“**opcode Rd [<Rn>,<offset>]**”。

- 通常把存放在寄存器中的地址信息称作**基址（base address）**，该寄存器称作**基址寄存器**。而指令中给出的地址**偏移量（offset）**存在如下三种不同的形式。
  - 立即数（Immediate）
  - 寄存器（Register）
  - 寄存器移位（Scaled register）

寄存器偏移寻址的不同形式

LDR（Load Register with word）
STR（Store Register word）

```
LDR R0, [R1, #4]
;将R1中的值加4形成操作数的地址，取得的操作数送入R0
LDR R0, [R1, R2]
;R1中的值加上R2中的值形成操作数地址，取得的操作数送入R0
STR R0, [R1, #-4]
; R1中的数值减4作为操作数地址，把R0中的数据存放到这个地址中
```

```
LDR <Rt>, [<Rn>{,#<imm>}]
LDR <Rt>,[<Rn>,<Rm>]
LDR <Rt>, [<Rn>, <Rm> {, LSL #<shift>}]
```

其中“<Rn>”表示基址寄存器，而“<offset>”表示偏移量，偏移量可以是：
①5位立即数（<imm5>）或8位立即数（<imm8>）或12位立即数（<imm12>）；
②寄存器值<Rm>；
③寄存器值移位，如<Rm>, LSL #<shift>，#<shift>表示移位的位数。

6) 前变址寻址（Pre-indexed addressing，也译作前序寻址）

- 执行指令时自动把基址与偏移加和形成的操作数地址写回到基址寄存器中。
- 其汇编语法为“opcode Rd [<Rn>,<offset>]!”
- 如下四条指令分别使用了寄存器偏移寻址、前变址寻址、寄存器偏移寻址、前变址寻址

```
LDR R0, [R1, #4]
;将R1中的值加4形成操作数的地址，取得的操作数送入R0
LDR R0, [R1, #4]!
;与上一条指令的区别：“!”表示指令执行后操作数地址存入R1
LDR R0, [R1, R2]
;R1中的值加上R2中的值形成操作数地址，取得的操作数送入R0
LDR R0, [R1, R2]!
; 与上一条指令的区别：“!”表示指令执行后操作数地址存入R1
```

7) 后变址寻址（Post-indexed addressing，也译作后序寻址）

- 在执行指令的时候，操作数地址从基址寄存器获取，指令执行后再将操作数地址加上偏移量生成一个新的地址，并将该新地址写入基址寄存器。
- 其汇编语法为“opcode Rd [<Rn>],<offset>”
- 如下三条指令分别使用了寄存器偏移寻址、前变址寻址、后变址寻址方式。

```
LDR R0, [R1, #4]
;将R1中的值加4形成操作数的地址，取得的操作数送入R0
LDR R0, [R1, #4]!
;与上一条指令的区别：“!”表示指令执行后操作数地址存入R1
LDR R0, [R1], #4
;R1中的值做操作数地址，取得操作数送入R0，R1中数值加4
```

8) 多寄存器寻址

- 从一块连续的存储器区域装载多个数据到多个寄存器中。
- **LDM（Load Multiple registers）使用此寻址方式。**
- 语法：**LDM {addr_mode} <Rn>{!}, <registers>**
  - <Rn>为基址寄存器
  - <registers>为需要载入数据的的寄存器集合
  - 可选项{!}表示需要将修改后的地址写入基址寄存器<Rn>
  - 可选后缀{addr_mode}可选择如下四种方式。
    - IA (Increment address After each access)，取操作数后基址寄存器递增
    - IB (Increment address Before each access)，取操作数前基址寄存器递增
    - DB (Decrement address Before each access)，取操作数前基址寄存器递减
    - DA (Decrement address After each access)，取操作数后基址寄存器递减

说明：Cortex-M3/M4只支持IA和DB

装载多个数据到多个寄存器中

- 例如，如下格式指令可以完成四个寄存器数据的载入。其中R1载入的数据的存储器地址是[R0]，即R0中的数值为操作数的地址；而R2载入的数据在存储器中的地址是[R0+4]；R3载入的数据在存储器中的地址是[R0+8]；R4载入的数据在存储器中的地址是[R0+12]。

- **LDMIA R0!, {R1, R2, R3, R4}**

- 该指令的后缀IA表示在每次执行完加载/存储操作后，R0按字长度增加，对于32位的ARM指令，每次地址的增加和减少的单位都是4个字节单位。因此，该指令可将连续存储单元的32比特数值传送到R1~R4。在使用多寄存器寻址指令时，寄存器集合的顺序如果由小到大的顺序排列，可以使用“-”连接，否则用“,”分隔书写。下面的指令完成和上面一条指令相同的功能。

- **LDMIA R0!, {R1-R4}**

IA (Increment address After each access)
IB (Increment address Before each access)
DB (Decrement address Before each access)
DA (Decrement address After each access)

保存多个寄存器中存储区连续区域

- 多寄存器寻址是ARM处理器较有特色的功能。需要注意的是，在很多ARM的资料中，也把上述多寄存器寻址方式称作块拷贝寻址。这种称呼更加突出多个寄存器数据批量复制的特点。以下为另外四个示例。
- **STMIA R0!, {R1-R7}** IA (Increment address After each access)
  - ;R1~R7的数保存到R0指向的地址，每取一个数后R0递增4
- **STMIB R0!, {R1-R7}** IB (Increment address Before each access)
  - ;R1~R7的数保存到R0指向的地址，每取一个数前R0递增4
- **STMDA R0!, {R1-R7}** DB (Decrement address Before each access)
  - ;R1~R7的数保存到R0指向的地址，每取一个数后R0递减4
- **STMDB R0!, {R1-R7}** DA (Decrement address After each access)
  - ;R1~R7的数保存到R0指向的地址，每取一个数前R0递减4

9) 堆栈寻址

- 如果把多寄存器寻址方式中LDM或STM指令中的基址寄存器更换为堆栈指针寄存器SP，并添加可选项{!}（意为每次存/取操作数就更新一下SP），则寻址方式变成堆栈寻址。

```
STMFD SP!, {R1-R7}
;将R1~R7寄存器中的数压入堆栈
LDMFD SP!, {R1-R7}
;将堆栈中的数取出存入R1~R7寄存器
```

IA (Increment address After each access)
IB (Increment address Before each access)
DB (Decrement address Before each access)
DA (Decrement address After each access)

- STMFD指令则与前述STMDB指令格式相同，区别在于STMFD指令中基址寄存器为SP。

关于ARM的堆栈工作方式

第5章内容回顾

Cortex-M系列处理器只能使用满递减类型

- 依据堆栈指针位置分类
  - 当堆栈指针指向最后压入堆栈的数据时，称满堆栈（Full Stack）
  - 当堆栈指针指向下一个将要放入数据的空位置时，称空堆栈（Empty Stack）
- 根据堆栈的生成方式分类
  - 递增堆栈（Ascending Stack）：堆栈由低地址向高地址生成
  - 递减堆栈（Descending Stack）：堆栈由高地址向低地址生成
- 从而形成了四种类型的堆栈工作方式
  - **满递减堆栈（Full descending）**：堆栈首部是高地址，堆栈向低地址增长。栈指针总是指向最后一个元素。注意，最后一个元素是最后压入的数据。
  - 空递减堆栈（Empty descending）：堆栈首部是高地址，堆栈向低地址增长。栈指针总是指向下一个将要放入数据的空位置。
  - 满递增堆栈（Full ascending）：堆栈首部是低地址，堆栈向高地址增长。栈指针总是指向堆栈最后一个元素。
  - 空递增堆栈（Empty ascending）：堆栈首部指向低地址，堆栈向高地址增长。栈指针总是指向下一个将要放入数据的空位置。

10) PC相对寻址

- **PC相对（PC-relative）寻址是一种特殊的基址变址寻址，常简称为相对寻址。以程序计数器（PC）寄存器的当前值作为基地址，指令中的地址标号作为偏移量，将两者相加获得操作数的地址。**
- **偏移量有两种表示方法**
  - 汇编语句标号（Label）
  - 存储器代码区数据块相对当前代码的位置
    - 被称作文本池（literal pool）方式
    - 如果指令需要使用一个4字节的常量数据（可以是内存地址或数字常量），为了解决指令长度受限问题，编译器或汇编器在代码区中分配一块内存，保存这个4字节的数据常量。指令执行时，再使用一条指令把这个常量加载到寄存器中参与运算

PC相对寻址示例

用BL指令为跳转到“MY_SUB”标号所对应的语句

```
   BL   MY_SUB   ;相对寻址，跳转到MY_SUB处执行
   ;……;其他指令
MY_SUB
   ;……;其他指令
```

用ADR（Address to Register）指令获取PC相对寻址结果

```
start MOV R0, #10
   ;……;其他指令
   ADR   R2，start   ;将标号（Label）为“start”语句的地址送入R2
```

利用LDR指令将相对当前代码位置后12字节位置的数据传送到R0寄存器。

```
LDR R0, [PC, #0xC]
```

小结：寻址方式分类

操作数位置	寻址方式	操作数地址
内含在指令中	1) 立即数寻址	直接从指令得到
存放在寄存器中	2) 寄存器寻址	即寄存器的地址
	3) 寄存器移位寻址	寄存器中存放的数值经过移位
在存储器数据区	4) 寄存器间接寻址	寄存器中存放的数值
	5) 寄存器偏移寻址	寄存器中存放的数值与偏移量相加
	6) 前变址寻址	基址寄存器中存放的数值与偏移量相加
	7) 后变址寻址	基址寄存器中存放的数值
	8) 多寄存器寻址	基址寄存器中存放的数值
	9) 堆栈寻址	堆栈指针寄存器SP
在存储器代码区	10) PC相对寻址	PC寄存器中存放的数值与偏移量相加

小结：操作数地址的获取（寻址）

- 1) 立即数寻址　　#imm
- 2) 寄存器寻址　　<Rn>
- 3) 寄存器移位寻址　　[<Rn> , LSL #imm]
- 4) 寄存器间接寻址　　[<Rn>]
- 5) 寄存器偏移寻址　　[<Rn>,<offset>]
- 6) 前变址寻址　　[<Rn>,<offset>]!
- 7) 后变址寻址　　[<Rn>],<offset>
- 8) 多寄存器寻址　　<Rn>{!}, <registers>
- 9) 堆栈寻址　　<SP>{!}, <registers>
- 10) PC相对寻址　　[PC, #imm] 或 Label

目录

6.4.1 处理器内的数据传送指令

- 微处理器内部不同电路单元之间来回传送数据
  - 将数据从一个寄存器送到另外一个寄存器
  - 通用寄存器和特殊功能寄存器之间传送数据
  - 将立即数送到寄存器

MOV, MOVS: Move
MRS: Move from Special Register to general register
MSR: Move from general register to Special Register
MVN, MVNS: Move NOT

指令	目的	源	操作
MOV	R4,	R0	从R0复制数据到R4
MOVS	R4,	R0	从R0复制数据到R4，并更新APSR标志位
MRS	R7	PRIMASK	将数据从 RPIMASK寄存器复制到R7
MSR	CONTROL	R2	将数据从R2复制到 CONTROL寄存器
MOV	R3,	#0x34	设置R3为0x34
MOVW	R6	#0x3434	设置R6为16位常量0x3434
MOVT	R6	#0x8787	设置R6的高16位为0x8787
MVN	R3	R7	将R7中数据取反后送至R3

6.4.2 存储器访问指令

- **ARM处理器提供了系列Load/Store指令用于存储器访问**

- **指令的寻址模式**
  - 立即数寻址、寄存器寻址、寄存器移位寻址、寄存器间接寻址、基址变址寻址、多寄存器寻址（块拷贝寻址）、堆栈寻址、相对寻址
- **指令中操作数类型**
  - 无符号：字节（8bits）、半字（16bits）、字（32bits）、双字（64bits）、多字
  - 有符号：字节（8bits）、半字（16bits）、字（32bits）、双字（64bits）
- **针对提升安全性的增强**
  - 特权访问等级的程序以非特权方式访问存储器数据…
- **针对提升并发控制能力的增强**
  - 若某个存储单元被多个应用任务甚至多个处理器共享…

读/写存储器单个数据的系列指令

Data type	Load	Store	Load unprivileged	Store unprivileged	Load exclusive	Store exclusive
32-bit word	LDR	STR	LDRT	STRT	LDREX	STREX
16-bit halfword	-	STRH	-	STRHT	-	STREXH
16-bit unsigned halfword	LDRH	-	LDRHT	-	LDREXH	-
16-bit signed halfword	LDRSH	-	LDRSHT	-	-	-
8-bit byte	-	STRB	-	STRBT	-	STREXB
8-bit unsigned byte	LDRB	-	LDRBT	-	LDREXB	-
8-bit signed byte	LDRSB	-	LDRSBT	-	-	-
Two 32-bit words	LDRD	STRD	-	-	-	-

ARM. (2018, 2019-12-28). *ARM®v7-M Architecture Reference Manual Reference Manual.* Available: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0204j/Cihcdbca.html

不同的操作数类型

- **ARMv7-M，存储器中可存放的数据类型：Byte、Halfword、Word**
- **读/写存储器的时候则支持如下数据类型**

- 32-bit指针
- 无符号/有符号的32-bit整数
- 无符号的16-bit或8-bit整数
  - 须零扩展
- 有符号的16-bit或8-bit整数
  - 须符号扩展
- 无符号/有符号的64-bit整数
  - 由2个寄存器保存的
- 多个32-bit数

数据类型	加载/读存储器	存储/写存储器
8位无符号	LDRB	STRB
8位有符号	LDRSB	
16位无符号	LDRH	STRH
16位有符号	LDRSH	
32位	LDR	STR
多个32位	LDM	STM
双字（64位）	LDRD	STRD
栈操作（32位）	POP	PUSH

零扩展和符号扩展

用途如C程序中强制类型转换

```
short  sh_a;
Int  i_b;
i_b = sh_a;
```

- 零扩展（zero-extended）即在高位补“0”
  - 例，8-bit数值“1000 1010”零扩展至16-bit“0000 0000 1000 1010”
- 符号扩展（sign-extended）操作用于保护有符号数的符号位
  - 例，8-bit数值“0000 1010”扩展至16-bit“0000 0000 0000 1010
  - 例，8-bit数值“1000 1010”扩展至16-bit“1111 1111 1000 1010”

```
short int a = -4;
short int b = 8;
print_binary(a);
print_binary(b);
unsigned short int c = a; //类型转换
unsigned int d = a;
print_binary(c);
print_binary(d);
return 0;
```

```
11111111 11111100

00000000 00001000

11111111 11111100

11111111 11111111 11111111 11111100
请按任意键继续...
FFFFFFFC
00000008
0000FFFC
FFFFFFFC
```

零扩展/符号扩展…以LDRSB为例

- 例，**LDRSB和LDRSH会对被加载数据自动执行符号位扩展**
  - LDRSB R7, 0x83 ;数据在被放到R7前会被转换为0xFFFF FF83
  - LDRSB R7, 0x03 ;数据在被放到R7前会被转换为0x0000 0003

- 讨论：低位宽的数转换至高位宽的数时，零扩展和符号扩展能保证数值不变吗？
  - 有符号数 0x83 → $[1000\ 0011b]_{补}$ → $[1111\ 1101b]_{原}$ → 0xFD、十进制-125
  - 0xFFFF FF83 → $[11…1\ 1000\ 0011b]_{补}$ → $[10…0\ 0111\ 1101b]_{原}$
    → 0x8000 007D、十进制-125

- 思考：如果是高位宽的数转换为低位宽情况又如何？如16-bit → 8-bit

不同寻址模式，以LDRB为例

- 偏移寻址（Offset addressing）模式

  LDRB Rd, [Rn, #offset]

  ;从存储器位置[Rn+ offset]读取字节

- 前变址（Pre-indexed addressing）寻址模式

  LDRB Rd, [Rn, #offset] !

  ;从存储器位置[Rn+ offset]读取字节，并将Rn+ offset写回Rn

- 后变址（Post-indexed addressing）寻址模式

  LDRB Rd, [Rn], #offset

  ;读取存储器[Rn]处的字节到Rd，然后更新Rn到Rn+ offset

- 寄存器移位寻址（Shifts applied to a register）模式

  LDRB Rd, [Rn, Rm, LSL #n]

  ;从存储器位置Rn+(Rm<<n)处读取字节

读/写存储器多个数据的系列指令

Instruction	Description
Load Multiple, Increment After or Full Descending	*LDM, LDMIA, LDMFD*
Load Multiple, Decrement Before or Empty Ascending	*LDMDB, LDMEA*
Pop multiple registers off the stack [a]	*POP*
Push multiple registers onto the stack [b]	*PUSH*
Store Multiple, Increment After or Empty Ascending	*STM, STMIA, STMEA*
Store Multiple, Decrement Before or Full Descending	*STMDB, STMFD*

a. 指令等价与LDM指令以SP作为基址寄存器（且更新基址寄存器）
b.指令等价与STMDB指令以SP作为基址寄存器（且更新基址寄存器）

多加载和多存储模式

第5章，四种类型的堆栈工作方式
满递减堆栈（Full descending）
空递减堆栈（Empty descending）
满递增堆栈（Full ascending）
空递增堆栈（Empty ascending）

Mnemonic	Operands	Brief description
LDM	Rn{!}, reglist	Load Multiple registers, increment after
LDMDB, LDMEA	Rn{!}, reglist	Load Multiple registers, decrement before
LDMFD, LDMIA	Rn{!}, reglist	Load Multiple registers, increment after
STM	Rn{!}, reglist	Store Multiple registers, increment after
STMDB, STMFD	Rn{!}, reglist	Store Multiple registers, decrement before
STMEA, STMIA	Rn{!}, reglist	Store Multiple registers, increment after

- LDM（Load Multiple registers）指令
- STM（Store Multiple registers）指令
- IA（Increment address After each access），取操作数后基址寄存器递增
- DB（Decrement address Before each access），取操作数前基址寄存器递减

堆栈操作：压栈和出栈

- 以下两条指令等价（Equivalent）

  PUSH<cond><q> <registers>

  STMDB<cond><q> SP!, <registers>

  - PUSH {R0, R4-R6, R8}
  - ;将R0、R4、R5、R6和R8压入栈中

- 以下两条指令等价（Equivalent）

  POP<cond><q> <registers>

  LDMIA<cond><q> SP!, <registers>

  - POP {R1, R2}
  - ;将栈中内容存入R1和R2

PC相对寻址模式

- 存储器访问还可以使用PC寄存器作为基地址，再加上偏移量形成待访问操作数地址。常用于将立即数加载到寄存器中，也可被称作文本池（Literal Pool）访问。

指令语法	描述
LDRB Rt, [PC, #offset]	利用PC偏移加载无符号字节到Rt
LDRSB Rt, [PC, #offset]	对字节数据进行有符号展开并利用PC偏移加载到Rt
LDRH Rt, [PC, #offset]	利用PC偏移加载无符号半字到Rt
LDRSH Rt. [PC, #offset]	对半字数据进行有符号展开并利用PC偏移加载到Rt
LDR Rt, [PC, #offset]	利用PC偏移加载字数据到Rt
LDRD Rt, Rt2, [PC, #offset]	利用PC偏移加载双字数据到Rt和Rt2

- 文本池（Literal Pool）的本质就是ARM汇编语言代码段（section）中的一块用来存放常量数据而非可执行代码的内存块。使用文字池的原因：例如，想要在一条指令中使用一个四字节长度的常量数据的时候，由于ARM指令是定长的，如T32指令集最多支持32比特指令长度，因而无法把这个四字节的常量数据编码在一条指令中。

非特权等级加载和存储
Load/Store with unprivileged access

- ARM处理器区分特权和非特权访问等级。运行于非特权访问等级的程序无法访问特权访问等级程序中的数据。ARM中提供了一组特殊的加载和存储指令，在特权访问等级程序中使用这些指令加载或保存的数据，如同非特权访问等级程序的访问效果。

指令语法	描述
LDRBT Rd, [Rn, #offset]	从存储器位置Rn+ offset读取字节
LDRSBT Rd, [Rn, #offset]	从存储器位置Rn+ offset读取有符号展开的字节
LDRHT Rd, [Rn, #offset]	从存储器位置Rn+ offset读取半字
LDRSHT Rd, [Rn, #offset]	从存储器位置Rn+ offset读取有符号展开的半字
LDRT Rd, [Rn, #offset]	从存储器位置Rn+ offset读取字
STRBT Rd, [Rn, #offset]	往存储器位置Rn+ offset存储字节
STRHT Rd, [Rn, #offset]	往存储器位置Rn+ offset存储半字
STRT Rd, [Rn, #offset]	往存储器位置Rn+ offset存储字

- 如，特权等级的OS代码在API中向非特权等级的用户程序传递数据时，可使用这些特殊指令存取目标存储器位置，如果存取过程发生异常，则OS可知用户程序是无法访问这些存储器位置的，不能把这些存储器位置上存储的数据传递给普通的用户程序，需要调整。

5.4.8 排他（exclusive）访问

在讨论排他式访问前，回顾一下第5章中关于排他访问的定义

- 某个特定资源共享给多个用户使用时，常利用信号量来协调。特别是，若某共享资源只能满足一个用户使用时，被称为互斥体（Mutex）。
  - 在这种情况下，若某个资源被一个用户占用，它就会被锁定至这个用户，在锁定解除前无法用于其他用户。要创建互斥信号量，需要将互斥资源定义为锁定状态，以表示资源已被一个用户锁定。每个用户在使用资源前，需要先检查资源是否已被锁定，若未被使用，则设置为锁定状态，再开始使用资源。
- ARM7TDMI等传统的ARM处理器，锁定状态的访问由SWP指令执行，该指令可确保读写锁定状态的原子性，避免资源被两个用户同时锁定。而较新的ARM处理器，如Cortex-M3和Cortex-M4，读/写访问可由独立的总线执行。但是SWP指令的锁定流程中只适用于读写位于同一总线的场景，此时使用SWP指令无法保证存储器访问的原子性，故已被新的排他（exclusive）访问指令取代了。

排他式访问（exclusive access）

LDREX Rx, [Ry]

;排他加载，[Ry]→Rx，同时对Ry指向的内存区域标记独占访问，如果执行时发现已被标记，对指令执行没有影响。

STREX Rx, Ry, [Rz]

;排他存储，Ry→[Rz]，如果成功，则将Rx更新为0；若不成功，则将Rx置1。

指令语法	描述
LDREXB Rt, [Rn]	从存储器位置Rn排他读取字节，零扩展后存放至Rt
LDREXH Rt, [Rn]	从存储器位置Rn排他读取半字，零扩展后存放至Rt
LDREX Rt, [Rn, #offset]	从存储器位置Rn + offset排他读取字，存放至Rt
STREXB Rd, Rt, [Rn]	将Rt最低字节排他存储至存储器位置Rn，返回状态位于Rd中
STREXH Rd, Rt, [Rn]	将Rt低半字排他存储至存储器位置Rn，返回状态位于Rd中
STREX Rd, Rt, [Rn, #offset]	将Rt内容排他存储至存储器位置Rn + offset，返回状态位于Rd中
CLREX	清空一个排他访问标识位

小结：读/写存储器单个数据的系列指令

- 指令中操作数类型
  - LDRB、LDRSB、LDRH、LDRSH、LDR、POP
  - LDRD、LDM、LDMIA、LDMDB
- 指令中操作数地址获取方式
  - LDRB Rd, [Rn, #offset]
  - LDRB Rd, [Rn, #offset] !
  - LDRB Rd, [Rn], #offset
  - LDRB Rd, [Rn, Rmi, LSL #n)]
- 针对提升安全性的增强
  - LDRBT、LDRSBT、LDRHT、LDRSHT、LDRT
- 针对提升并发控制能力的增强（排他访问）
  - LDREXB、LDREXH、LDREX

6.4.3 算术运算指令

算术指令（可选后缀未列出）	操作
ADD Rd, Rn, Rm ;Rd= Rn+ Rm	加法运算
ADD Rd, Rn, # immed ;Rd= Rn+#immed	
ADC Rd, Rn, Rm ;Rd=Rn+Rm+进位	带进位的加法运算
ADC Rd, #immed ;Rd=Rd+# immed+进位	
ADDW Rd, Rn, #immed ;Rd= Rn +# immed	寄存器和12位立即数相加
SUB Rd, Rn, Rm ;Rd= Rn-Rm	减法
SUB Rd, # immed ;Rd= Rd-# immed	
SUB Rd, Rn,#immed ;Rd= Rn-#immed	
SBC Rd, Rn,# immed ;Rd=Rn-# immed-借位	带借位的减法
SBC Rd, Rn, Rm ;Rd=Rn-Rm-借位	
SUBW Rd, Rn, # immed ;Rd= Rn-# immed	寄存器和12位立即数相减
RSB Rd, Rn, # immed ;Rd=# immed-Rn	减反转
RSB Rd, Rn, Rm ;Rd= Rm- Rn	
MUL Rd, Rn, Rm ;Rd= Rn*Rm	乘法(32位)
UDIV Rd, Rn, Rm ;Rd= Rn/Rm	无符号和有符号除法
SDIV Rd, Rn, Rm ;Rd= Rn/Rm	

Add

ADC: Add with carry

Subtract

SBC: Subtract with carry

RSB: Reverse Subtract

UDIV: Unsigned Divide

SDIV: Signed Divide

MUL: Multiply

ADD指令用不同后缀

- 后缀不同，进位方式、对APSR标志位的影响不一样，对应的二进制机器码也不同。按照传统的Thumb汇编语法，在使用16位Thumb代码时，ADD指令会修改APSR中的标志。不过，32位Thumb-2指令可以修改这些标志，也可以不修改。为了区分这两种操作，根据统一汇编语言（UAL）语法，如需更新APSR标志位，应该使用S后缀。

```
ADD R0, R0, R1          ;R0=R0+R1
ADDS R0, R0, 0x12       ;R0=R0+0x12，更新APSR
ADC R0, R1, R2          ;R0=R1+R2+进位
```

Cortex-M3/M4乘法和MAC指令

- Cortex-M3和 Cortex-M4处理器都支持具有32位和64位结果的32位乘法指令和乘并累加（MAC）指令，APSR标志不受这些指令的影响。Cortex-M4处理器还支持额外的快速MAC指令。

指令（由于APSR不更新，因此无S后缀）		备注
MLA Rd, Rn, Rm, Ra	;Rd= Ra + Rn * Rm	32位结果
MLS Rd, Rn, Rm, Ra	;Rd= Ra - Rn * Rm	32位结果
SMULL RdLo, RdHi, Rn, Rm	;{ RdHi, RdLo }=Rn*Rm	64位结果
SMLAL RdLo, RdHi, Rn, Rm	;{ RdHi, RdLo }+=Rn*Rm	
UMULL RdLo, RdHi, Rn, Rm	; { RdHi, RdLo } =Rn*Rm	64位结果
UMLAL RdLo,RdHi,Rn,Rm	;{ RdHi, RdLo }+=Rn*Rm	

MLA: Multiply with Accumulate
MLS: Multiply and Subtract
SMULL: Signed Multiply (32 x 32), 64-bit result
UMULL: Unsigned Multiply (32 x 32), 64-bit result
UMLAL: Unsigned Multiply with Accumulate (32 x 32 + 64), 64-bit result

6.4.4 逻辑运算指令

指令(可选的S后缀未列出来)	操作
AND Rd, Rn ;Rd=Rd &Rn	按位与
AND Rd, Rn, # immed ;Rd=Rn &.#immed	
AND Rd, Rn, Rm ;Rd=Rn &Rm	
ORR Rd, Rn ;Rd=Rd|Rn	按位或
ORR Rd, Rn, #immed ;Rd= Rn|# immed	
ORR Rd, Rn, Rm ;Rd=Rn|Rm	
BIC Rd, Rn ;Rd=Rd &(~Rn)	位清除
BIC Rd, Rn,#immed ;Rd=Rn &(~#immed)	
BIC Rd, Rn, Rm ;Rd=Rn &(~Rm)	
ORN Rd, Rn, #immed ;Rd=Rn|(w#immed)	按位或非
ORN Rd, Rn, Rm ;Rd=Rn|(wRm)	
EOR Rd, Rn ;Rd=Rd^Rn	按位异或
EOR Rd, Rn, #immed ;Rd=Rn|#immed	
EOR Rd, Rn, Rm ;Rd=Rn|Rm	

ORN, ORNS: Logical OR NOT

ORR, ORRS: Logical OR

BIC, BICS: Bit Clear

EOR, EORS: Exclusive OR

移位运算示例

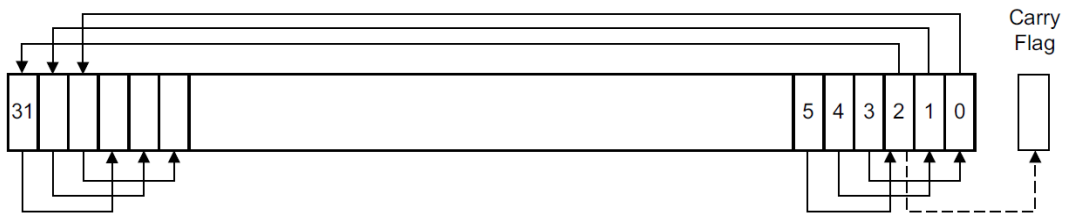
接下来将讨论移位和循环移位指令
回顾，6.3 T32指令集寻址方式

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

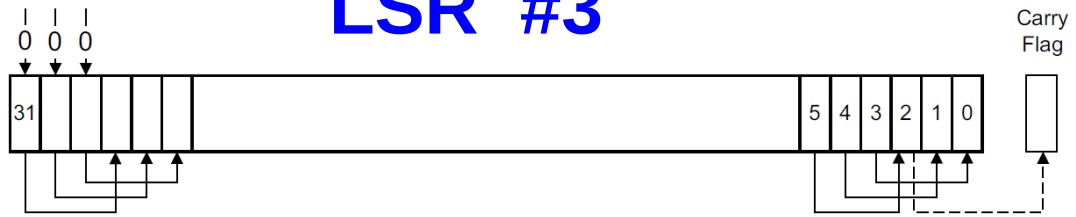
LSL #3

逻辑左移=算数左移

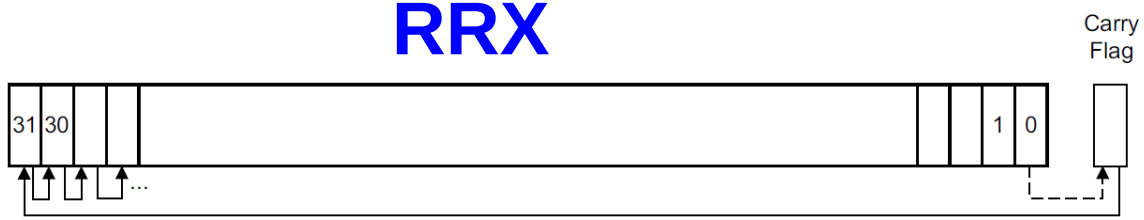
ROR #3



LSR #3



RRX



ASR #3

6.4.5 移位和循环移位指令

ASR, ASRS: Arithmetic Shift Right
LSL, LSLS: Logical Shift Left
ROR, RORS: Rotate Right
RRX, RRXS: Rotate Right with Extend

指令(可选的S后缀未列出)	操作
ASR Rd, Rn, #immed ;Rd=Rn>>immed	算术右移
ASR Rd, Rn ;Rd=Rd>>Rn	
ASR Rd, Rn, Rm ;Rd=Rn>>Rm	
LSL Rd, Rn, #immed ;Rd=Rn<<#immed	逻辑左移
LSL Rd, Rn ;Rd=Rd<<Rn	
LSL Rd, Rn, Rm ;Rd=Rn<<Rm	
LSR Rd, Rn, #immed ;Rd=Rn>>#immed	逻辑右移
LSR Rd, Rn ;Rd=Rd>>Rn	
LSR Rd, Rn, R ;Rd=Rn>>R	
ROR Rd, Rn ;Rd右移Rn	循环右移
ROR Rd, Rn, Rm ;Rd=Rn右移Rm	
RRX Rd, Rn ;{C, Rd} = {Rn, C}	循环右移并展开

6.4.6 数据格式转换

- 符号扩展（Sign eXtend）和无符号扩展（Zero extend）

指令	操作
SXTB Rd, Rn ;Rd=有符号展开(Rn[7:0])	有符号展开字节为字
SXTH Rd, Rn ;Rd=有符号展开(Rn[15:0])	有符号展开半字为字
UXTB Rd, Rn ;Rd=无符号展开(Rn[7:0])	无符号展开字节为字
UXTH Rd, Rn ;Rd=无符号展开(Rn[15:0])	无符号展开半字为字

*还可以选择在进行有符号展开运算前将输入数据循环右移，参见讲义

- SXTB/SXTH使用Rn的bit7/bit15进行符号扩展，而UXTB和UXTH则做零扩展。若R0中数值为0x55AA8765，则

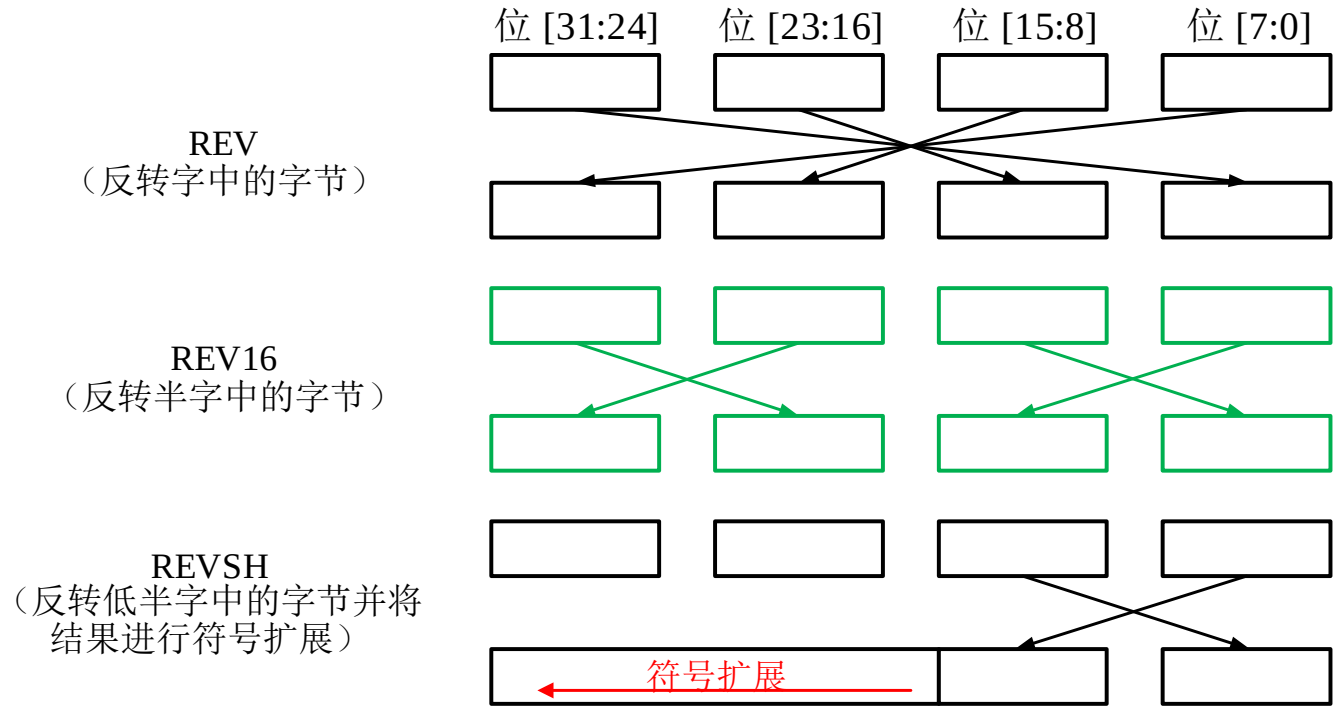
```
SXTB Rl, R0      ;R1=0x00000065
SXTH R1,R0       ;R1=0XFFFF8765
UXTB Rl, R0      ;R1=0x00000065
UXTH R1, R0      ;R1=0x00008765
```

0x65 = 0110 0101

0x8765 = 1000 0111 0110 0101

数据反转指令

指令	操作
REV Rd, Rn ;Rd= rev(Rn)	反转字中的字节
REV16 Rd, Rn ;RD=rev16(Rn)	反转每个半字中的字节
REVSH Rd, Rn ;Rd=revsh(Rn)	反转低半字中的字节并将结果有符号展开




```
; 若R0为0x12345678
REV R1, R0
; R1变为0x78563412
REVH R2, R0
; R2则会变为0x34127856
REVSH R3, R0
; R3则会变为0x00007856
```

6.4.7 位域(Bit Field)处理指令

指令	操作
BFC Rd, #<lsb>,#<width>	清除寄存器中的位域
BFI Rd, Rn, #<lsb>, #<width>	将位域插入寄存器
CLZ Rd, Rm	前导零计数
RBIT Rd, Rn	反转寄存器中的位顺序
SBFX Rd, Rn, #<lsb>, #<width>	从源中复制位域并有符号展开
UBFX Rd, Rn, #<lsb>,#<width>	从源寄存器中复制位域

BFC: Bit Field Clear
BFI: Bit Field Insert
CLZ: Count Leading Zeros
RBIT: Reverse Bits
SBFX: Signed Bit Field Extract
UBFX: Unsigned Bit Field Extract

```
LDR R0, =0x1234FFFF
;把0x1234FFFF装载到R0，注意不是从地址0x1234FFFF装载
BFC R0, #4, #8                ;R0=0x1234F00F

LDR R0, =0x12345678           ;把0x12345678装载到R0
LDR R1, =0x3355AACC           ;把0x3355AACC装载到R1
BFI R1, R0, #8, #16
;将R0[15:0]插入R1[23:8]，将得到R1=0x335678CC
```

6.4.8 比较和测试指令

指令	操作
CMP<Rn>,<Rm>	比较：计算Rn-Rm
CMP<Rn>,# <immed>	比较：计算Rn-立即数
CMN<Rn>, <Rm>	负比较：计算Rn+Rm
CMN<Rn>,#<immed>	负比较：计算Rn+立即数
TST<Rn>,<Rm>	测试(按位与)：计算Rn和Rm相与后的结果，APSR中的N位和Z位更新，若使用了桶形移位则更新C位
TST<Rn>, #<immed>	测试(按位与)：计算Rn和立即数相与后的结果，APSR中的N位和Z位更新
TEQ<Rn>, <Rm>	测试(按位异或)：计算Rn和Rm异或后的结果，APSR中的N位和Z位更新，若使用了桶形移位则更新C位
TEQ<Rn>, #<immed>	测试(按位异或)：计算Rn和立即数异或后的结果，APSR中的N位和z位更新

执行这些指令，APSR更新但运算结果不会保存

6.4.9 程序流控制指令

- 用于程序流控制的指令（也称为分支控制指令）有多种，如跳转、函数调用、条件跳转、比较和条件跳转的组合、条件执行（IF-THEN指令）、表格跳转等。在ARM处理器中，更新R15（PC）的数据处理指令（如MOV、ADD），或写入PC的读存储器指令（如LDR、LDM、POP）也会引起跳转操作。

助记符	英文名称	中文名称
B	Branch	无条件跳转
BL	Branch with Link	无条件跳转并保存返回地址，用于函数调用
BLX	Branch and Link with eXchange	寄存器间接寻址跳转并保存返回地址，用于函数调用
BX	Branch and eXchange	寄存器间接寻址跳转
CBNZ	Compare and Branch if Non Zero	比较不为零则跳转
CBZ	Compare and Branch if Zero	比较为零则跳转
IT	If-Then	条件执行指令
TBB	Table Branch Byte	按照跳转表跳转（字节）
TBH	Table Branch Halfword	按照跳转表跳转（半字）

无条件跳转

- 无条件跳转指令可以跳转到标号（<label>）所在的语句，或跳转到<Rm>中的存放地址位置。不同跳转指令所支持的最大跳转范围不一样

跳转指令	跳转的范围	功能
B <label>	−1MB to +1MB	跳转到标号地址
BL <label>	−16 MB to +16 MB	跳转到标号地址并将返回地址保存在LR(R14)中
BX <Rm>	寄存器中的任何值	跳转到Rm指定的地址
BLX <Rm>	寄存器中的任何值	跳转到Rm指定的地址，并将返回地址保存在LR(R14)中

- BL(Branch with Link)以及BLX(Branch and Link with eXchange)跳转并保存返回地址到LR(R14)，故可用于函数调用

条件跳转

- 条件跳转指的是，根据APSR寄存器中的标志位（N、Z、C和V标志位）决定是否跳转。无条件跳转指令添加条件码后缀即条件跳转指令。

跳转指令	跳转的范围	功能
B [cond] <label> (在IT block外)	−1 MB to +1 MB	跳转到标号地址 稍后会解释IT block
B [cond] <label> (在IT block内)	−16 MB to +16 MB	跳转到标号地址
BL[cond] <label>	−16 MB to +16 MB	跳转到标号地址并将返回地址保存在LR(R14)中
BX[cond] <Rm>	寄存器中的任何值	跳转到Rm指定的地址
BLX[cond] <Rm>	寄存器中的任何值	跳转到Rm指定的地址，并将返回地址保存在LR(R14)中

- 注意，B指令的二进制编码格式中预留了四比特的条件码，故与一般的条件执行指令（如MOVEQ）不同，不需要与IT指令配合，条件跳转指令也可以执行。

比较和跳转

因讲义撰写和修订仍在进行中，已授课内容中出现的不妥描述在更新版PPT中会进行标识。感谢同学们对新课程建设的理解和支持！

- ARMv7-M架构提供了两个新的指令，合并了和零比较以及条件跳转操作：CBZ（比较为零则跳转）和CBNZ（比较非零则跳转）。

~~CBZ R0, #1~~

~~;比较R0和1，如果相等则跳转到R1中存放数值对应的地址~~

~~CNBZ R5, #9~~

~~;比较R5和9，如果相等则跳转到R5中存放数值对应的地址~~

CBZ R0, label

;比较R0和0，如果相等则跳转到label对应的地址

CNBZ R5, label

;比较R5和0，如果不相等则跳转到lable对应的地址

等价于

CMP Rn, #0
BEQ label

等价于

CMP Rn, #0
BNE label

- CBZ指令的作用相当于CMP指令和BEQ指令的功能组合，区别在于APSR的值不受CBZ和CBNZ指令的影响。另外，CBZ和CBNZ指令只支持前向跳转，不支持向后跳转，跳转范围为当前指令后的4~130字节。且，只能使用R0~R7。
- 鉴于这样的特性，往往CBZ和CBNZ指令被适用于较小的循环体内的分支控制。

条件执行指令

- 条件跳转指令是根据APSR标志位决定是否跳转，即在B指令后添加条件码后缀（如EQ、NE、…）。
- 除此之外，ARM处理器还支持条件执行，对应的指令为IT，“IT”意为“If Then”。
- IT指令允许跟随其后的最多四条指令是条件执行的，IT指令后的几条指令被称作一个IT块（IT block）。
- “IT”指令的汇编语法为：IT{x{y{z}}} cond

- 汇编工具软件会在带有条件码后缀的语句前自动插入所需的IT指令，故程序员无须自己在代码中使用IT指令。

IT 指令的汇编语法

回顾之前章节

语法格式： **IT{x{y{z}}} cond**

- cond 为IT块中第一条指令使用的条件码；
- x 指定IT块中第二条指令的是否执行的开关；
- y 指定IT块中第三条指令的是否执行的开关；
- z 指定IT块中第四条指令的是否执行的开关。

几种加了后缀的IT指令（还有其他形式如ITEEE）

IT，“If-Then”，下一条指令是条件执行的；

ITT，“If-Then-Then”，下两条指令是条件执行的；

ITE，“If-Then-Else”，下两条指令是条件执行的；

ITTE，“If-Then-Then-Else”，下三条指令条件执行；

ITTEE，“If-Then-Then-Else-Else”，下四条指令条件执行。

条件执行指令示例

- **IT EQ**
  - ;意为“If-Then”，下一条指令是条件执行的
- **ADDEQ R0, R1, R2**
  - ;如果APSR的Z标志位是1则将R1+R2后结果传送至R0
- **ITETT NE**
  - ;意为“If-Then-Else-Then-Then”，随后四条指令是条件执行的
- **ADDNE R0, R0, R1**
  - ;如果APSR的Z标志位是0则将R0+R1后结果传送至R0
- **ADDEQ R0, R0, R3**
  - ;如果APSR的Z标志位是1则将R0+R3后结果传送至R0
- **ADDNE R2, R4, #1**
  - ;如果APSR的Z标志位是0则将R0+R1后结果传送至R0
- **MOVNE R5, R3**
  - ;如果APSR的Z标志位是0则将R3中的数传送至R5

按跳转表跳转
为什么要用一个表来定义跳转方式？

- 在汇编编程时直接实现类似C语言中switch(case)这样的多分支跳转控制十分困难，因为需要跳转的目标地址与一个变量有关。
- 无条件跳转或条件跳转，跳转的目标地址是一个绝对物理地址或者一个PC相对地址，而switch(case)中变量取不同值的时候需要跳转的偏移量是不同的。
- 如果预先定义一个数组，每个数组元素是一个目标地址，那么通过改变数组下标，我们就可以通过数组元素得到不同的目标地址。跳转表就是一个这样的数组，跳转表的表项就是数组元素，通过跳转表首地址和表内的偏移可以实现类似数组不同下标的访问方式。

按跳转表跳转

- **ARM支持两个按跳转表跳转（Table Branch）指令：**
  - TBB（按照字节为单位的跳转表跳转）
  - TBH（按照半字为单位的跳转表跳转）
- **跳转表**
  - 可以被组织成字节数组（相对于基地址的偏移小于2×2^8=512B）
  - 或被组织成半字数组（相对于基地址的偏移小于2×2^{16}=128KB）

- **TBB指令的语法为：“TBB [Rn, Rm]”**
- **TBH的语法稍微不同：“TBH [Rn, Rm, LSL #1]”**
  - Rn中存放跳转表的基地址，不能是SP；
  - Rm则为跳转表偏移，不能是SP或PC。
  - TBB指令执行的时候，跳转的目标地址是[Rn+Rm]）
  - TBH指令执行的时候，跳转的目标地址是[Rn+Rm<<1]

按跳转表跳转示例


```
    ADR.W R0, BranchTable_Byte
    TBB [R0, R1] ; R1 is the index, R0 is the base address of the branch table
Case1
  ; 其他指令
Case2
  ; 其他指令
Case3
  ; 其他指令
BranchTable_Byte
    DCB 0 ; Case1 offset calculation
    DCB ((Case2-Case1)/2) ; Case2 offset calculation
    DCB ((Case3-Case1)/2) ; Case3 offset calculation
```

跳转表基地址对应标号（Label）为“BranchTable_Byte”的语句。每个表项占据一个字节，表项的数值则是与标号为“Case1”语句的偏移量。程序执行到TBB所在行的时候，根据R1取值的不同，会分别跳转到不同的分支。

DCB是ARM汇编语言中的伪指令，用于分配一片连续的字节存储单元并用指定的数据初始化

小结：Thumb指令集中的跳转指令

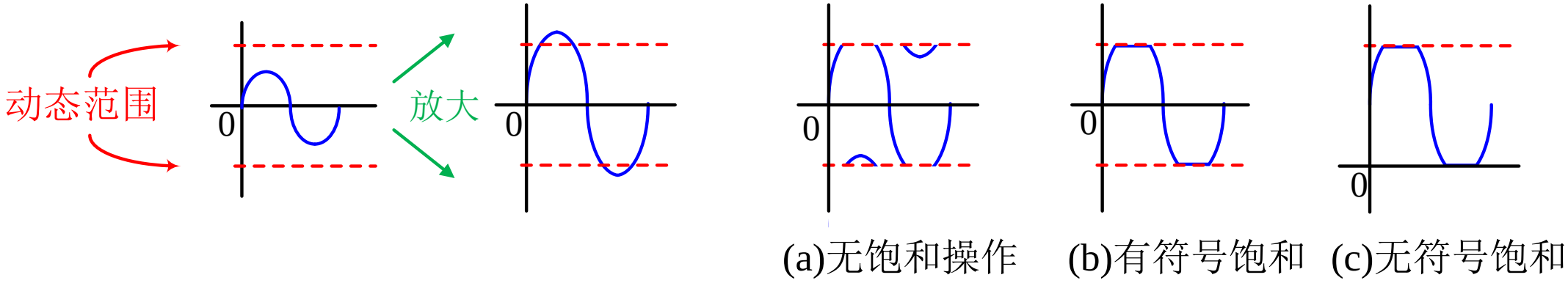
Instruction	Usage	Range
B on page A7-205	Branch to target address	+/–1 MB
CBNZ, CBZ on page A7-216	Compare and Branch on Nonzero, Compare and Branch on Zero	0-126 B
BL on page A7-213	Call a subroutine	+/–16 MB
BLX (register) on page A7-214	Call a subroutine, optionally change instruction set	Any
BX on page A7-215	Branch to target address, change instruction set	Any
TBB, TBH on page A7-416	TBB: Table Branch, byte offsets	0-510 B
	TBH: Table Branch, halfword offsets	0-131070 B

ARM®v7-M Architecture Reference Manual Reference Manual. Available:
http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0204j/Cihcdbca.html

6.4.10 饱和运算

- 定点运算中经常遇到运算结果溢出的情况，有上溢和下溢两种。
- 溢出（Overflow）运算：传统运算方式（即“环绕式”运算）的处理通常是自动回到计数零点。
- 饱和（Saturation）运算：当达到最大值时不再增加，而是保持在这个最大值上。只有加、减指令才有饱和方式。
  - 例如：在图像和图形处理中，用8位无符号数表示一个象素值或颜色分量值，最大数255表示亮度的最大值，如果运算结果超出这个范围，仍然表示为“最亮”是符合实际情况的。

有符号数和无符号数的饱和



(a)无饱和操作　(b)有符号饱和　(c)无符号饱和

例，一个32位有符号数值要被饱和为16位有符号数

SSAT R1, #16, R0

;if R0=0x00020000 then R1=0x00007FFF，此时做了符号扩展　16bit 有符号数上界 0x7FFF

;if R0=0xFFFF8000 then R1=0xFFFF8000，此时做了符号扩展　16bit 有符号数下界 0xFFFF

例，一个32位有符号数转换为16位无符号数:

USAT R1, #16, R0

;if R0=0x00020000 then R1=0x0000FFFF　16bit 无符号数上界 0xFFFF

;if R0=0xFFFF8000 then R1=0x00000000　16bit 无符号数下界 0x0000

6.4.11 其他杂类指令

定性了解

助记符	英文名称	中文名称
BKPT	Breakpoint	断点
CPSID	Change Processor State, Disable Interrupts	改变处理器状态并禁止中断
CPSIE	Change Processor State, Enable Interrupts	改变处理器状态并使能中断
DMB	Data Memory Barrier	数据内存屏障
DSB	Data Synchronization Barrier	数据同步屏障
ISB	Instruction Synchronization Barrier	指令同步隔离
NOP	No Operation	空指令
SEV	Send Event	发送事件，用于多处理器系统
SVC	Supervisor Call	产生SVC异常
WFE	Wait For Event	有条件地进入休眠状态
WFI	Wait For Interrupt	等待中断（进入休眠状态）

存储器屏障指令

- 在具有超矢量或乱序执行能力的处理器中，当处理器检测到当前访问存储器指令需要等待，而下一条指令并不依赖于当前指令执行结果时，可以不必等待当前指令执行完毕而直接执行程序中的下一条指令。这种对存储器访问的重新排序，在多处理器共享数据的情形，可能会导致另一个处理器看到的数据顺序可能和设定的不同，从而可能会引起错误。
- 数据内存屏障：所有在DMB指令之前的内存访问完成后，才可以执行DMB之后的内存访问相关的指令。
- 数据同步屏障：所有在DSB指令之前的指令完成后才可以执行DSB指令后的指令。
- 指令同步隔离：清空流水线，ISB指令之后执行的指令都是从内存或缓存中获得的。

休眠模式指令

- **WFI**指令会使处理器立即进入休眠模式，中断、复位或调试操作可以将处理器从休眠中唤醒。
- **WFE**会使处理器有条件地进入休眠。在Cortex-M3/M4处理器内部，有一个只有一位的寄存器会记录事件。若该寄存器置位，WFE指令不会进入休眠而只是清除事件寄存器并继续执行下一条指令；若该寄存器清零，则处理器会进入休眠而且会被事件唤醒，事件可以是中断、调试操作、复位或外部事件输入的脉冲信号。
- **SEV**指令可用于发送事件，多处理器系统中可用于向其他处理器传递信号。Cortex-M处理器的接口信号包括一个事件输入和一个事件输出。处理器的事件输入可由多处理器系统中其他处理器的事件输出产生，因此，处于WFE休眠的处理器可由其他的处理器唤醒。

异常相关指令

- 管理调用（SVC）指令用于产生SVC异常（异常类型为11）。SVC一般用于嵌入式操作系统（OS），其中，运行在非特权执行状态的应用可以请求运行在特权状态的OS的服务。SVC异常机制提供了从非特权到特权的转换。SVC机制也可以作为应用任务访问各种服务（包括OS服务或其他API函数）的入口，这样应用任务就可以在无须了解服务的实际存储器地址的情况下请求所需服务，只需知道SVC服务编号、输入参数和返回结果。
- 另一个和异常相关的指令是CPS指令，用于改变处理器状态。使用这条指令可以设置或清除 PRIMASK和 FAULTMASK等中断屏蔽寄存器。

空指令和断点指令

- Cortex-M处理器支持NOP指令，可用于产生指令对齐或延时。有一点需要注意，NOP指令产生的延时在不同系统间可能会存在差异。若需要精确的延时，应该使用硬件定时器。
- 在软件开发/调试过程中，断点（ BKPT: Breakpoint ）指令用于实现应用程序中的软件断点。若程序在SRAM中执行，则该指令一般由调试器插入以替换原有的指令。当到达断点时，处理器会被暂停，然后调试器就会恢复原有的指令，用户也可以通过调试器执行调试任务。BKPT指令也可以用于产生调试监控异常，它具有一个8位立即数，调试器或调试监控异常可以将该数据提取出来，并根据该信息确定要执行的动作。

6.4.12 Cortex-M4特有指令

- 与Cortex-M3处理器相比，Cortex-M4支持的指令更多。新增的功能包括：单指令多数据（SIMD）、饱和指令、其他的乘法和MAC（乘累加）指令、打包和解包指令、可选的浮点指令等。这些指令可以让Cortex-M4可以更加高效地进行实时数字信号处理。

仅了解一下数字信号处理算法对指令系统的要求

- 往往对数组的不同元素做相同的运算
- 循环体内往往是两个操作数相乘之后进行累加
- 往往操作数动态范围很大，用的是浮点数

$$y[n] = \sum_{k=0}^{N-1} h[k] \cdot x[n-k]$$

总结：ARM指令系统

- 指令集体系结构：A64、A32和T32
- ARM汇编指令要素
  - <opcode>*[cond][q][S]*_<Rd>,_<Rn>*[,_Oprand2]*
- 寻址：根据指令中的地址信息来寻找操作数有效地址
  - 立即数、基址寄存器、地址偏移量
- Cortex-M指令
  - 通用：各种运算、分支跳转、数据格式转换…
  - 特色：存储器访问指令丰富（寻址）、位域处理…

中国科学技术大学
University of Science and Technology of China
信息科学技术学院

计算机原理与嵌入式系统

第6章 ARM处理器的指令系统