

# 《嵌入式系统》

## （第二讲）

厦门大学信息学院软件工程系 曾文华

2023年9月12日

- 第1章：嵌入式系统概述
- 第2章：ARM处理器和指令集
- 第3章：嵌入式Linux操作系统
- 第4章：嵌入式软件编程技术
- 第5章：开发环境和调试技术
- 第6章：Boot Loader技术
- 第7章：ARM Linux内核
- 第8章：Android操作系统（增加）
- 第9章：文件系统
- 第10章：设备驱动程序设计基础
- 第11章：字符设备和驱动程序设计
- 第12章：块设备和驱动程序设计
- 第13章：网络设备驱动程序开发
- 第14章：华为昇腾AI处理器及应用（增加）
- 第15章：嵌入式GUI及应用程序设计（不讲）



# 第2章 ARM处理器和指令集

- 2.1 ARM处理器简介
- 2.2 ARM指令集简介
- 2.3 ARM指令的寻址方式
- 2.4 ARM指令简介
- 2.5 Thumb指令简介

# 2.1 ARM处理器简介

## • 2.1.1 ARM公司和ARM产品简介

软件 VHDL (硬件描述语言)

- **ARM公司**：本身不生产芯片，而是通过转让设计方案（**IP核**）由合作伙伴（**Samsung**、**Intel**、飞思卡尔（**Freescale**）、意法半导体（**ST**）等公司）生产各具特色的芯片

**Intellectual Property: 知识产权**

- **ARM产品**：**ARM系列处理器**

- ARM7 系列
- ARM9 系列
- ARM9E 系列
- ARM10E 系列
- ARM11 系列
- SecurCore 系列
- Intel XScale 系列
- Intel StrongARM 系列
- Cortex 系列

最新的  
Cortex 系列

- » Cortex-A: Cortex-A9, **Freescale i.MX6**
- » Cortex-R
- » Cortex-M: Cortex-M3, **STM32F103**

## • 2.1.2 ARM指令集体系结构版本

- ARM 指令集体系结构的版本：
  - v1版本: ARM1
  - v2版本: ARM2
    - » v2a版本: ARM3
  - v3版本: ARM6
  - v4版本: ARM7、ARM9、StrongARM
    - » v4T版本
  - v5版本: ARM9E、ARM10、Xscale
    - » v5T版本
    - » v5TE版本
  - v6版本: ARM11
  - v7版本: ARM Cortex (A系列, R系列, M系列)
  - v8版本: 支持64位指令集
  - v9版本: AI、安全和机密计算

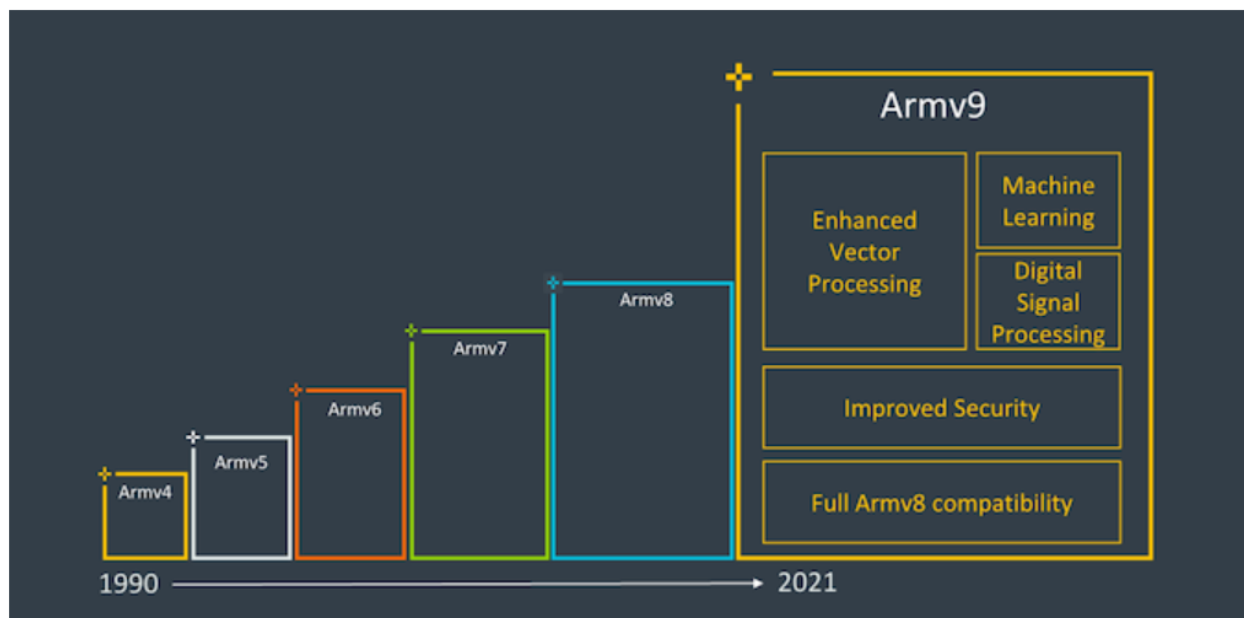
# 剑指3000亿颗芯片！时隔十年，Arm发布全新Armv9架构

kk\_wu • 21-03-31

**2021年3月31日**

今天，Arm 公司正式宣布推出全新的 Armv9 架构。这是自 2011 年 10 月推出 Armv8 架构近十年之后，该公司对其 CPU 架构的首次重大调整变革。该公司表示，本次 v9 架构旨在为移动端设备、计算机和服务器提供更强的算法支持。

Arm 公司表示，未来 Armv9 架构将装备在 3000 亿颗 Arm 芯片中。并且，按照现有发展速度，预计未来五年内 Arm 设备的出货量将超过 1000 亿台，100% 的共享数据将在 Arm 芯片上处理，无论是在终端还是在云端。



**Armv9的重大升级：AI、安全和机密计算**

# Arm发布新一代Armv9 CPU，及首款移动端支持光追的Immortalis GPU

🔊 播报文章



热点科技

2022-06-29 17:44 上海 | 电商达人, ITheat热点科技官方帐号, 优质数码领域创作者

2022年6月29日

关注

今天，Arm召开了2022年全面计算解决方案发布会，推出了基于Armv9架构新一代的CPU，包括Arm Cortex-X3、Arm Cortex-A715等，同时还带来了全新旗舰Immortalis GPU，而这也是首款可在移动端支持硬件光线追踪的GPU，致力于带来更为真实沉浸的游戏体验。



更富有沉浸感的游戏体验

百家号/热点科技

# 彻底换代！Arm发布全新Armv9芯片：正式放弃32位

2022-07-20 21:50

2022年7月20日

6月29日，英国芯片设计公司Arm正式发布了基于Armv9架构设计的全新一代公版处理器，分别为大核心Cortex-X3、中核心Cortex-A715和小核心Cortex-A510 Refresh。在此次的更新中，中核心Cortex-A715不再支持32位应用。





## • 2.1.3 ARM处理器系列

- 包括：ARM系列处理器、授权厂家基于ARM体系结构设计的处理器
  - ARM7系列
  - ARM9系列
  - ARM9E系列
  - ARM10E系列
  - ARM11系列
  - SecurCore系列：专为安全需要设计的
  - StrongARM系列：Intel公司基于ARM体系结构设计的处理器
  - XScale系列：Intel公司基于ARM体系结构设计的处理器
  - Cortex系列（Cortex-A、Cortex-R、Cortex-M）：最新的处理器

最新的  
Cortex系列

- **ARM7系列**

- ARM7TDMI: 一款使用比较广泛的ARM处理器核，没有MMU

典型产品

» **S3C44B0X** (Samsung公司生产)

- **ARM9系列**

- ARM920T

典型产品

» **S3C2410** (Samsung公司生产)

- **ARM9E系列**

- **ARM10E系列**

- **ARM11系列**

- **SecurCore系列**：专为安全需要而设计的
- **StrongARM系列**：基于ARM v4版本，由Intel公司开发，用于手持式消费类电子设备和移动计算与通信领域
- **XScale系列**：基于ARM v5TE，由Intel公司开发
  - **PXA系列**：用于手持和无线设备
    - » **PXA255**
    - » **PXA270**

- **Cortex系列**：ARM公司最新的处理器
  - **Cortex-A系列**：Highest Performance（高性能）
    - » Cortex-A53、Cortex-A57、Cortex-A77

典型产品

» Cortex-A9, **Freescall i.MX6**

- **Cortex-R系列**：Real-Time Processing（实时处理）
  - » Cortex-R52

- **Cortex-M系列**：Lowest Power, Lower Cost（低功耗、低价格）
  - » Cortex-M35P

典型产品

» Cortex-M3, **STM32F103**

## 2.2 ARM指令集简介

- 2.2.1 RISC简介

- **RISC**: Reduced Instruction Set Computer, 精简指令集计算机
- **CISC**: Complex Instruction Set Computer, 复杂指令集计算机
- 采用**RISC结构**的处理器:
  - **ARM公司**: **ARM系列处理器**
  - **HP公司**: 例如HP PA-RISC
  - **IBM公司**: 例如IBM Power-PC
  - **SUN公司**: 例如SUN SPARC

## • 2.2.2 ARM状态和Thumb状态

指令字长为32位

- **ARM指令集**（32位），**Thumb指令集**（16位）

指令字长为16位

- Thumb指令集是ARM指令集的子集

- **ARM状态**（32位），**Thumb状态**（16位）

- ARM指令必须在ARM状态下执行，Thumb指令必须在Thumb状态下执行

- ARM处理器可以在两种状态（ARM状态、Thumb状态）下进行**切换**；只要遵循ATPCS调用规则，ARM子程序（ARM指令集编写的程序）和Thumb子程序（Thumb指令集编写的程序）之间可以进行相互调用

- ATPCS调用规则：ARM-Thumb Procedure Call Standard（ARM-Thumb过程调用标准）

- ARM状态和Thumb状态切换可以通过BX指令来实现

# ARM处理器的寄存器

- ARM处理器的寄存器（37个32位的寄存器）：

- 31个通用寄存器

- R0、R1、R2、R3、R4、R5、R6、R7 8个
    - R8、R8\_fiq 2个
    - R9、R9\_fiq 2个
    - R10、R10\_fiq 2个
    - R11、R11\_fiq 2个
    - R12、R12\_fiq 2个
    - R13、R13\_svc、R13\_abt、R13\_und、R13\_irq、R13\_fiq 6个
    - R14、R14\_svc、R14\_abt、R14\_und、R14\_irq、R14\_fiq 6个
    - R15（PC） 1个
  - R13: SP, 堆栈指针, Stack Pointer
    - R14: LR, 链接寄存器, Link Register
    - R15: PC, 程序计数器, Program Counter

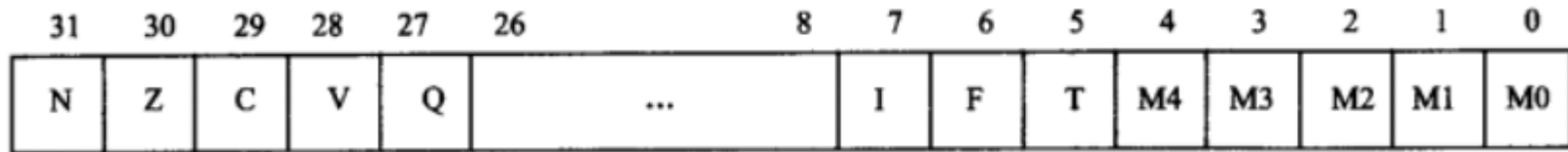
- 6个专用状态寄存器

- CPSR 1个
    - SPSR\_svc、SPSR\_abt、SPSR\_und、SPSR\_irq、SPSR\_fiq 5个
  - CPSR: Current Program Status Register, 当前程序状态寄存器
    - SPSR: Saved Program Status Register, 程序状态保存寄存器

# ARM处理器的通用寄存器（31个）

System & User	FIQ	Supervisor	About	IRG	Undefined
R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7
R8	R8_fiq	R8	R8	R8	R8
R9	R9_fiq	R9	R9	R9	R9
R10	R10_fiq	R10	R10	R10	R10
R11	R11_fiq	R11	R11	R11	R11
R12	R12_fiq	R12	R12	R12	R12
R13	R13_fiq	R13_svc	R13_abt	R13_irq	R13_und
R14	R14_fiq	R14_svc	R14_abt	R14_irq	R14_und
R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)
16个	7个	2个	2个	2个	2个

# CPSR、SPSR



- **N: 负数标志位**。如果目标寄存器中的有符号数为负数，则N=1，否则N=0。
- **Z: 零标志位**。如果目标寄存器中的数为0，则N=1，否则N=1。
- **C: 进位标志位**。有以下3种情况
  - 1、无符号加法运算和CMN指令，如果产生进位，则C=1，否则C=0；
  - 2、无符号减法运算和CMP指令，如果产生借位，则C=0，否则C=1；
  - 3、进行移位操作的时候，C中保存最后一位移出的值。
- **V: 溢出标志位**。进行有符号运算时如果发生错误，则V=1，否则V=0。

PSW 状态寄存器

**CPSR: Current Program Status Register**，当前程序状态寄存器  
**SPSR: Saved Program Status Register**，程序状态保存寄存器

- 说明：当一条指令中同时含有算术运算指令和移位指令时，影响C的值是算术运算而不是移位操作。
- 一些指令如CMN、TEQ等会无条件的刷新CPSR中的条件标志位，其他指令必须要在指令后面加上S后缀才会改变CPSR中的条件标志位。
- **Q: 标识位**。主要用于指示增强的DSP指令是否发生了溢出。SPSR的Q标识位，用于在异常中断发生时保存和恢复CPSR中的Q标识位。
- **I: IRQ中断禁止位**。I=1代表禁止IRQ中断，I=0代表允许IRQ中断。
- **F: FIQ中断禁止位**。F=1代表禁止FIQ中断，F=0代表允许FIQ中断。这里和51单片机中的中断使能位有点小差别，51中的是中断使能位，所以为1的时候应该是中断使能，即允许中断。而这里是中断禁止位，为1的时候应该是禁止中断。
- **T: 这一位只在ARMv4T指令集版本及以上才有效。因为ARMv4版本及以下都不支持Thumb指令集。在支持Thumb指令集的处理器中，T=0表示处于ARM状态，T=1表示处于Thumb状态。**
- **M[4:0]: 用于控制7种模式位。**

M[4:0]	处理器模式
0b10000	User
0b10001	FIQ
0b10010	IRQ
0b10011	Supervisor
0b10111	Abort
0b11011	Undefined
0b11111	System



# ARM处理器的运行模式

## — ARM的处理器运行模式（7个）：

1. 系统模式（SYS, System）
2. 用户模式（USR, User）
3. 快速中断模式（FIQ）
4. 管理模式（SVC, Supervisor）
5. 数据访问终止模式（ABT, Abort）
6. 外部中断模式（IRQ）
7. 未定义指令终止模式（UND, Undefined, 未定义模式）

M[4:0]	处理器模式
0b10000	User
0b10001	FIQ
0b10010	IRQ
0b10011	Supervisor
0b10111	Abort
0b11011	Undefined
0b11111	System

## — 特权模式（除用户模式外，6个）：

1. 系统模式（SYS, System）
2. 快速中断模式（FIQ）
3. 管理模式（SVC, Supervisor）
4. 数据访问终止模式（ABT, Abort）
5. 外部中断模式（IRQ）
6. 未定义指令终止模式（UND, Undefined, 未定义模式）

## — 异常模式（除系统模式、用户模式外，5个）：

1. 快速中断模式（FIQ）
2. 管理模式（SVC, Supervisor）
3. 数据访问终止模式（ABT, Abort）
4. 外部中断模式（IRQ）
5. 未定义指令终止模式（UND, Undefined, 未定义模式）

## • 2.2.3 ARM指令类型和指令的条件域

### • 指令类型（8类）：

- ① 跳转指令
- ② 通用数据处理指令
- ③ 乘法指令
- ④ Load/Store内存访问指令
- ⑤ ARM协处理器指令
- ⑥ 杂项指令
- ⑦ 饱和算术指令
- ⑧ ARM伪指令

操作码

if

目的  
操作数

源  
操作数  
1

源  
操作数  
2

### • 指令格式：<opcode> {<cond>} {S} <Rd>, <Rn> {, <shift\_op2>}

- <>内的项是必须的，{}内的项是可选的
- **opcode**: 指令助记符（操作码），如LDR，STR等
- **cond**: 执行条件（条件域/条件码），如EQ，NE等，可以没有
- **S**: 可选后缀，加S时影响CPSR中的条件码标志位，不加S时则不影响
- **Rd**: 目标寄存器
- **Rn**: 第1个源操作数的寄存器
- **op2**: 第2个源操作数，可以没有
- **shift**: 位移操作，可以没有

- 条件码 (<cond>) : CPSR[31:28] N Z C V

- N (负数标志位) : 运算结果的b31位值。对于有符号二进制补码，结果为负数时N=1，结果为正数或零时N=0
- Z (零标志) : 指令结果为0时Z=1，否则Z=0
- C (进位标志) : 使用加法运算（包括CMN指令），b31位产生进位时C=1，否则C=0。使用减法运算（包括CMP），b31位产生借位时C=0，否则C=1。对于结合移位操作的非加法 / 减法指令，C为b31位最后的移出值，其它指令C通常不变
- V (溢出标志) : 使用加法 / 减法运算，当发生有符号溢出时V=1，否则V=0，其它指令V通常不变

## CPSR

31	30	29	28	27	26	8	7	6	5	4	3	2	1	0
N	Z	C	V	Q	...		I	F	T	M4	M3	M2	M1	M0

- 条件码（**NZCV**）的16种组合

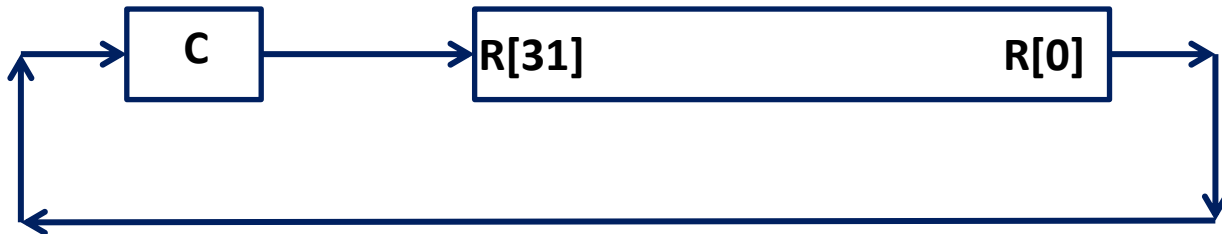
AMR指令条件码	助记符	描述	CPSR条件码标志位的值
0000	EQ	相等，运行结果为0	Z置位
0001	NE	不相等，运行结果不为0	Z清零
0010	CS/HS	无符号数大于等于	C置位
0011	CC/LO	无符号数小于	C清零
0100	MI	负数	N置位
0101	PL	非负数	N清零
0110	VS	上溢出	V置位
0111	VC	没有上溢出	V清零
1000	HI	无符号数大于	C置位且Z清零
1001	LS	无符号数小于等于	C清零且Z置位
1010	GE	带符号数大于等于	N=V
1011	LT	带符号数小于	N!=V
1100	GT	带符号数大于	Z清零且N=V
1101	LE	带符号数小于等于	Z置位且N!=V
1110	AL	无条件执行	
1111	系统保留		

- 位移操作<shift\_op2>的11种形式

语法	含义
#<immediate>	立即数寻址
<Rm>	寄存器寻址
<Rm>, LSL #<shift_imm>	立即数逻辑左移
<Rm>, LSL <Rs>	寄存器逻辑左移
<Rm>, LSR #<shift_imm>	立即数逻辑右移
<Rm>, LSR <Rs>	寄存器逻辑右移
<Rm>, ASR #<shift_imm>	立即数算术右移
<Rm>, ASR <Rs>	寄存器算术右移
<Rm>, ROR #<shift_imm>	立即数循环右移
<Rm>, ROR <Rs>	寄存器循环右移
<Rm>, RRX	寄存器扩展循环右移

- 位移操作的5种方式:

- LSL: 逻辑左移
- LSR: 逻辑右移
- ASR: 算术右移
- ROR: 循环右移
- RRX: 带扩展的循环右移 (加上CPSR的C位一起循环右移)



**RRX**

## 2.3 ARM指令的寻址方式

- 9种寻址方式:

- ① 立即寻址
- ② 寄存器寻址
- ③ 寄存器偏移寻址
- ④ 寄存器间接寻址
- ⑤ 基址变址寻址
- ⑥ 多寄存器寻址
- ⑦ 堆栈寻址
- ⑧ 相对寻址
- ⑨ 块复制寻址

表5.6 x86的主要寻址方式

x86

序号	寻址方式	有效地址EA/操作数S	指令示例
1	立即数寻址	$S=Disp$	MOV EAX,1000
2	直接寻址	$EA=Disp$	MOV EAX,[1080H]
3	寄存器寻址	$S=R[R/M]$	MOV EAX,ECX
4	寄存器间接寻址	$EA=R[R/M]$	MOV EAX,[EBX]
5	寄存器相对寻址 (基址寻址)	$EA=R[R/M]+Disp8$	MOV EAX,[ESI+100H]
6	基址+比例变址寻址	$EA=5*index+Base$	MOV EAX,[EBX+EDI*4]
7	基址+比例变址+偏移量寻址	$EA=5*index+Base+Disp$	MOV EAX,[EBX+EDI*4+66]
8	相对寻址	$EA=PC+Disp$	JMP 1000H

表5.11 MIPS指令的寻址方式

MIPS

序号	寻址方式	有效地址EA/操作数S	指令示例
1	立即数寻址	$S=imm$	addi rt,rs,imm
2	寄存器寻址	$S=R[rt]$	add rd,rs,rt
3	寄存器相对寻址 (基址寻址)	$EA=R[rs]+imm$	lw rt,imm(rs)
4	相对寻址	$EA=PC+4+Disp$	beq rs,rt,imm
5	伪直接寻址	$EA=((PC+4)_{31:26}+address,00)$	j address

表5.19 RISC-V指令的寻址方式

RISC-V

序号	寻址方式	有效地址EA/操作数S	指令示例
1	立即数寻址	$S=imm$	addi rd,rs1,imm
2	寄存器寻址	$S=R[rs1]$	add rd,rs1,rs2
3	寄存器相对寻址 (基址寻址)	$EA=R[rs1]+imm$	lw rd,imm(rs1)
4	相对寻址	$EA=PC+imm<<1$	beq rs1,rs2,imm

- 指令格式:  $\langle opcode \rangle \{ \langle cond \rangle \} \{ S \} \langle Rd \rangle, \langle Rn \rangle \{, \langle shift\_op2 \rangle \}$

- $\langle \rangle$ 内的项是必须的,  $\{ \}$ 内的项是可选的
- opcode: 指令助记符 (操作码), 如LDR, STR等
- cond: 执行条件 (条件码), 如EQ, NE等
- S: 可选后缀, 加S时影响CPSR中的条件码标志位, 不加S时则不影响
- Rd: 目标寄存器
- Rn: 第1个源操作数的寄存器
- op2: 第2个源操作数
- shift: 位移操作

## • 2.3.1 立即寻址

- `<opcode> {<cond>} {S} <Rd>, <Rn> {, <shift_op2>}`

• `ADD R1, R1, #0x1 ; R1+1 -> R1`

3个操作数

- **#**: 立即寻址的前缀

**#0x1**: 表示是一个十六进制的立即数“1”

- **#0x**: 十六进制
- **#0b**: 二进制
- **#0d**: 十进制

- ARM指令通常有3个操作数: 1个目的操作数, 2个源操作数



## • 2.3.2 寄存器寻址

- `<opcode> {<cond>} {S} <Rd>, <Rn> {, <shift_op2>}`

- `ADD R1, R1, R2` ; R1+R2 -> R1

3个操作数

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

2个操作数

## • 2.3.3 寄存器偏移寻址（寄存器移位寻址）

- <opcode> {<cond>} {S} <Rd>, <Rn> {, <shift\_op2>}

• ADD R1, R1, R2, ROR #0x2 ; R2循环右移2位后与 R1相加，结果放入R1 3个操作数

• MOV R1, R0, LSL R2 ; R0逻辑左移R2位后 -> R1 2个操作数

## • 2.3.4 寄存器间接寻址

- <opcode> {<cond>} {S} <Rd>, <Rn> {, <shift\_op2>}
- STR R1, [R2] ; R1 -> [R2],  
; 将R1存入[R2]存储单元中
- SWP R1, R1, [R2] ; [R2] 和 R1 的内容进行交换
- STR存数指令：
  - 将寄存器中的数，存入存储器
- SWP交换指令：
  - SWP{cond} {B} Rd, Rm, [Rn] ; 先将[Rn]的内容存入Rd,  
; 再将Rm的内容存入[Rn],  
; B 为可选后缀，若有B,  
; 则交换字节，否则交换32 位字

## • 2.3.5 基址变址寻址（基址寻址）

- 基址变址寻址有4种形式：

- （1）零偏移（Zero Offset）

- op Rd, [Rn, R1] ; 有效地址 =  $Rn + R1$

- （2）前索引偏移（Pre-Indexed）

- op Rd, [Rn, FlexOffset] ; 有效地址 =  $Rn + FlexOffset$

- （3）带写回的前索引偏移（Pre-Indexed with Writeback）

- op Rd, [Rn, FlexOffset]! ; 有效地址 =  $Rn + FlexOffset$ ,  
; 并且  $Rn + FlexOffset \rightarrow Rn$   
; “!” 表示自增（自减）

- （4）后索引偏移（Post-Indexed）

- op Rd, [Rn], FlexOffset ; 有效地址 =  $Rn$ ,  
; 并且  $Rn + FlexOffset \rightarrow Rn$

- **FlexOffset**: 称为**灵活的偏移量**，它可以有以下两种形式：
  - **#expr**
  - **{-} Rm {, shift}**
- **FlexOffset**与**op2**（第2个源操作数）很相似，但有一些不同的地方：
  - **#expr**表示的整数范围是： **-4095 ~ +4095**
  - **Rm**不允许是**R15**（即**PC**，程序计数器）
  - 有 **{-}** 选项
- **R13**: **SP**，堆栈指针
- **R14**: **LR**，链接寄存器
- **R15**: **PC**，程序计数器

## • 2.3.6 多寄存器寻址

- **LDMIA R0, {R1, R2, R3, R4, R5}**

- [R0] -> R1
- [R0+4] -> R2
- [R0+8] -> R3
- [R0+12] -> R4
- [R0+16] -> R5

- **STMIA R0, {R2-R5, R7}**

- R2 -> [R0]
- R3 -> [R0+4]
- R4 -> [R0+8]
- R5 -> [R0+12]
- R7 -> [R0+16]

- **LDM批量加载（读取）指令**

- 连续从一片存储器读数，送一组寄存器中


- **STM批量存储指令**

- 将一组寄存器中的数，连续存入一片存储器中

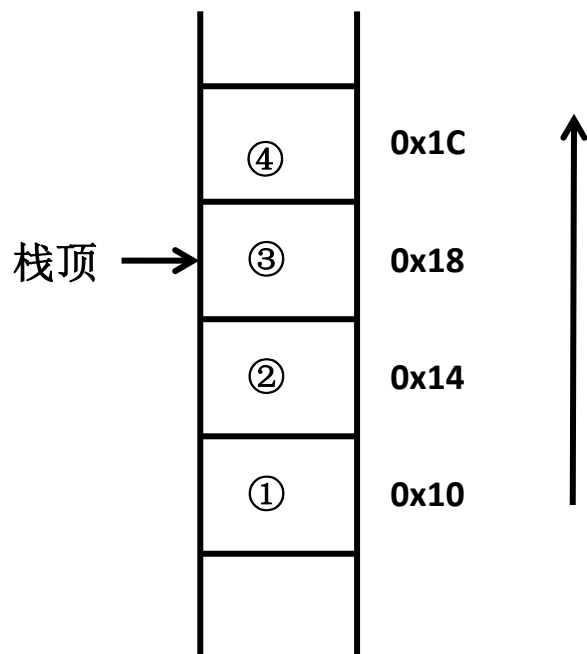
- **IA（后增）**

- 操作完成后，地址递增

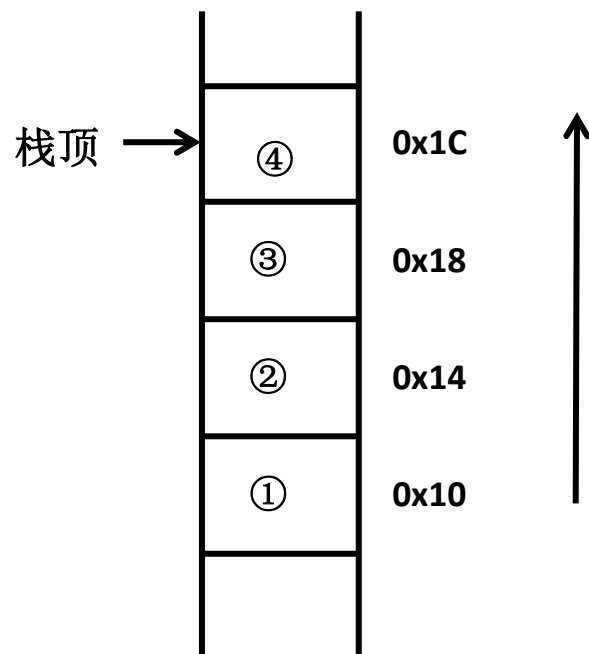
### • 2.3.7 堆栈寻址

- 堆栈指针（SP）： R13
- 满堆栈： Full Stack
- 空堆栈： Empty Stack
- 递增堆栈： Ascending Stack
- 递减堆栈： Descending Stack
- 满递增堆栈（满增）： FA
- 满递减堆栈（满减）： FD
- 空递增堆栈（空增）： EA
- 空递减堆栈（空减）： ED
- **STM****FD** **SP!**, {R1-R7, LR} ; 将LR、 R7-R1, 存放到堆栈中,  
 ; LR -> [SP-4], R7 -> [SP-8], R6 -> [SP-12],  
; R5 -> [SP-16], R4 -> [SP-20], R3 -> [SP-24],  
; R2 -> [SP-28], R1 -> [SP-32],  
; “!” 表示自减

**FA**  
(满增)

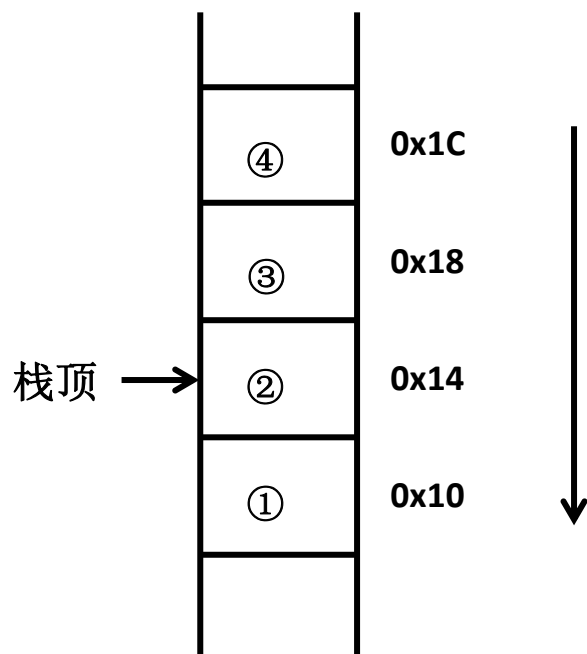


**EA**  
(空增)

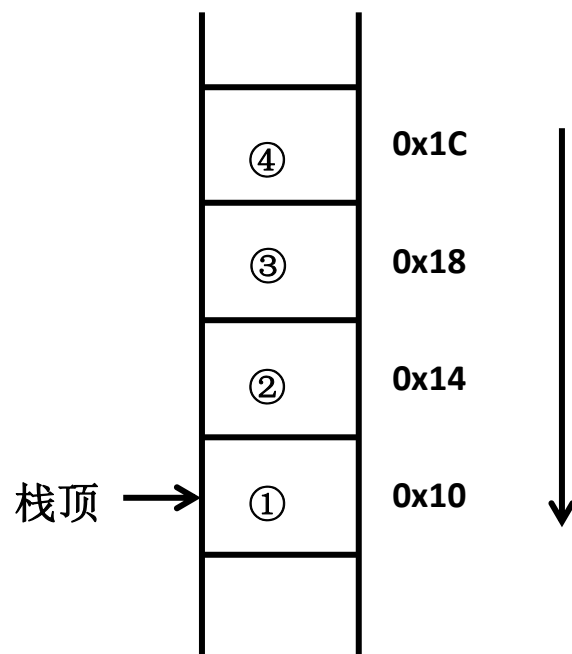




**FD**  
(满减)



**ED**  
(空减)



STM**FD** **SP**!, {R1-R7, LR}

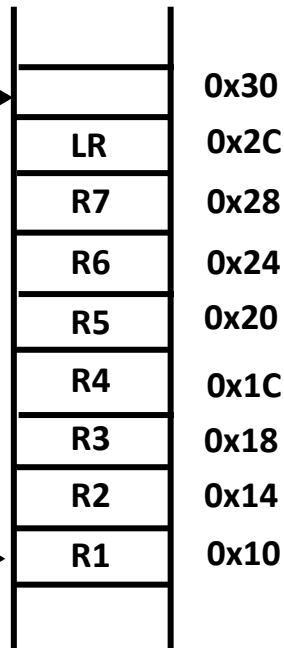


;将R1-R7、LR，存放到堆栈中

**FD**  
(满减)

开始时: SP=0x30

SP



结束时: SP=0x10

SP



## • 2.3.8 相对寻址

- 基址变址寻址的一个特例：PC作为基地址
- 一般用于指令跳转

```
        BL Label        ;转到Label标签处
        ...
Label:
        ...
```

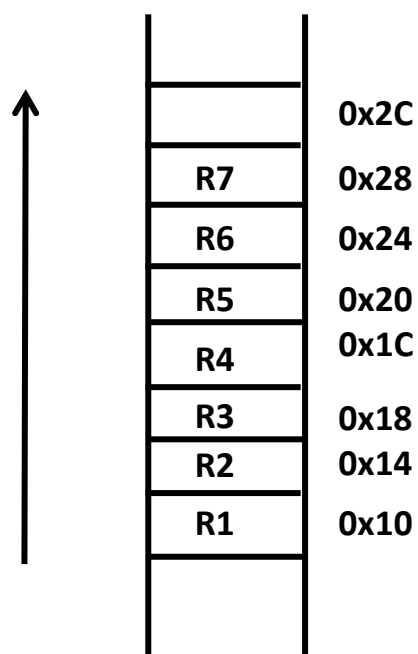
- BL指令：
  - 带链接的跳转指令

## • 2.3.9 块复制寻址（块拷贝寻址）

- **STMIA** **R0!**, {R1-R7} ;将R1~R7的数据保存到[R0]开始的存储器中，存储指针在保存第一个值之后递增
- **STMIB** **R0!**, {R1-R7} ;将R1~R7的数据保存到[R0]开始的存储器中，存储指针在保存第一个值之前递增
- **STMDA** **R0!**, {R1-R7} ;将R7~R1的数据保存到[R0]开始的存储器中，存储指针在保存第一个值之后递减
- **STMDB** **R0!**, {R1-R7} ;将R7~R1的数据保存到[R0]开始的存储器中，存储指针在保存第一个值之前递减
- **IA**（后增）：操作完成后，地址递增
- **IB**（先增）：地址先增，而后完成操作
- **DA**（后减）：操作完成后，地址递减
- **DB**（先减）：地址先减，而后完成操作
- **I**：增加（增），Increase
- **D**：减少（减），Decrease
- **A**：之后（后），After
- **B**：之前（先），Before
- **!**：表示自增（自减）

# STM (存数)

IA  
(后增)

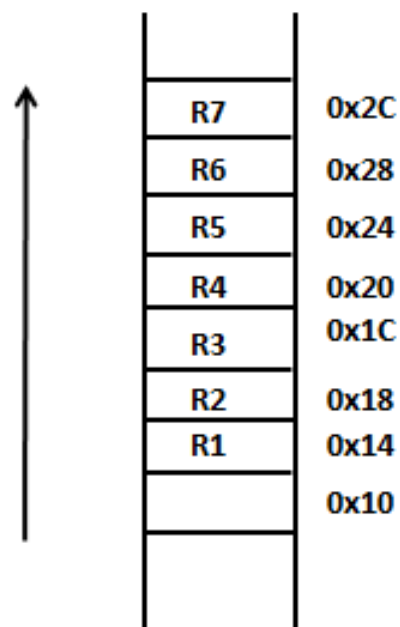


假设R0=0x10

→  
**STMIA** R0!, {R1-R7}

存储器操作

IB  
(先增)



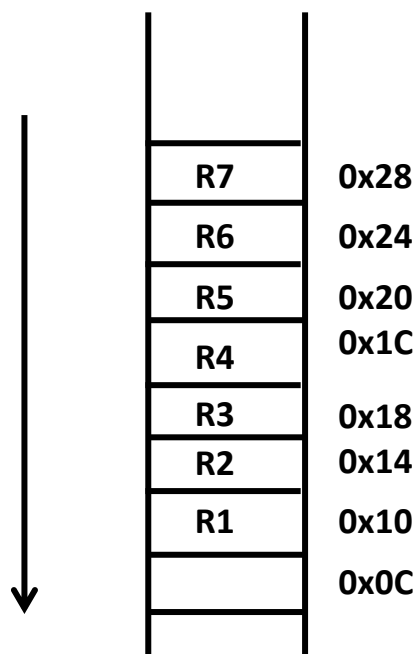
假设R0=0x10

→  
**STMIB** R0!, {R1-R7}

存储器操作

# STM (存数)

DA  
(后减)

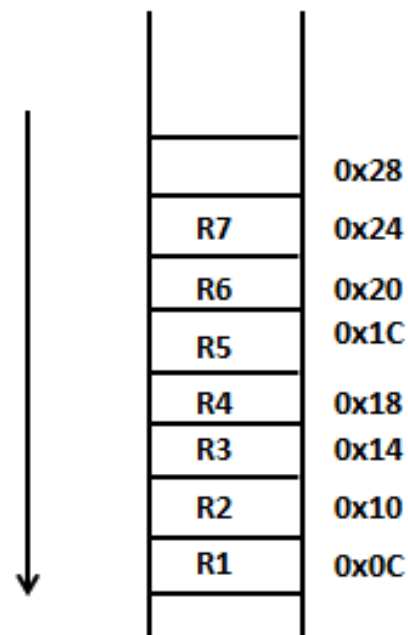


假设R0=0x28

**STMDA** R0!, {R1-R7}

存储器操作

DB  
(先减)



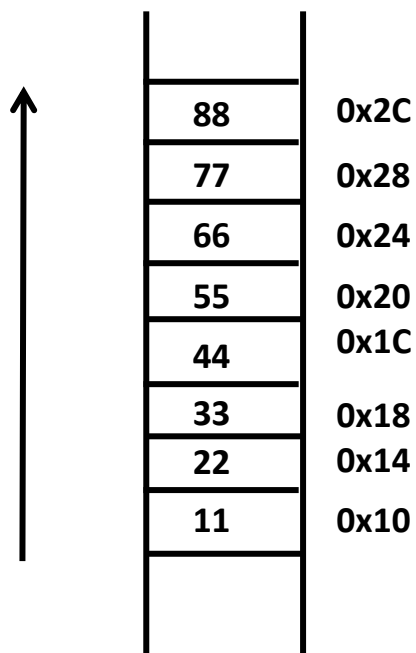
假设R0=0x28

**STMDB** R0!, {R1-R7}

存储器操作

# LDM (取数)

IA  
(后增)



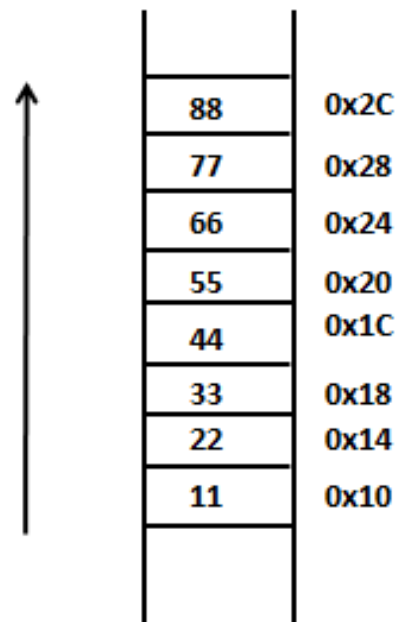
假设R0=0x10

**LDMIA** R0!, {R1-R7}

存储器操作

R1=11  
R2=22  
R3=33  
R4=44  
R5=55  
R6=66  
R7=77

IB  
(先增)



假设R0=0x10

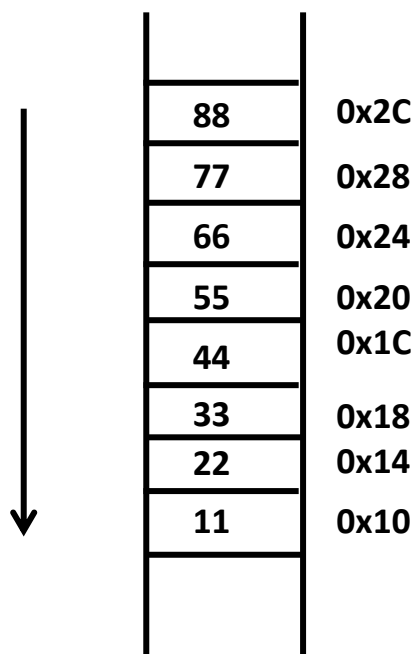
**LDMIB** R0!, {R1-R7}

存储器操作

R1=22  
R2=33  
R3=44  
R4=55  
R5=66  
R6=77  
R7=88

# LDM (取数)

**DA**  
(后减)



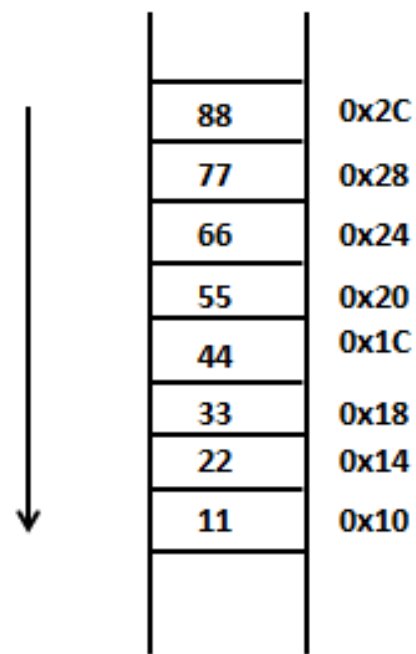
假设R0=0x2C

**LDMDA** R0!, {R1-R7}

存储器操作

R7=88  
R6=77  
R5=66  
R4=55  
R3=44  
R2=33  
R1=22

**DB**  
(先减)



假设R0=0x2C

**LDMDB** R0!, {R1-R7}

存储器操作

R7=77  
R6=66  
R5=55  
R4=44  
R3=33  
R2=22  
R1=11



## 2.4 ARM指令简介

- 指令类型（8类）：
  - ① 跳转指令
  - ② 通用数据处理指令
  - ③ 乘法指令
  - ④ Load/Store内存访问指令
  - ⑤ ARM协处理器指令
  - ⑥ 杂项指令
  - ⑦ 饱和算术指令
  - ⑧ ARM伪指令
- 指令格式：<opcode> {<cond>} {S} <Rd>, <Rn> {, <shift\_op2>}
  - <>内的项是必须的，{}内的项是可选的
  - opcode: 指令助记符（操作码），如LDR, STR等
  - cond: 执行条件（条件码），如EQ, NE等
  - S: 可选后缀，加S时影响CPSR中的条件码标志位，不加S时则不影响
  - Rd: 目标寄存器
  - Rn: 第1个源操作数的寄存器
  - op2: 第2个源操作数
  - shift: 位移操作

## • 2.4.1 跳转指令

– 方法1：将PC值改为跳转（转移）的目的地址

- **MOV PC, #immediate** ; PC <- #immediate

- **LDR PC, [PC, #offset]** ; PC <- [PC + offset]

- **LDR**指令：从存储器读数，送寄存器

## – 方法2：跳转指令（4条）

- (1) **B**: 基本的跳转指令
  - 指令格式: **B {cond}, Label**
- (2) **BL**: 带链接的跳转指令
  - 指令格式: **BL {cond}, Label**
- (3) **BX**: 用于ARM状态和Thumb状态之间的切换
  - 指令格式: **BX {cond}, Rm** ;Rm中存储了目标跳转地址
  - 例子:

**CODE32**

...

**ADR R1, Label + 1**

**BX R1**

**Label2**

...

**CODE16**

...

**LDR R2, = Label2**

**BX R2**

- (4) **BLX**: 完成指令跳转、返回位置保存、处理器工作状态切换等3个动作

- 指令格式:

- » **BLX {cond}, Rm**
- » **BLX Label**

- 例子:

- » **BLX R2**
- » **BLXNE R2** ;NE表示“不相等”
- » **BLX Thumblabel**

- 错误的指令:

- » **BLXMI Thumblabel** ;MI表示“负数”  
; 相对地址跳转指令必须是  
; 无条件的: **BLX Label**

## • 2.4.2 通用数据处理指令

– 使用通用数据处理指令时，有3条注意事项（具体见P32）

– 1、数据传送指令（2条）

- （1）**MOV**：传送
- （2）**MVN**：取反后传送

• 指令格式：

- **MOV** {cond} {S} Rd, Operand2 ; Rd <- Operand2
- **MVN** {cond} {S} Rd, Operand2 ; Rd <- /Operand2

» cond 执行条件，如EQ, NE 等

» S 是否影响CPSR 寄存器的值，书写时影响CPSR，否则不影响

• 例子：

- **MOV** R0, R0, ASR R2 ;R0算术右移R2位 -> R0
- **MVN****NE** R1, #0x22 ;不相等时, /#0x22 -> R1
- ; 0xFFFFFFFFDD -> R1

• 错误的指令：

- **MVN** **R15**, R0, ASR R2 ;违反第3条注意事项

## – 2、算术逻辑运算指令

- 算术运算指令（6条）：

- (1) **ADD**: 加法                      **ADD R1,R2,R3 ; R2+R3 -> R1**
- (2) **ADC**: 带进位加法
- (3) **SUB**: 减法                      **SUB R1,R2,R3 ; R2-R3 -> R1**
- (4) **SBC**: 带进位减法
- (5) **RSB**: 逆向减法              **RSB R1,R2,R3 ; R3-R2 -> R1**
- (6) **RSC**: 带进位逆向减法

- 指令格式: **op {cond} {S} Rd, Rn, Operand2**

- » **cond** 执行条件, 如EQ, NE 等

- » **S** 是否影响CPSR 寄存器的值, 书写时影响CPSR, 否则不影响

- 例子:

- » **ADD****S** R0, R1, #1280

- » ;带S后缀, R1 + #1280 -> R0

- » **SUB****HI** R1, R2, R3

- » ;如果“无符号数大于”,

- » ;则: R2-R3 -> R1

- » **RSB****ES** R1, R4, R1, LSL R3

- » ;如果,

- » ;则: R1逻辑左移R3位 – R1 -> R4

- 错误的指令:

- » **RSB****ES** R1, **R15**, R1, LSL R3

- » ;违反第3条注意事项

- 逻辑运算指令（4条）：

- （1）**AND**：逻辑与
- （2）**ORR**：逻辑或
- （3）**EOR**：逻辑异或
- （4）**BIC**：位清零

- » **BIC R0, R1, R1** ; R1 逻辑与 /R1 -> R0, 相当于 0 -> R0

- 指令格式：op {cond} {S} Rd, Rn, Operand2

- » cond 执行条件，如EQ, NE 等

- » S 是否影响CPSR 寄存器的值，书写时影响CPSR，否则不影响

- 例子：

- » **EOR R0, R1, #xFF00**

- » ; R1 逻辑异或 #xFF00 -> R0

- » **ORR R1, R2, R4, LSR #2**

- » ; R2 逻辑或 R4逻辑右移2位 -> R1

- » **BICNES R5, R6, R1, RRX**

- » ; 带S后缀，如果“NE，不相等”，

- » ; 则：R6 逻辑与

- » ; “R1带扩展循环右移后取反” -> R5

- » **NE**：不相等

- » **S**：带S后缀

- » **RRX**：寄存器扩展循环右移

- 错误的指令：

- » **ANDS R0, R15, R1, LSL R3**

- » ;违反第3条注意事项

### – 3、比较指令（4条）

- (1) **TST**: 位测试指令
  - TST R1, R2 ;R1和R2按位进行**逻辑与**操作
- (2) **TEQ**: 相等测试指令
  - TEQ R1, R2 ;R1和R2按位进行**逻辑异或**操作
- (3) **CMP**: 比较指令
  - CMP R1, R2 ;R1和R2**相减**
- (4) **CMN**: 反值比较指令
  - CMN R1, R2 ;R1和R2**相加**
- 指令格式: op {cond} Rn, Operand2
  - cond 执行条件, 如EQ, NE 等
- 例子:
  - TST R1, #0x0F ; R1 和 #0x0F 按位进行**逻辑与**操作
  - TEQ**NE** R9, #0x4000 ;如果“NE, 不相等”,  
;则: R19 和 #0x4000 按位进行**逻辑异或**操作
- 错误的指令:
  - TST**NE** **R15**, R1, LSL R0 ;违反第3条注意事项



#### — 4、前导零计数指令（1条）

- **CLZ**

- 指令格式: **op {cond} Rd, Rm**
  - cond 执行条件, 如EQ, NE 等

- 例子:
  - **CLZ R1, R2** ;统计R2中从高位开始连续零的个数,  
;结果放到R1中; 如果R2=0, 则结果为32;  
;如果R2=0x0FF00000, 则结果为4

## • 2.4.3 乘法指令

### – 1、MUL和MLA指令（2条）

- （1）**MUL**：乘法指令（32位X32位->取64位的低32位）

- 指令格式：MUL {cond} {S} Rd, Rm, Rs

- 例子：MUL R1, R2, R3 ;R2XR3 -> R1

- （2）**MLA**：乘加指令（32位X32位+32位->取64位的低32位）

- 指令格式：MLA {cond} {S} Rd, Rm, Rs, Rn

- 例子：MLA R1, R2, R3, R4 ;R2XR3+R4 -> R1

- 例子：

- **MULT** R7, R7, R8

;如果“LT，带符号数小于”，

;则：R7XR8（结果的低32位） -> R7

- MLA R3, R2, R4, R6

;R2XR4+R6（结果的低32位） -> R3

## – 2、UMULL、UMLAL、SMULL和SMLAL指令（4条）

- (1) **UMULL**: 无符号长整形乘法指令（32位X32位=64位）
- (2) **UMLAL**: 无符号长整形乘加指令（32位X32位=64位）
- (3) **SMULL**: 带符号长整形乘法指令（32位X32位=64位）
- (4) **SMLAL**: 带符号长整形乘加指令（32位X32位=64位）

– U: 无符号; S: 带符号; L: 长整形（64位）

### • 指令格式:

– op {cond} {S} RdLo, RdHi, Rm, Rs ;RmXR<sub>s</sub> -> RdHi RdLo（64位）

### • 例子:

– UMULL**S** R1, R2, R3, R4 ;带S后缀, R3XR<sub>4</sub> -> R2 R1（64位）

– UMLAL**NE** R1, R2, R3, R1 ;如果“NE, 不相等”, 则: R3XR<sub>1</sub> -> R2 R1（64位）

– SMULL R5, R4, R3, R2 ;R3XR<sub>2</sub> -> R4 R5（64位）

– SMULL**LES** R1, R2, R3, R4 ;如果“LE, 带符号数小于等于”,  
;则: R3XR<sub>4</sub> -> R2 R1（64位）  
;带S后缀

### – 3、SMULxy和SMLAxy指令（2条）

- （1）**SMULxy**：16位带符号乘法指令（16位X16位=32位）
- （2）**SMLAxy**：16位带符号乘加指令（16位X16位=32位）
- 指令格式：
  - **SMUL**<x><y> {cond} Rd, Rm, Rs
  - **SMLA**<x><y> {cond} Rd, Rm, Rs, Rn
- **x、y**：分别对应第一个操作数和第二个操作数，可以是**B**或**T**，**B**时取操作数的低16位，**T**时取操作数的高16位
- 例子：
  - **SMULBT** R1,R2,R3 ;R2[15:0] X R3[31:16] -> R1
  - **SMLABT** R1,R2,R3,R4 ;R2[15:0] X R3[31:16] + R4 -> R1

## – 4、SMULW<sub>y</sub>和SMLAW<sub>y</sub>指令（2条）

- （1）**SMULW<sub>y</sub>**: 32位X16位（取48位的高32位）带符号乘法指令
- （2）**SMLAW<sub>y</sub>**: 32位X16位（取48位的高32位）带符号乘加指令
- 指令格式:
  - SMULW<y> {cond} Rd, Rm, Rs
  - SMLAW<y> {cond} Rd, Rm, Rs, Rn
- **y**: 对应第二个操作数，可以是B或T，**B**时取操作数的低16位，**T**时取操作数的高16位
- 例子:
  - **SMULWTVS R1,R2,R1** ;如果“VS，溢出”，则：R2XR1[31:16]（将相乘后48位（32位X16位，得到48位）的高32位）-> R1
  - **SMLAWT R2,R2,R4,R4** ;R2XR4[31:16]（将相乘后48位（32位X16位，得到48位）的高32）+ R4 -> R2

## • 2.4.4 Load/Store内存访问指令

### – 1、LDR和STR指令（2条）

- （1）**LDR**：从存储器读数，送寄存器
- （2）**STR**：将寄存器中的数，存入存储器

• 共有3种情况：

### • ①字（32位）或无符号字节（8位）传输

– 指令格式：

- » **op{cond} {B} {T} Rd,[Rn]**
- » **op{cond} {B} Rd,[Rn, FlexOffset] {!}** ;（带写回的）前索引偏移
- » **op{cond} {B} Rd,label**
- » **op{cond} {B} {T} Rd,[Rn],FlexOffset** ;后索引偏移
- » **B**：加上此后缀，表示进行无符号字节传输，**Rd[7:0]**
- » **T**：加上此后缀，内存系统会认为处理器运行在**用户模式**上，虽然可能处理器实际是运行在**特权模式**下

- FlexOffset（灵活的偏移量），用于基址变址寻址

- 基址变址寻址有4种形式：

- （1）零偏移（Zero Offset）

- op Rd, [Rn, R1] ; 有效地址 = Rn + R1

- （2）前索引偏移（Pre-Indexed）

- op Rd, [Rn, FlexOffset] ; 有效地址 = Rn + FlexOffset

- （3）带写回的前索引偏移（Pre-Indexed with Writeback）

- op Rd, [Rn, FlexOffset]! ; 有效地址 = Rn + FlexOffset,  
; 并且 Rn + FlexOffset -> Rn

- （4）后索引偏移（Post-Indexed）

- op Rd, [Rn], FlexOffset ; 有效地址 = Rn,  
; 并且 Rn + FlexOffset -> Rn

– 例子:

» LDR R0, [R1]

; [R1]存储单元的内容 -> R0

» STRT R2, [R0, R3, LSL #0x2]!

; 有效地址=R0 + R3逻辑左移2位

; 并且 R0 + R3逻辑左移2位 -> R0

; R2存入 [R0 + R3逻辑左移2位]存储单元中

; 处理器运行在用户模式上

» LDRB R4, [R8], #0x4

; 有效地址=R8

; 并且 R8 + #0x4 -> R8

; [R8]存储单元内容的低8位（字节） -> R



- ②半字（16位）或带符号字节（8位）传输

- 指令格式:

- » `op{cond} type Rd,[Rn]`
    - » `op{cond} type Rd,[Rn, Offset] {!}` ; (带写回的) 前索引偏移
    - » `op{cond} type Rd,label`
    - » `op{cond} type Rd,[Rn],Offset` ;后索引偏移
    - » type: (**S**: 带符号; **H**: 半字 (16位); **B**: 字节 (8位))
      - SH: 带符号的半字 适用指令: LDR
      - H: 无符号的半字 适用指令: LDR、STR
      - SB: 带符号的字节 适用指令: LDR

- 例子:

- » `LDRSH R0, [R1, #0xF2]` ;[R1 + #0xF2]存储单元内容的带符号半字 -> R0
    - » `STRH R2, [R0, R3]!` ;[R0 + R3]存储单元内容的无符号半字 -> R2  
; 并且 R0+R3 -> R0
    - » `LDRB R4, Lable` ;[Label]存储单元的无符号字节 -> R4

- 错误的指令:

- » `LDRSH R0, [R1, R5, LSL #0x2]` ;第2个操作数 (R5) 不允许有移位操作 (LSL)

### • ③双字（64位）传输

#### – 指令格式：

- » `op{cond} D Rd,[Rn]`
- » `op{cond} D Rd,[Rn, Offset] {!}` ;（带写回的）前索引偏移
- » `op{cond} D Rd,label`
- » `op{cond} D Rd,[Rn], Offset` ;后索引偏移

» **D**: 表示双字（64位）

#### – 例子：

- » `LDRD R8, [R1], #0xF2` ;[R1]开始的2个存储单元的内容 -> R8和R9
- » `STREQD R2, [R0, -R4]!` ;如果“EQ，相等”，则：  
; [R0-R4]开始的2个存储单元的内容 -> R2和R3  
; 并且 R0-R4 -> R0
- » `LDRMID R4, Lable` ;如果“MI，是负数”，则：  
; [Label]开始的2个存储单元的内容 -> R4和R5

#### – 错误的指令：

- » `LDRD R5, [R1], #0xF2` ;Rd（现在是R5）必须是偶数寄存器
- » `STREQD R2, [R3, -R4]!` ;Rn（现在是R3）不能是Rd或R(d+1),  
;即不能是R2或R3
- » `LDRMID R14, Lable` ;Rd不能是R14

## – 2、LDM和STM指令（2条）

- **LDM**: 批量加载（连续从一片存储器读数，送一组寄存器中）
- **STM**: 批量存储（将一组寄存器中的数，连续存入一片存储器中）

- 指令格式: **op{cond} mode Rn{!}, reglist {^}**

- **mode**（8种，2大类）：

- » 用于数据的加载和存储（4种）

- **IA**（后增）：操作完成后，地址递增
      - **IB**（先增）：地址先递增，而后完成操作
      - **DA**（后减）：操作完成后，地址递减
      - **DB**（先减）：地址先递减，而后完成操作

- » 用于堆栈操作（4种）

- **FA**（满增）：满递增堆栈
      - **FD**（满减）：满递减堆栈
      - **EA**（空增）：空递增堆栈
      - **ED**（空减）：空递减堆栈

- » 缩写含义：

- **I**: 增加, **Increase**
      - **D**: 减少, **Decrease**
      - **A**: 之后（后）, **After**
      - **B**: 之前（先）, **Before**
      - **F**: 满, **Full**
      - **E**: 空, **Empty**

- **Rn**: 存储器的有效地址

- **{!}**: 表示自增（自减）

- **reglist**: 表示多个寄存器集合

- **{^}**: 具体含义见P39-40

- STMIB = STMFA
- STMIA = STMEA
- STMDB = STMFD
- STMDA = STMED

- LDMIB = LDMED
- LDMIA = LDMFD
- LDMDB = LDMEA
- LMDA = LDMFA

• 例子:

» LDMFA R5, {R1,R3,R5,R7} ;将[R5]开始的4个存储单元的内容  
; -> R1、R3、R5、R7

» STMFD R13!, {R0-R4,LR} ;将R0-R4及LR，压入堆栈（R13即SP）

» LDMFD R13!, {R0-R4,PC} ;从堆栈中读（恢复R0-R4及PC）  
;R13即SP

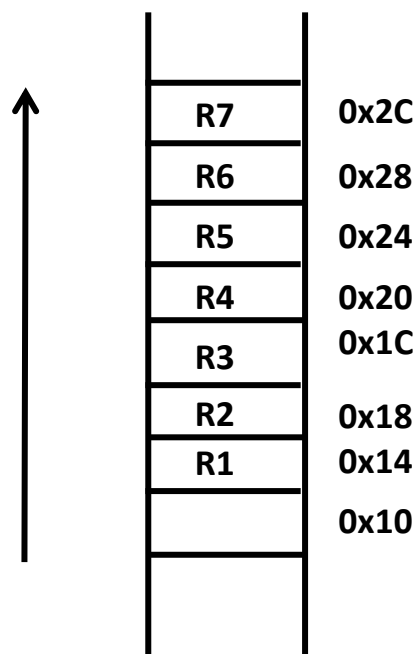
• 错误的指令:

» STMIB R6!, {R1,R3,R6,R7} ;

» LDMFA R3, {} ;{}中至少包含一个寄存器

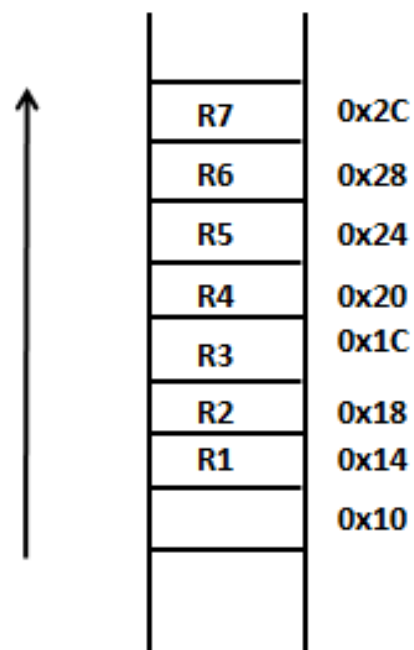
# STM（存数）

**IB**  
(先增)



假设R0=0x10

**FA**  
(满增)



假设R0=0x10

→  
**STMIB R0!, {R1-R7}**

存储器操作

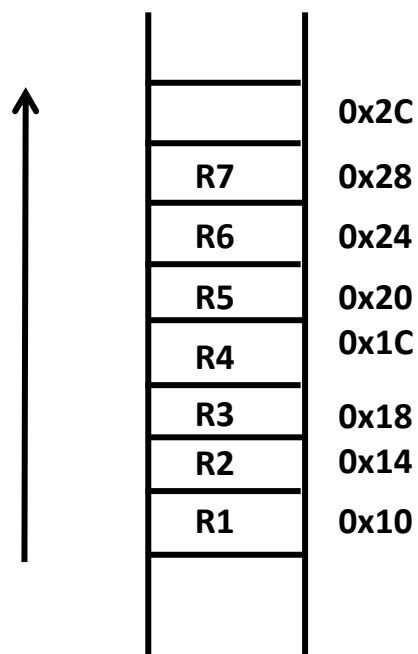
=

→  
**STMFA R0!, {R1-R7}**

存储器操作

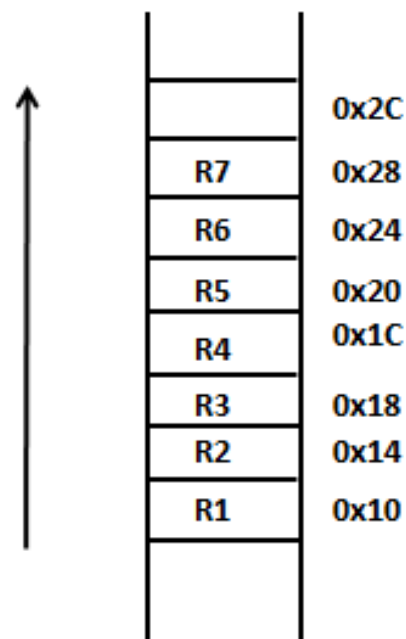
# STM（存数）

IA  
(后增)



假设R0=0x10

EA  
(空增)



假设R0=0x10

→  
**STMIA R0!, {R1-R7}**

存储器操作

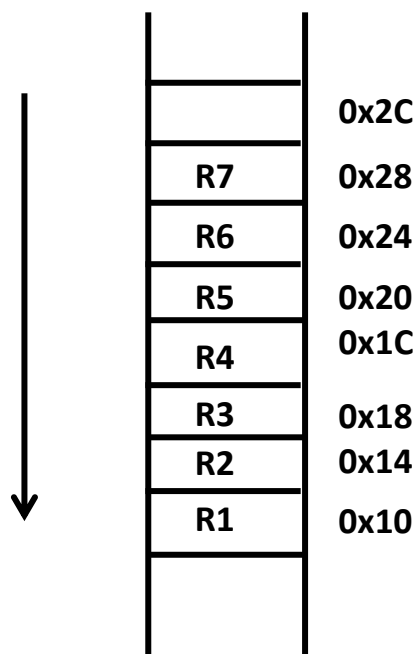
=

→  
**STMEA R0!, {R1-R7}**

存储器操作

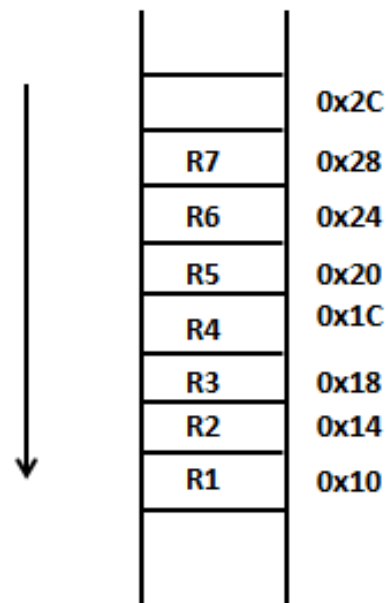
# STM (存数)

**DB**  
(先减)



假设R0=0x2C

**FD**  
(满减)



假设R0=0x2C

**STMDB R0!, {R1-R7}**

存储器操作

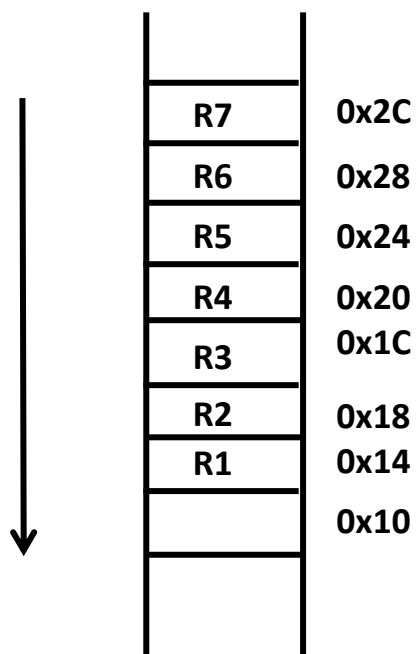
=

**STMFD R0!, {R1-R7}**

存储器操作

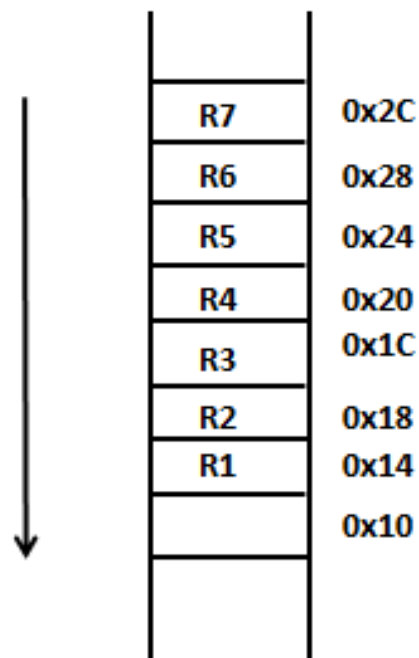
# STM (存数)

DA  
(后减)



假设R0=0x2C

ED  
(空减)



假设R0=0x2C

←  
**STMDA R0!, {R1-R7}**

存储器操作

=

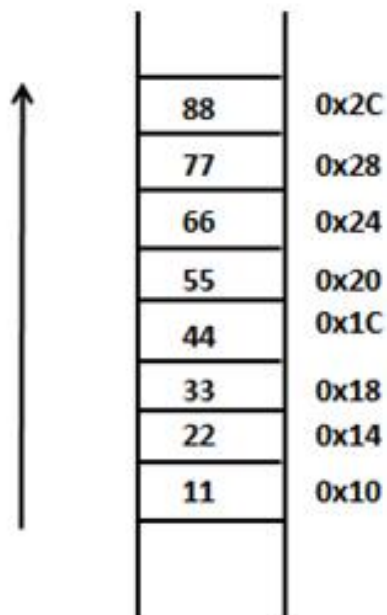
←  
**STMED R0!, {R1-R7}**

存储器操作



# LDM (取数)

**IB**  
(先增)



假设R0=0x10

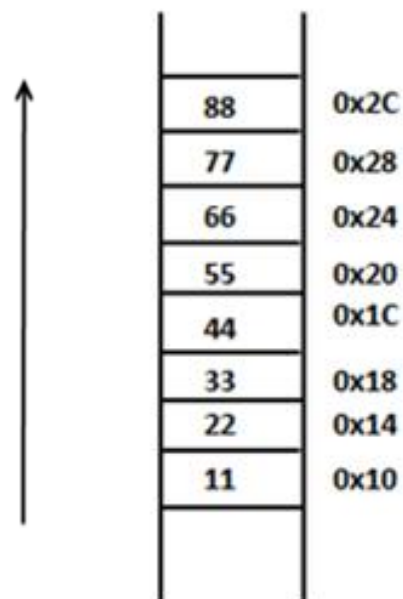
**LDMIB** R0!, {R1-R7}

存储器操作

R1=22  
R2=33  
R3=44  
R4=55  
R5=66  
R6=77  
R7=88

=

**ED**  
(空减)



假设R0=0x10

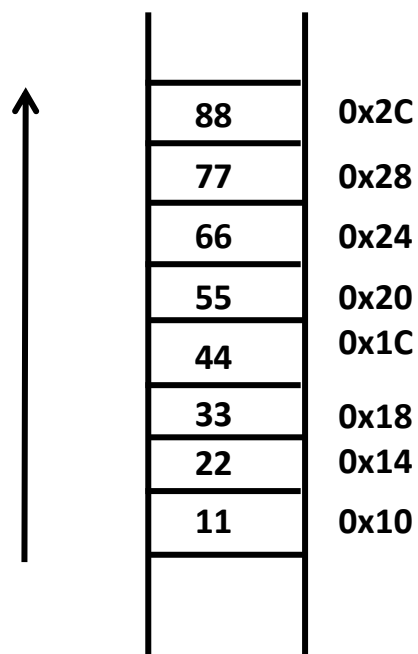
**LDMED** R0!, {R1-R7}

存储器操作

R1=22  
R2=33  
R3=44  
R4=55  
R5=66  
R6=77  
R7=88

# LDM (取数)

IA  
(后增)



假设R0=0x10

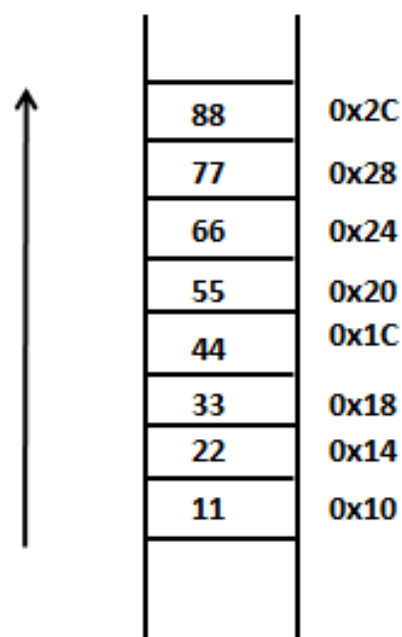
**LDMIA R0!, {R1-R7}**

存储器操作

R1=11  
R2=22  
R3=33  
R4=44  
R5=55  
R6=66  
R7=77

=

FD  
(满减)



假设R0=0x10

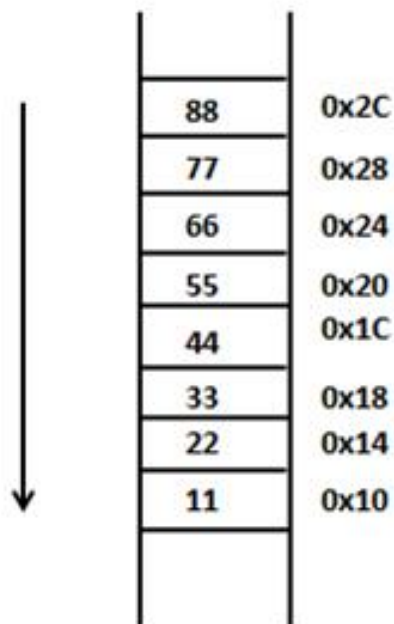
**LDMFD R0!, {R1-R7}**

存储器操作

R1=11  
R2=22  
R3=33  
R4=44  
R5=55  
R6=66  
R7=77

# LDM (取数)

DB  
(先减)



假设R0=0x2C

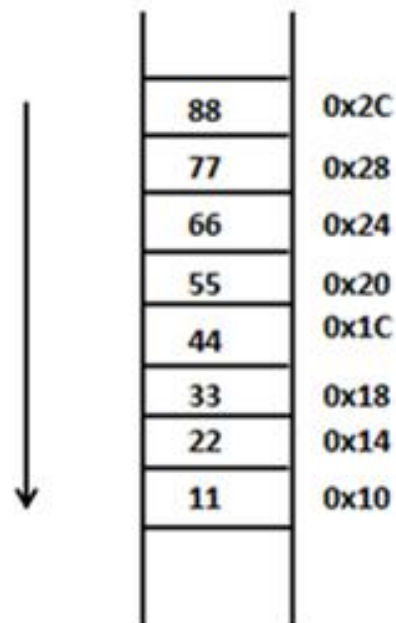
**LDMDB** R0!, {R1-R7}

存储器操作

R1=77  
R2=66  
R3=55  
R4=44  
R5=33  
R6=22  
R7=11

=

EA  
(空增)



假设R0=0x2C

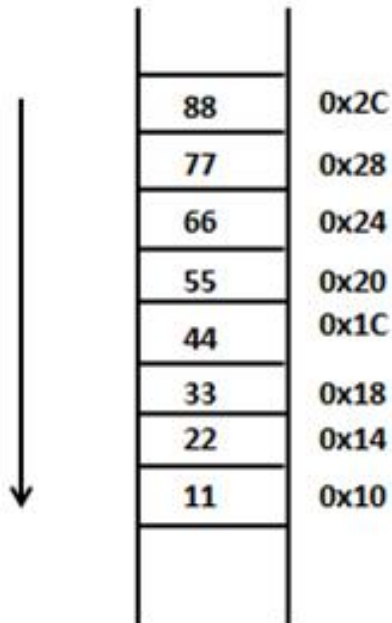
**LDMEA** R0!, {R1-R7}

存储器操作

R1=77  
R2=66  
R3=55  
R4=44  
R5=33  
R6=22  
R7=11

# LDM (取数)

DA  
(后减)



假设R0=0x2C

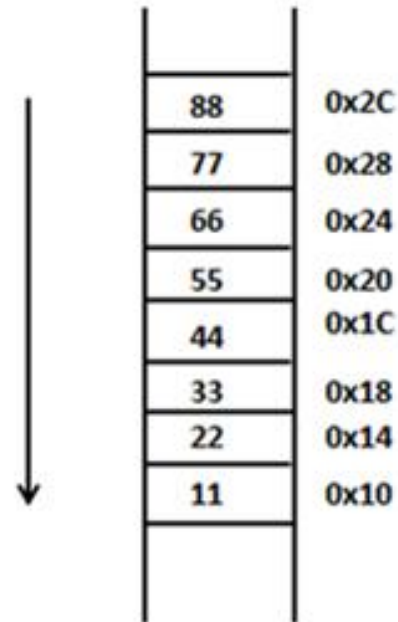
←  
**LDMDA R0!, {R1-R7}**

存储器操作

R1=88  
R2=77  
R3=66  
R4=55  
R5=44  
R6=33  
R7=22

=

FA  
(满增)



假设R0=0x2C

→  
**LDMFA R0!, {R1-R7}**

存储器操作

R1=88  
R2=77  
R3=66  
R4=55  
R5=44  
R6=33  
R7=22

### – 3、SWP指令（1条）

- **SWP**：交换指令，用于寄存器和存储器之间的内容交换
- 指令格式：**SWP{cond} {B} Rd, Rm, [Rn]**
  - 先将[Rn]的内容存入Rd，再将Rm的内容存入[Rn]
  - **B**：表示交换的数据是字节
- 例子：
  - **SWPB R2, R3, [R4]** ; 字节交换：[R4] -> R2, R3 -> [R4]
  - **SWP R1, R1, [R5]** ; [R5] -> R1, R1 -> [R5]  
; 即R1和[R5]的内容交换

## – 4、PLD指令（1条）

- **PLD**: 预读取指令，指示存储器系统在接下去的几条指令中很可能会有**Load**指令，存储系统以此做好相应的准备，从而加速内存访问过程
- 指令格式: **PLD [Rn{, FlexOffset}]**
  - **Rn**: 对应内存地址的基地址
  - **FlexOffset**: 灵活的偏移量，用于基址变址寻址
- 例子:
  - **PLD [R2, #Label \*5]** ;有效地址= $R2 + \#Label * 5$
  - **PLD [R3, R2, LSR #0x2]** ;有效地址= $R3 + R2$ 逻辑右移2位

## • 2.4.5 ARM协处理器指令

- ARM共有**16个协处理器**（Coprocessor, P0-P15）
- 1、CDP和CDP2指令（2条）
  - （1）**CDP**：协处理器数据处理指令
    - 指令格式：CDP {cond} coproc, opcode1, CRd, CRn, CRm {, opcode2}
      - » coproc：协处理器的名字（16个，P0-P15）
      - » CRd、CRn、CRm：协处理器寄存器（16个，C0-C15）
      - » opcode1、opcode2：协处理器相关的操作码
  - （2）**CDP2**：协处理器数据处理指令（不能条件执行）
    - 指令格式：CDP2 coproc, opcode1, CRd, CRn, CRm {, opcode2}
- 例子：
  - CDP P3, 2, C12, C10, C3, 4 ;完成协处理器P3的初始化
  - CDP2 P6, 1, C3, C4, C5 ;完成协处理器P6的初始化

## – 2、LDC、STC、LDC2和STC2指令（4条）

- (1) **LDC**: 将存储器的内容, 复制到协处理器寄存器中
- (2) **STC**: 将协处理器寄存器的内容, 复制到存储器中

– 指令格式:

- » `op {cond} {L} coproc, CRd, [Rn]` ;零偏移
- » `op {cond} {L} coproc, CRd, [Rn, #-}offset]{!}` ; (带写回的) 前索引偏移
- » `op {cond} {L} coproc, CRd, [Rn], #-}offset` ;后索引偏移
  
- » **L**: 表示长整数传送
- » **coproc**: 协处理器的名字
- » **CRd**: 协处理器寄存器

- (3) **LDC2**: 将存储器的内容, 复制到协处理器寄存器中 (不能进行条件执行, 且没有**L**后缀)
- (4) **STC2**: 将协处理器寄存器的内容, 复制到存储器中 (不能进行条件执行, 且没有**L**后缀)

– 指令格式:

- » `op coproc, CRd, [Rn]` ;零偏移
- » `op coproc, CRd, [Rn, #-}offset]{!}` ; (带写回的) 前索引偏移
- » `op coproc, CRd, [Rn], #-}offset` ;后索引偏移

• 例子:

- `LDC P5, C2, [R4, #0x8]!` ;将[R4+#0x8] -> P5的C2
- `STC P6, C2, [R3], #-0x7` ;将P6的C2 -> [R3]



### – 3、MCR、MCR2和MCRR指令（3条）

- （1）**MCR**：将ARM的寄存器内容，传送到协处理器的寄存器中
  - 指令格式：MCR {cond} coproc, opcode1, Rd, CRn, CRm {, opcode2}
- （2）**MCR2**：将ARM的寄存器内容，传送到协处理器的寄存器中（不能进行条件执行）
  - 指令格式：MCR2 coproc, opcode1, Rd, CRn, CRm {, opcode2}
- （3）**MCRR**：将ARM的寄存器内容，传送到协处理器的寄存器中（只有1个协处理器相关的操作码）
  - 指令格式：MCRR {cond} coproc, opcode1, Rd, Rn, CRm
- 例子：
  - MCR P7, 3, R1, C3, C2, 1 ;将ARM的R1寄存器内容，传送到协处理器P7的寄存器C2、C3中

## – 4、MRC、MRC2和MRRC指令（3条）

- （1）**MRC**：将协处理器的寄存器内容，传送到ARM的寄存器中
  - 指令格式：MRC {cond} coproc, opcode1, Rd, CRn, CRm {, opcode2}
- （2）**MRC2**：将协处理器的寄存器内容，传送到ARM的寄存器中（不能进行条件执行）
  - 指令格式：MCR2 coproc, opcode1, Rd, CRn, CRm {, opcode2}
- （3）**MRRC**：将协处理器的寄存器内容，传送到ARM的寄存器中（只有1个协处理器相关的操作码）
  - 指令格式：MRRC {cond} coproc, opcode1, Rd, Rn, CRm
- 例子：
  - MRC P4, 3, R1, C5, C6, 1 ;将协处理器P4的C5、C6寄存器内容，传送到ARM的寄存器R1中

- **2.4.6 杂项指令**

- **1、SWI指令（1条）**

- **SWI**: 软件中断指令

- 指令格式: **SWI {cond} immed\_24**
        - » **immed\_24**: 表示24位的立即数

- 例子:
        - » **SWI 0x22222**

## – 2、MRS和MSR指令（2条）

- (1) **MRS**: 状态寄存器（CPSR、SPSR）读指令

– 指令格式:

» MRS {cond} Rd, psr ; psr -> Rd

- psr: CPSR或SPSR

- (2) **MSR**: 状态寄存器（CPSR、SPSR）写指令

– 指令格式:

» MSR {cond} <psr>\_<field>, #immed\_8r ; #immed\_8r -> psr

» MSR {cond} <psr>\_<field>, Rm ; Rm -> psr

- psr: CPSR或SPSR
- field: CPSR或SPSR的域（4个域）
  - 标志位域: f, [31,24]
  - 状态域: s, [23,16]
  - 扩展域: x, [15,8]
  - 控制域: c, [7,0]
- #immed\_8r: 8位结构的常数

- 例子:

– MRS R5, SPSR ; SPSR -> R5

– MSR CPSR\_f, R7 ; R7->CPSR\_f, 即更新状态寄存器的标志位

### – 3、BKPT指令（1条）

- **BKPT**: 断点指令，使ARM处理器进入Debug模式

- 指令格式: **BKPT immed\_16**
  - » **immed\_16**: 16位的整数

- 例子:
  - » **BKPT 0xFF32**
  - » **BKPT 640**

## • 2.4.7 饱和算术指令

- 4条指令：
  - (1) **QADD**指令：加法指令，Q位会被置位
  - (2) **QSUB**指令：减法指令，Q位会被置位
  - (3) **QDADD**指令：加法指令，饱和运算， $SAT(Rm + SAT(Rn \times 2))$
  - (4) **QDSUB**指令：减法指令，饱和运算， $SAT(Rm - SAT(Rn \times 2))$
- Q位（Q标识位）：CPSR的第27位，在ARM V5的E系列处理器中，CPSR的bit[27]称为q标识位，主要用于指示增强的dsp指令是否发生了溢出。同样的spsr的bit[27]位也称为q标识位，用于在异常中断发生时保存和恢复CPSR中的Q标识位。在ARM V5以前的版本及ARM V5的非E系列的处理器中，Q标识位没有被定义
- SAT：饱和运算
- 指令格式：op {cond} Rd, Rm, Rn
- 例子：
  - QADD R3, R3, R2 ; R3+R2 -> R3
  - QDSUB R4, R3, R8 ; SAT(R3 - SAT(R2X2)) -> R4

## • 2.4.8 ARM伪指令

- 共4条指令
- 1、**ADR**: 小范围的地址读取伪指令（用于读取基于PC相对偏移的地址或基于寄存器相对偏移的地址）
  - 指令格式: **ADR {cond} register, expr**
    - » **register**: 目的寄存器
    - » **expr**: 基于PC或基于寄存器的表达式（相对偏移量）
  - 在汇编阶段，ADR伪指令通常会被ADD或SUB指令替代
  - 例子: **SUB R2, PC, 0xC** ;相对PC偏移12个字节
- 2、**ADRL**: 中等范围的地址读取伪指令
  - 指令格式: **ADRL {cond} L register, expr**
  - 在汇编阶段，ADRL伪指令通常会用两条指令替代
  - 例子:
    - » **Start MOV R4, #0x22**
    - » **ADR R2, start + 60000**
  - 替换为:
    - » **ADD R4, PC, #0xE800**
    - » **ADD R4,R4,#0x254**

- **3、LDR：**大范围的地址读取伪指令
  - 指令格式：LDR {cond} register, =[expr | label-expr]
  - 例子：
    - » LDR R1, =0xFF0
    - » LDR R2, =0xFFF
    - » LDR R3, =place
  - 替换为：
    - » MOV R1, 0xFF0
    - » LDR R2, [PC, offset\_to\_litpool]
    - » LDR R3, [PC, offset\_to\_litpool]
- **4、NOP：**空操作伪指令
  - 汇编时通常会替换为：
    - » MOV R0, R0



## 2.5 Thumb指令简介

指令字长为16位

指令字长为32位

- Thumb指令是**16位**的指令集，ARM指令是**32位**的指令集；Thumb指令集是ARM指令集的一个**子集**
- 许多Thumb指令都是**2地址格式**；大多数ARM指令是**3地址格式**
  - 指令格式：<opcode> {<cond>} {S} <**Rd**>, <**Rn**> {, <shift\_**op2**>}
- Thumb指令只有**B指令**（基本的跳转指令）是可以条件执行的指令，其他都不能条件执行；而大多数ARM指令是可以条件执行的

## • 2.5.1 Thumb跳转指令

### – 1、**B**: 基本的跳转指令

- 指令格式: **B{cond}**, label

可以条件执行

- 例子:

- **BEQ** label ;如果 (EQ, 相等), 则跳转到label
- **B** loop

### – 2、**BL**: 带链接的长跳转指令

- 指令格式: **BL** label

- 例子: **BL** section1

### – 3、**BX**: 在跳转的同时, 会选择性的切换指令集

- 指令格式: **BX** Rm

- 例子: **BX** R3

### – 4、**BLX**: 带链接的跳转, 并选择性的切换指令集

- 指令格式:

- **BLX** Rm
- **BLX** label

- 例子:

- **BLX** R4
- **BLX** armsub

## • 2.5.2 Thumb通用数据处理指令

### – 1、AND、ORR、EOR和BIC指令（4条）

- (1) **AND**: 逻辑与
- (2) **ORR**: 逻辑或
- (3) **EOR**: 逻辑异或
- (4) **BIC**: 位清零

– 指令格式: **op Rd, Rm** ;2个操作数

– 例子: **ORR R2, R3** ;R2 逻辑或 R3 -> R2

## – 2、ASR、LSL、LSR和ROR指令（4条）

- (1) **ASR**: 算术右移
- (2) **LSL**: 逻辑左移
- (3) **LSR**: 逻辑右移
- (4) **ROR**: 循环右移

### – 指令格式

- » **op Rd, Rs** ; Rd移Rs位, 结果放到Rd中
- » **op Rd, Rm, #expr** ; Rm移expr位, 结果放到Rd中

### – 例子:

- » **ASR R3, R5** ; R3算术右移R5位, 结果放到R3中
- » **LSR R0, R3, #5** ; R3逻辑右移5位, 结果放到R0中
- » **LSL R1, R4, #0** ; R4逻辑左移0位, 结果放到R1中

### – 错误的指令:

- » **ROR R2, R7, #5** ; ROR（循环右移）不允许使用立即数
- » **LSL R10, R1** ; 寄存器只能是R0-R7

### – 3、CMP和CMN指令（2条指令）

- (1) **CMP**: 比较指令
- (2) **CMN**: 反值比较指令

#### – 指令格式:

- » **CMP Rn, #expr** ;Rn-expr
- » **CMP Rn, Rm** ;Rn-Rm
- » **CMN Rn, Rm** ;Rn+Rm

#### – 例子:

- » **CMP R7, #255** ; R7-#255
- » **CMP R7, R12** ; R7-R12
- » **CMN R2,R3** ; R2+R3

#### – 错误的指令:

- » **CMP R3, #333** ; 立即数（#333）超出范围（0-255）
- » **CMP R12, #24** ;寄存器只能是R0-R7
- » **CMN R3,R10** ;寄存器只能是R0-R7

## – 4、MOV、MVN和NFG指令（3条指令）

- (1) **MOV**: 传送指令
- (2) **MVN**: 反值（按位取反）传送指令
- (3) **NFG**: 负值（乘以-1）传送指令

### – 指令格式:

- » **MOV Rd, #expr** ;expr -> Rd
- » **MOV Rd, Rm** ;Rm -> Rd
- » **MVN Rd, Rm** ;/Rm -> Rd
- » **NFG Rd, Rm** ;RmX(-1) -> Rd

### – 例子:

- » **MOV R3, #0** ; #0 -> R3
- » **MOV R0, R12** ;R12 -> R0
- » **MVN R7, R1** ;/R1 -> R7
- » **NFG R3, R3** ;R3X(-1) -> R3

## – 5、TST指令（1条）

- **TST**: 位测试指令

- 指令格式:

- » **TST Rn, Rm** ;执行Rn与Rm的逻辑与操作

- 例子:

- » **TST R3, R4** ; R3 逻辑与 R4

## • 2.5.3 Thumb算术指令

### – 1、低寄存器（R0-R7）的ADD（加法）和SUB（减法）指令

- 指令格式:

- op Rd, Rn, Rm ;Rn+Rm -> Rd
- op Rd, Rn, #expr3 ;Rn + #expr3 -> Rd
- op Rd, #expr8 ; Rd + #expr8 -> Rd

- 例子:

- ADD R2, R3, R4 ;R3+R4 -> R2
- SUB R1, R2, #33 ;R2 - 33 -> R1
- ADD R5, #244 ;R5+244 -> R5

### – 2、高寄存器（R8-R15）或低寄存器（R0-R7）的ADD（加法）指令

- 指令格式:

- op Rd, Rm ;Rd+Rm -> Rd

- 如果Rd和Rm都是低寄存器（R0-R7）时，汇编时，将其翻译为:

- op Rd, Rd, Rm ; Rd+Rm -> Rd

- 例子:

- ADD R11, R3 ;R11+R3 -> R11
- ADD R2, R3 ;R2 + R3 -> R2
- ADD R3, R12 ;R3+R12 -> R3



### – 3、SP（堆栈指针）的**ADD**（加法）和**SUB**（减法）指令

- 指令格式:

- ADD SP, #expr
- SUB SP, #expr

- 例子:

- ADD SP, #256
- SUB SP, #vc + 8

### – 4、PC或SP相关的**ADD**（加法）指令

- 指令格式:

- ADD Rd, Rp, #expr
  - » Rd: R0-R7
  - » Rp: SP或PC

- 例子:

- ADD R5, sp, #64
- ADD R3, pc, #980

### – 5、**ADC**（带进位的加法）、**SBC**（带进位的减法）和**MUL**（乘法）指令

- 指令格式:

- op Rd, Rm

- 例子

- ADC R3, R5
- SBC R5, R6

## • 2.5.4 Thumb内存访问指令

### – 1、立即数偏移的LDR（取数）和STR（存数）指令

#### • 指令格式：

- op Rd,[Rn, #immed\_5\*4]
- opH Rd,[Rn, #immed\_5\*2]
- opB Rd,[Rn, #immed\_5\*1]
  - » H: 半字（16位）
  - » B: 无符号字节（8位）

#### • 例子：

- LDR R4, [R3, #0]
- STRB R2, [R2, #33]
- LDRH R6, [R3, #2]

## – 2、寄存器偏移的**LDR**（取数）和**STR**（存数）指令

- 8中形式:

- (1) **LDR**: 取数（4个字节）指令
  - (2) **STR**: 存数（4个字节）指令
  - (3) **LDRH**: 取数（2个字节无符号数）指令
  - (4) **LDRSH**: 取数（2个字节带符号数）指令
  - (5) **STRH**: 存数（2个字节）指令
  - (6) **LDRB**: 取数（1个字节无符号数）指令
  - (7) **LDRSB**: 取数（1个字节带符号数）指令
  - (8) **STRB**: 存数（1个字节）指令
- » B: 字节
  - » H: 半字（2个字节）
  - » S: 带符号

- 指令格式: **op Rd,[Rn, Rm]**

- 例子:

- **LDR** R2, [R3, R4]
- **LDRSH** R0, [R0, R7]

### – 3、PC或SP相关的**LDR**（取数）和**STR**（存数）指令

- 指令格式:

- LDR Rd,[PC, #immed\_8\*4]
- LDR Rd, label
- LDR Rd,[SP, #immed\_8\*4]
- STR Rd,[SP, #immed\_8\*4]

- 例子:

- LDR R3,[PC, #1016]
- LDR R4, next
- STR R3,[SP, #24]

## – 4、**PUSH**（压栈）和**POP**（出栈）指令

- 指令格式：

- **PUSH {reglist}**
- **POP {reglist}**
- **PUSH {reglist, LR}**
- **POP {reglist, pc}**
  - » **LR**: **LR**（**R14**），连接寄存器
  - » **SP**: **SP**, **R13**, 堆栈指针
  - » **PC**: **PC**, **R15**, 程序计数器

- 例子：

- **PUSH {R0, R2-R4}**
- **PUSH {R2, LR}**
- **POP {R0-R7, PC}**

- 错误的指令：

- **POP {R2-R8}** ;不能用**R8**（只能用**R0-R7**）
- **PUSH { }** ;{ }不能空
- **PUSH {R1-R4, PC }** ;不允许将**PC**压栈
- **POP {R1-R4, LR}** ;不允许弹出（出栈）**LR**

## – 5、LDMIA和STMIA指令

- **LDM**: 批量加载（连续从一片存储器读数，送一组寄存器中）
- **STM**: 批量存储（将一组寄存器中的数，连续存入一片存储器中）
- **IA（后增）**: 先完成指令操作，再完成地址递增
- 指令格式:
  - **op Rn!, {reglist}**
    - » **!**: 表示自增（自减）
- 例子:
  - **LDMIA R2!, {R0, R3}**
  - **STMIA R0!, {R6,R7}**
- 错误的指令:
  - **LDMIA R3!, {R4, R8}** ;不允许使用**R8**（只能用**R0-R7**）
  - **STMIA R5!, {R1-R6}** ;从**R5**开始存入的值不可预计

## • 2.5.5 Thumb软中断和断点指令

- 教材P53写成“断电”，应该为“断点”
- 1、**SWI**：软中断指令，会引起SWI异常，导致处理器切换到ARM状态
  - 指令格式：
    - SWI imm8
  - 例子：
    - SWI 12
- 2、**BKPT**：断点指令，使ARM处理器进入Debug模式
  - 指令格式：
    - BKPT imm8
  - 例子：
    - BKPT 2\_10110

## • 2.5.6 Thumb伪指令

### – 1、**ADR** Thumb伪指令：读取一个程序的相对地址到寄存器

- 指令格式：ADR register, expr

- 例子：

```
ADR R4, txampl; => ADD R4, PC, #nm  
; code  
ALIGN  
Txampl DCW 0,0,0,0
```

### – 2、**LDR** Thumb伪指令：读取32位的常量到低寄存器（R0-R7）中

- 指令格式：LDR register, =[expr | label-exp]

- 例子：

- LDR R1, =0xfff
- LDR R2, =labelname



# 补充：ARM汇编语言程序设计案例

- 例1：20的阶乘

- $20 \times 19 \times 18 \times \dots \times 2 \times 1 = 2,432,902,008,176,640,000$

- 用ARM汇编语言设计程序实现求 20!（20的阶乘），并将其64位的结果放在[R9:R8]中（R9中放置高32位，R8中放置低32位）

$$2^{64} = 18,446,744,073,709,551,616$$

$$-9,223,372,036,854,775,808 \sim +9,223,372,036,854,775,807$$

- 程序设计思路：64位结果的乘法指令通过两个32位的寄存器相乘，可以得到64位的结果，在每次循环相乘中，我们可以将存放64位结果两个32位寄存器分别与递增量相乘，最后将得到的高32位结果相加。

— 程序代码如下：

伪操作

**.global \_start**

@声明全局变量 “\_start”

**.text**

@代码段

**\_start:**

**MOV R8, #20**

@低32位初始化为20

**MOV R9, #0**

@高32位初始化为0 [R9:R8]=20

**SUB R0, R8, #1**

@初始化计数器 R0=19

**Loop:**

**MOV R1, R9**

@暂存高位值

**UMULL R8, R9, R0, R8**

@[R9:R8]=R0\*R8 乘法指令

**MLA R9, R1, R0, R9**

@R9=R1\*R0+R9 乘加指令

**SUBS R0, R0, #1**

@计数器递减

**BNE Loop**

@计数器不为0，继续循环

**Stop:**

**B Stop**

伪操作

**.end**

@源文件结束（程序结束）

- 例2：设计一段程序完成数据块的复制，数据从源数据区复制到目标数据区，要求以4个字为单位进行复制，最后所剩不到4个字的数据，以字为单位进行复制。

– 程序代码如下：

伪操作

```
.global _start  
equ NUM, 18  
.text  
.arm  
_start:
```

```
@声明全局变量 “_start”  
@设置要拷贝的字数  
@代码段  
@ARM程序
```

NUM = 18

LDR R0, =SRC	@R0指向源数据区起始地址
LDR R1, =DST	@R1指向目的数据区起始地址
MOV R2, #NUM	@ R2存放待复制数据量大小， 以字为单位
MOV SP, #0X9000	@堆栈指针指向0X9000， 堆栈增长模式由装载指令的 类型域确定
MOV R3, R2, LSR #2	@ 将R2中值除以4后的结果存放在R3， R3中值表示NUM中有多少个4字单元
BEQ COPY_WORDS	@ 若Z=1(R3=0，数据少于1个4字单元)， 则跳转到COPY_WORDS处， 运行少于4字单元数据处理程序
STMFD SP!, {R5-R8}	@保存R5-R8的内容到堆栈，并更新栈指针， FD:满递减堆栈，由此可知堆栈长向

批量存数指令

## COPY\_4WORD:

LDMIA R0!, {R5-R8}

批量取数指令

STMIA R1!, {R5-R8}

批量存数指令

SUBS R3, R3, #1

BNE COPY\_4WORD

LDMFD SP!, {R5-R8}

批量取数指令

@从R0所指的源数据区装载4个字数据到R5-R8中，  
每次装载1个字后R0中地址加1，  
最后更新R0中地址

@将R5-R8的4个字数据存入R1所指的目的地数据区，  
每次装载1个字后R1中地址加1，  
最后更新R1中地址

@每复制一次，则R3=R3-1，  
表示已经复制了1个4字单元，结果影响CPSR

@若CPSR的Z=0(即运算结果R3不等于0)，  
跳转到COPY\_4WORD，  
继续复制下一个4字单元数据

@将堆栈内容恢复到R5-R8中，并更新堆栈指针，  
此时整4字单元数据已经复制完成，  
且出栈模式应和入栈模式一样

**COPY\_WORDS:**

**ANDS R2,R2, #3**

**@得到NUM除以4后余数，**

**即未满4字单元数据的字数(1个字=4个字节)**

**BEQ STOP**

**@若R2=0(NUM有整数个4字单元)，**

**则停止复制**

**COPY\_WORD:**

**LDR R3, [R0], #4**

**@将R0所指源数据区的4个字节(1个字)**

**数据装载至R3，然后R0=R0+4**

**STR R3, [R1], #4**

**@将R3中4个字节(1个字)数据存到**

**R1所指目的数据区，然后R1=R1+4**

**SUBS R2, R2, #1**

**@数据传输控制计数器减1(其总是小于4)，**

**成功复制一个字数据**

**BNE COPY\_WORD**

**@ 若R2不等于0，则转到WORDCOPY，**

**继续复制下一个字数据**

第一次复制: 1,2,3,4

第二次复制: 5,6,7,8

第三次复制: 9, 0xa,0xb,0xc

第四次复制: 0xd,0xe,0xf,0x10

第五次复制: 0x11

第六次复制: 0x12

STOP:

B STOP

伪操作

.ltorg

@声明一个数据缓冲池的开始,  
一般在代码的最后面

SRC:

.long 1,2,3,4,5,6,7,8,9,0xa,0xb,0xc,  
0xd,0xe,0xf,0x10,0x11,0x12 @18个源数据

DST:

.long 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 @18个目的数据

伪操作

.end

@程序结束

# 本章小结

- **ARM处理器的两种状态：**

- ARM状态（32位），Thumb状态（16位）
- ARM指令集（32位），Thumb指令集（16位），Thumb指令集是ARM指令集的子集
- ARM指令必须在ARM状态下执行，Thumb指令必须在Thumb状态下执行
- ARM处理器可以在两种状态（ARM状态、Thumb状态）下进行切换；ARM子程序（ARM指令集编写的程序）和Thumb子程序（Thumb指令集编写的程序）之间可以进行相互调用

- **ARM处理器的寄存器：**

- 37个32位的寄存器：31个通用寄存器，6个专用状态寄存器
- CPSR, SPSR, SP (R13), LR (R14), PC (R15)

- **ARM处理器的运行模式：**

- ① 系统模式（SYS）
- ② 用户模式（USR）
- ③ 快速中断模式（FIQ）
- ④ 管理模式（SVC）
- ⑤ 数据访问终止模式（ABT）
- ⑥ 外部中断模式（IRQ）
- ⑦ 未定义指令终止模式（UND，未定义模式）

- 特权模式：除用户模式外的6种模式
- 异常模式：除系统模式、用户模式外的5种模式

**ARM处理器的寄存器（37个32位的寄存器）：**

- **31个通用寄存器**

- R0、R1、R2、R3、R4、R5、R6、R7 8个
- R8、R8\_fiq 2个
- R9、R9\_fiq 2个
- R10、R10\_fiq 2个
- R11、R11\_fiq 2个
- R12、R12\_fiq 2个
- R13、R13\_svc、R13\_abt、R13\_und、R13\_irq、R13\_fiq 6个
- R14、R14\_svc、R14\_abt、R14\_und、R14\_irq、R14\_fiq 6个
- R15 (PC) 1个
- R13: SP, 堆栈指针, Stack Pointer
- R14: LR, 链接寄存器, Link Register
- R15: PC, 程序计数器, Program Counter

- **6个专用状态寄存器**

- CPSR 1个
- SPSR\_svc、SPSR\_abt、SPSR\_und、SPSR\_irq、SPSR\_fiq 5个
- CPSR: Current Program Status Register, 当前程序状态寄存器
- SPSR: Saved Program Status Register, 程序状态保存寄存器



- **ARM指令的寻址方式:**

- ① 立即寻址
- ② 寄存器寻址
- ③ 寄存器偏移寻址
- ④ 寄存器间接寻址
- ⑤ 基址变址寻址
- ⑥ 多寄存器寻址
- ⑦ 堆栈寻址
- ⑧ 相对寻址
- ⑨ 块复制寻址

- **ARM指令的类型:**

- ① 跳转指令
- ② 通用数据处理指令
- ③ 乘法指令
- ④ Load/Store内存访问指令
- ⑤ ARM协处理器指令
- ⑥ 杂项指令
- ⑦ 饱和算术指令
- ⑧ ARM伪指令

- **Thumb指令的类型:**

- ① Thumb跳转指令
- ② Thumb通用数据处理指令
- ③ Thumb算术指令
- ④ Thumb内存访问指令
- ⑤ Thumb软中断和断点指令
- ⑥ Thumb伪指令

- **ARM指令的格式: <opcode> {<cond>} {S} <Rd>, <Rn> {, <shift\_op2>}**

- <>内的项是必须的, {}内的项是可选的
- opcode: 指令助记符(操作码), 如LDR, STR等
- **cond**: 执行条件(条件码), 如EQ, NE等
- S: 可选后缀, 加S时影响CPSR中的条件码标志位, 不加S时则不影响
- Rd: 目标寄存器
- Rn: 第1个源操作数的寄存器
- op2: 第2个源操作数
- **shift**: 位移操作

- 条件码（（<cond>）：CPSR[31:28] N Z C V）的16种组合：

AMR指令条件码	助记符	描述	CPSR条件码标志位的值
0000	EQ	相等，运行结果为0	Z置位
0001	NE	不相等，运行结果不为0	Z清零
0010	CS/HS	无符号数大于等于	C置位
0011	CC/LO	无符号数小于	C清零
0100	MI	负数	N置位
0101	PL	非负数	N清零
0110	VS	上溢出	V置位
0111	VC	没有上溢出	V清零
1000	HI	无符号数大于	C置位且Z清零
1001	LS	无符号数小于等于	C清零且Z置位
1010	GE	带符号数大于等于	N=V
1011	LT	带符号数小于	N!=V
1100	GT	带符号数大于	Z清零且N=V
1101	LE	带符号数小于等于	Z置位且N!=V
1110	AL	无条件执行	
1111	系统保留		

- 位移操作<shift\_op2>的11种形式：

语法	含义
#<immediate>	立即数寻址
<Rm>	寄存器寻址
<Rm>, LSL #<shift_imm>	立即数逻辑左移
<Rm>, LSL <Rs>	寄存器逻辑左移
<Rm>, LSR #<shift_imm>	立即数逻辑右移
<Rm>, LSR <Rs>	寄存器逻辑右移
<Rm>, ASR #<shift_imm>	立即数算术右移
<Rm>, ASR <Rs>	寄存器算术右移
<Rm>, ROR #<shift_imm>	立即数循环右移
<Rm>, ROR <Rs>	寄存器循环右移
<Rm>, RRX	寄存器扩展循环右移

## 位移操作的5种方式：

- ① LSL：逻辑左移
- ② LSR：逻辑右移
- ③ ASR：算术右移
- ④ ROR：循环右移
- ⑤ RRX：带扩展的循环右移（加上CPSR的C位一起循环右移）

# 进一步探索

- 阅读ARM汇编代码，如Boot Loader（引导程序）的相关代码，结合第6章（Boot Loader技术）的内容，深入了解Boot Loader的工作机制
- 实验箱的Boot Loader（**u-boot2017.03**）代码中的ARM汇编语言程序位于Ubuntu的：`/home/uptech/fsl-6dl-source/u-boot2017.03/arch/arm/cpu/armv7/start.S`

```

/*
 * armboot - Startup Code for OMAP3530/ARM Cortex CPU-core
 *
 * Copyright (c) 2004 Texas Instruments <r-woodruff2@ti.com>
 *
 * Copyright (c) 2001 Marius Gröger <mag@sysgo.de>
 * Copyright (c) 2002 Alex Züpke <azu@sysgo.de>
 * Copyright (c) 2002 Gary Jennejohn <garyj@denx.de>
 * Copyright (c) 2003 Richard Woodruff <r-woodruff2@ti.com>
 * Copyright (c) 2003 Kshitij <kshitij@ti.com>
 * Copyright (c) 2006-2008 Syed Mohammed Khasim <x0khasim@ti.com>
 * Copyright (C) 2016 Freescale Semiconductor, Inc.
 *
 * SPDX-License-Identifier: GPL-2.0+
 */

```

/home/uptech/fsl-6dl-source/u-boot2017.03/arch/arm/cpu/armv7/start.S

```

#include <asm-offsets.h>
#include <config.h>
#include <asm/system.h>
#include <linux/linkage.h>
#include <asm/armv7.h>

```

```

/*****
 *
 * Startup Code (reset vector)
 *
 * Do important init only if we don't start from memory!
 * Setup memory and board specific bits prior to relocation.
 * Relocate armboot to ram. Setup stack.
 *
 *****/

```

```

        .globl reset
        .globl save_boot_params_ret
#ifdef CONFIG_ARMV7_LPAE
        .global switch_to_hypervisor_ret
#endif

```

```

reset:
        /* Allow the board to save important registers */
        b        save_boot_params

```

```

save_boot_params_ret:
#ifdef CONFIG_ARMV7_LPAE

```

```

/*
 * check for Hypervisor support
 */

```

```

        mrc        p15, 0, r0, c0, c1, 1        @ read ID_PFR1
        and        r0, r0, #CPUID_ARM_VIRT_MASK    @ mask virtualization bits
        cmp        r0, #(1 << CPUID_ARM_VIRT_SHIFT)
        beq        switch_to_hypervisor

```

```

switch_to_hypervisor_ret:
#endif

```

```

/*
 * disable interrupts (FIQ and IRQ), also set the cpu to SVC32 mode,
 * except if in HYP mode already
 */

```

```

        mrs        r0, cpsr
        and        r1, r0, #0x1f                @ mask mode bits
        teq        r1, #0x1a                    @ test for HYP mode
        hcneg     r0, r0, #0x1f                @ clear all mode bits

```

```

/*
 * Invalidate L1 I/D
 */
mov     r0, #0                @ set up for MCR
mcr     p15, 0, r0, c8, c7, 0 @ invalidate TLBs
mcr     p15, 0, r0, c7, c5, 0 @ invalidate icache
mcr     p15, 0, r0, c7, c5, 6 @ invalidate BP array
mcr     p15, 0, r0, c7, c10, 4 @ DSB
mcr     p15, 0, r0, c7, c5, 4 @ ISB

/*
 * disable MMU stuff and caches
 */
mrc     p15, 0, r0, c1, c0, 0
bic     r0, r0, #0x00002000    @ clear bits 13 (--V-)
bic     r0, r0, #0x00000007    @ clear bits 2:0 (-CAM)
orr     r0, r0, #0x00000002    @ set bit 1 (--A-) Align
orr     r0, r0, #0x00000800    @ set bit 11 (Z---) BTB
#ifdef CONFIG_SYS_ICACHE_OFF
bic     r0, r0, #0x00001000    @ clear bit 12 (I) I-cache
#else
orr     r0, r0, #0x00001000    @ set bit 12 (I) I-cache
#endif
mcr     p15, 0, r0, c1, c0, 0

#ifdef CONFIG_ARM_ERRATA_716044
mrc     p15, 0, r0, c1, c0, 0 @ read system control register
orr     r0, r0, #1 << 11      @ set bit #11
mcr     p15, 0, r0, c1, c0, 0 @ write system control register
#endif

#if (defined(CONFIG_ARM_ERRATA_742230) || defined(CONFIG_ARM_ERRATA_794072))
mrc     p15, 0, r0, c15, c0, 1 @ read diagnostic register
orr     r0, r0, #1 << 4        @ set bit #4
mcr     p15, 0, r0, c15, c0, 1 @ write diagnostic register
#endif

#ifdef CONFIG_ARM_ERRATA_743622
mrc     p15, 0, r0, c15, c0, 1 @ read diagnostic register
orr     r0, r0, #1 << 6        @ set bit #6
mcr     p15, 0, r0, c15, c0, 1 @ write diagnostic register
#endif

#ifdef CONFIG_ARM_ERRATA_751472
mrc     p15, 0, r0, c15, c0, 1 @ read diagnostic register
orr     r0, r0, #1 << 11      @ set bit #11
mcr     p15, 0, r0, c15, c0, 1 @ write diagnostic register
#endif

#ifdef CONFIG_ARM_ERRATA_761320
mrc     p15, 0, r0, c15, c0, 1 @ read diagnostic register
orr     r0, r0, #1 << 21      @ set bit #21
mcr     p15, 0, r0, c15, c0, 1 @ write diagnostic register
#endif

#ifdef CONFIG_ARM_ERRATA_845369
mrc     p15, 0, r0, c15, c0, 1 @ read diagnostic register
orr     r0, r0, #1 << 22      @ set bit #22
mcr     p15, 0, r0, c15, c0, 1 @ write diagnostic register
#endif

mov     r5, lr                @ Store my Caller

```

/home/uptech/fsl-6dl-source/u-boot2017.03/arch/arm/cpu/armv7/start.S

**Thanks**