

嵌入式系统设计与应用

第二章 ARM指令系统（1）

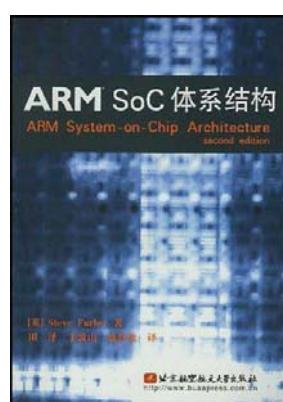
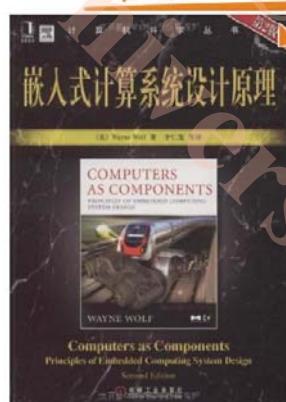
西安交通大学电信学院
孙宏滨



教学用书与参考书

- Wayne Wolf, 《嵌入式计算系统设计原理》, 机械工业出版社, 2002年
- Steve Furber, 《ARM SoC体系结构》, 北京航空航天大学出版社

在线试读!

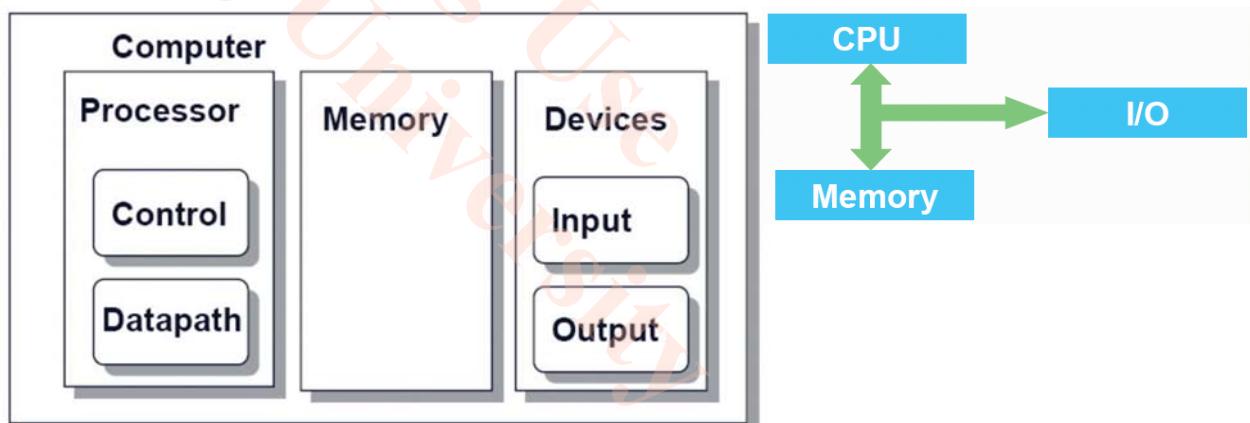




什么是处理器？

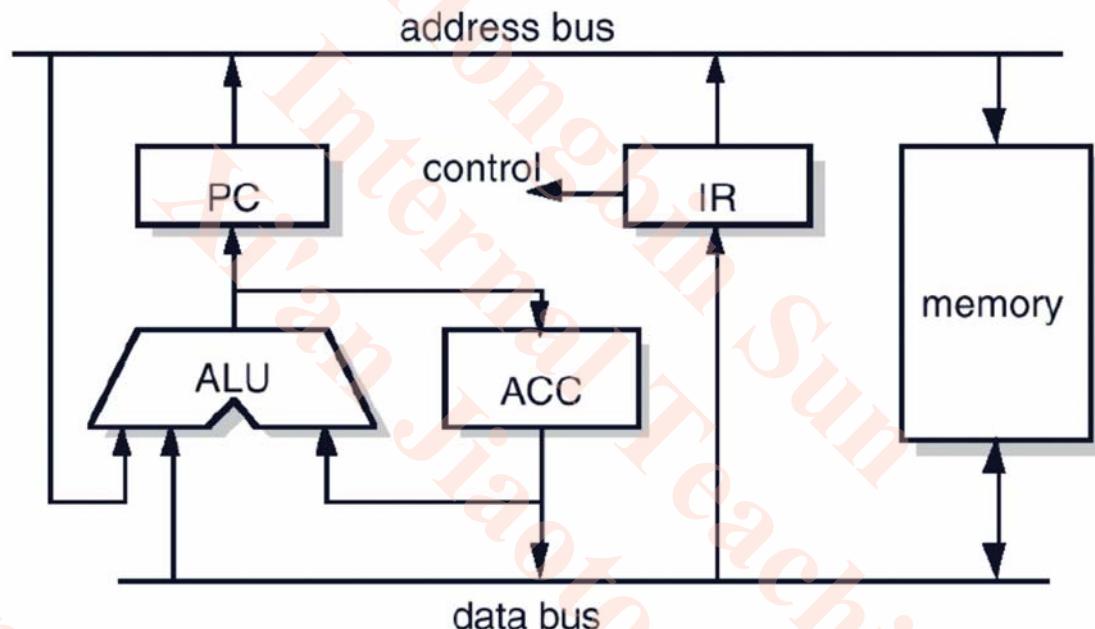
什么是处理器？

- 通用处理器是一个执行存储器中指令的有限状态机（**FSM**）。
- 系统的状态由存储器和处理器寄存器的数据定义。每条指令都规定了总状态变化的特定方式，还指定随后执行哪条指令。





MU0 – 一个简单的处理器



MU0 的基本部件

- 程序计数器（PC）寄存器：保存当前指令地址
- 累加器（ACC）寄存器：保存正在处理的数据
- 算术逻辑单元（ALU）：对二进制数进行操作
- 指令寄存器（IR）：保存当前执行的指令
- 指令译码器和控制逻辑：根据指令控制上述部件产生所需的结果。





MU0指令

- MU0是具有12位地址空间的16位机。它的存储器可组织为4096个独立寻址的16位字长的存储单元。
- 如下图所示，每条指令前4位是操作码（opcode），后12位是地址域（S），表示操作数的存储地址。
- 默认处理器从存储器的0地址开始顺序读取指令，直到执行到修改PC的指令为止。



MU0指令集

指令	操作码	效果
Instruction	Opcode	Effect
LDA S	0000	ACC := mem[S]
STO S	0001	mem[S] := ACC
ADD S	0010	ACC := ACC + mem[S]
SUB S	0011	ACC := ACC - mem[S]
JMP S	0100	PC := S
JGE S	0101	if ACC > 0, PC := S
JNE S	0110	if ACC ≠ 0, PC := S
STP	0111	stop

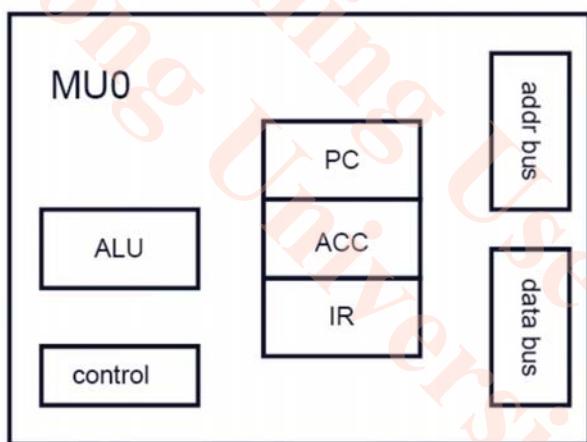


一个小程序

- 两个数求和
 - 假设两个数存储在两个连续的存储单元中，地址分别为**2E**和**2F**。
 - 假设两者之和存储到地址空间**30**。
 - 我们需要将第一个数装载到ACC，和另一个数相加后，再写回存储器。

LDA 02E	002E
ADD 02F	202F
STO 030	1030
STP	7???
助记符	机器码

初始状态

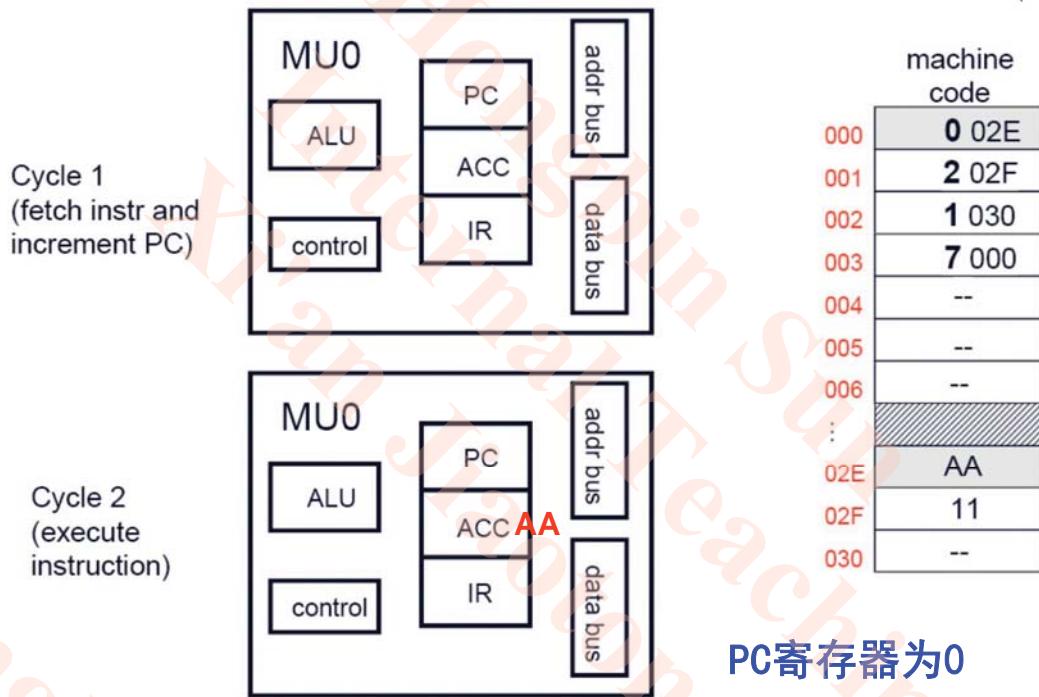


初始状态，PC寄存器为0

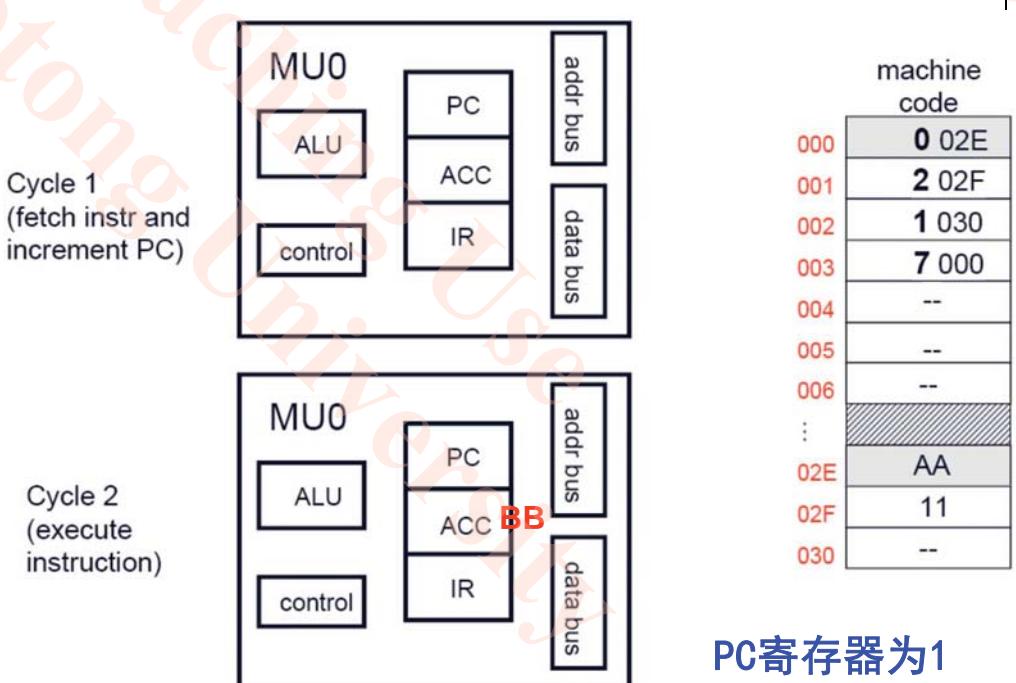
	mnemonic	machine code
000	LDA 02E	0 02E
001	ADD 02F	2 02F
002	STO 030	1 030
003	STP	7 000
004	--	--
005	--	--
006	--	--
:		
02E	AA	AA
02F	11	11
030	--	--



第一条指令： LDA 02E

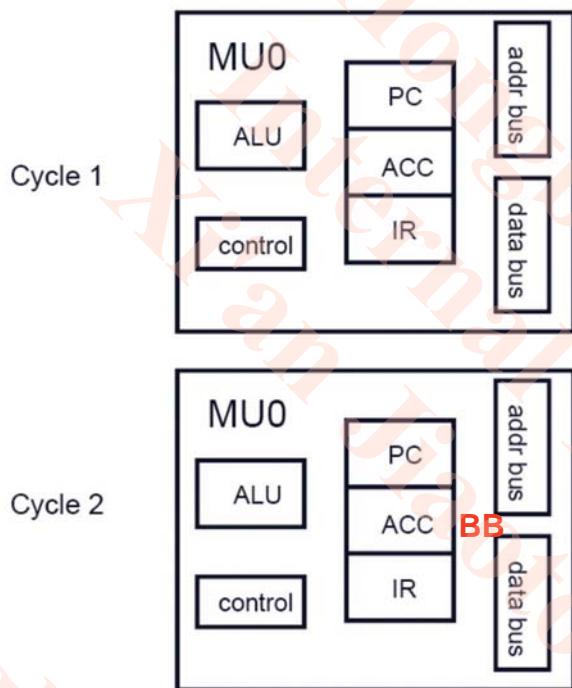


第二条指令： ADD 02F





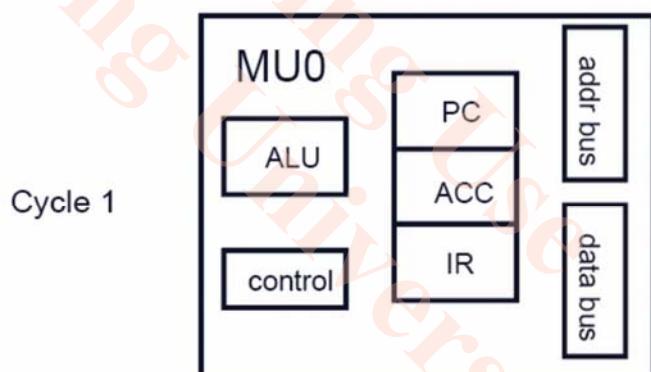
第三条指令： STO 030



machine code
0 02E
0 02F
1 030
7 000
--
--
--
...
02E AA
02F 11
030 BB

PC寄存器为2

第四条指令： STP



PC寄存器为3

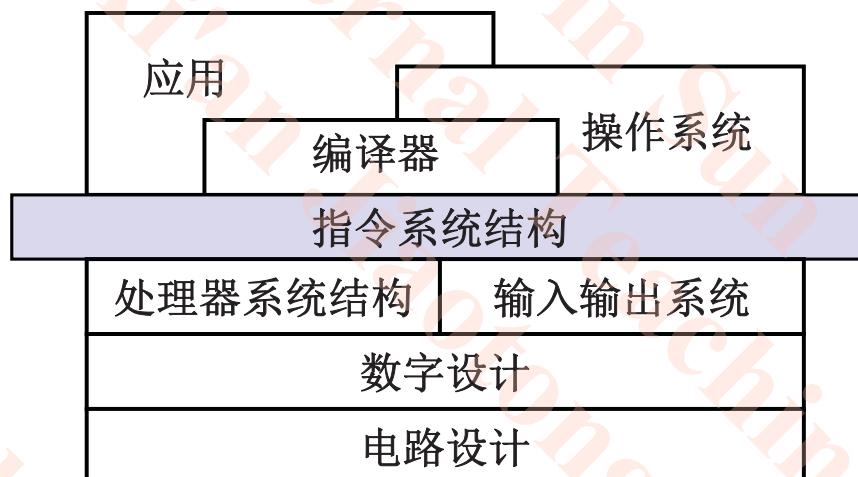
machine code
0 02E
0 02F
1 030
7 000
--
--
--
...
02E AA
02F 11
030 BB



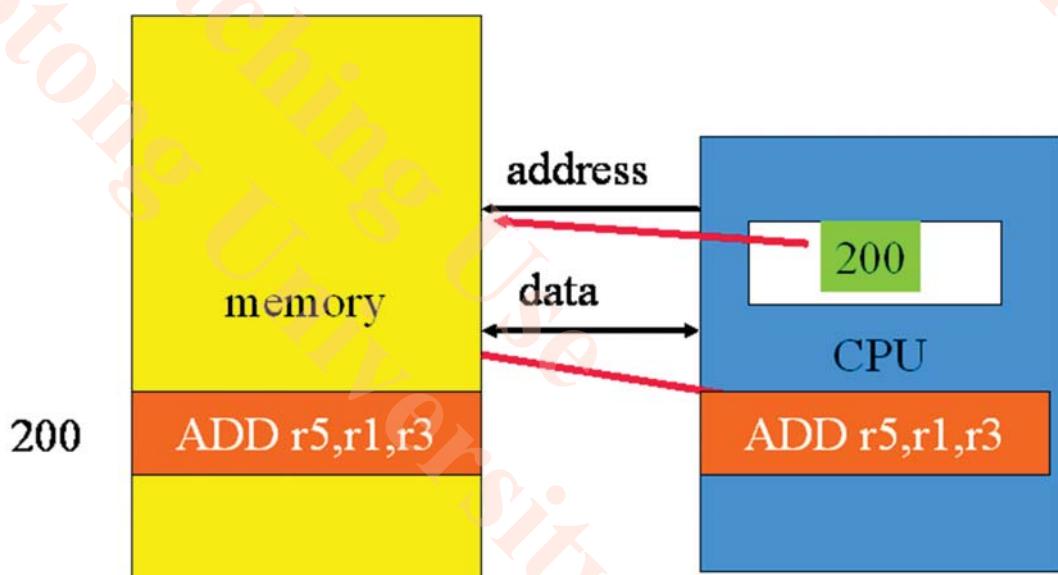


指令系统 (Instruction Set)

- 指令系统或指令系统结构 (Instruction Set Architecture, ISA) 是计算机体系结构中与程序设计相关的一部分，包括数据类型、指令、寄存器、寻址模式和存储结构等。
- 指令系统是编程人员与硬件的接口。

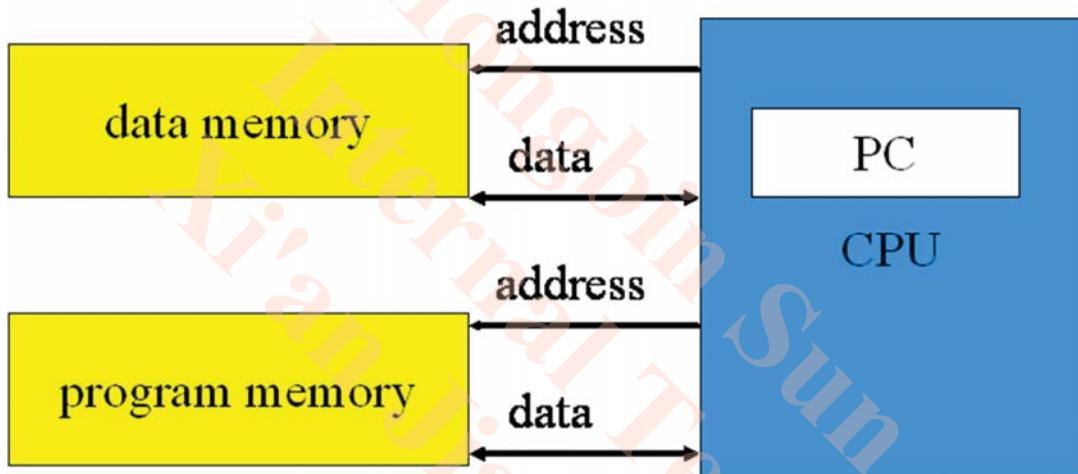


冯·诺伊曼体系结构





哈佛体系结构



独立的程序和数据存储器避免了存储端口的竞争，为数字信号处理提供了较高的性能，因此大部分**DSP**都采用哈佛体系结构。



冯·诺依曼 vs. 哈佛体系结构

- 冯·诺依曼体系结构：**X86、ARM7、MIPS**
 - 指令和数据空间统一编址，存储分配灵活，可以最大限度的利用资源。
 - 适合于复杂的、任务可扩展的嵌入式系统。比如：在通用计算或智能终端中，应用软件的多样性使得计算机要不断地变化所执行的代码的内容，并且频繁地对数据与代码占有的存储器进行重新分配。
- 哈佛体系结构：**8051、MSP430、ARM9、TI DSP**
 - 指令和数据空间独立编址，避免了存储端口的竞争。
 - 适合于那些程序固化、任务相对简单的嵌入式系统。
 - 不能编写自修改程序，**但指令与数据空间分开，安全。**



CISC vs. RISC

- **CISC** (**Complex Instruction Set Computer**) 复杂指令集计算机，诞生于**20世纪70年代**，是**1980年前**指令集设计的主要趋势。
 - 背景：处理器比主存储器速度快。
 - 采用大量变长形式的不同指令，指令执行周期差别较大。
 - Intel的x86系列CPU是CISC的代表。
- **RISC** (**Reduced Instruction Set Computer**) 精简指令集计算机，诞生于**1980年**--指令集日益复杂的时刻。



RISC体系结构

- **RISC**: 固定的(**32bit**)指令长度，指令类型很少。
- **CISC**: 变长指令集，指令类型很多。
- **RISC**: **Load-Store**结构，数据处理指令只访问寄存器，与访问存储器的指令是分开的。
- **CISC**: 一般允许将存储器中的数据作为数据处理指令的操作数。
- **RISC**: 大的通用寄存器堆，以使**Load-Store**有效工作。
- **CISC**: 寄存器一般较少，且多数都有特殊用途。
- **RISC**: 流水线执行、单周期执行。



RISC的优点与缺点

- **芯片面积小**: RISC CPU设计简单，电路规模小。
- **开发时间短**: 简单的处理器占用设计力量小，设计费用低。
- **性能高**: 流水线处理、高时钟频率和单周期执行。
- **代码密度低**: 指令集固定长度，占用更大的存储资源和带宽，造成更高的功耗。
- **RISC不能执行x86代码**: 在一段时间里，由于x86强大的生态环境，RISC主要应用在x86不涉及的领域。



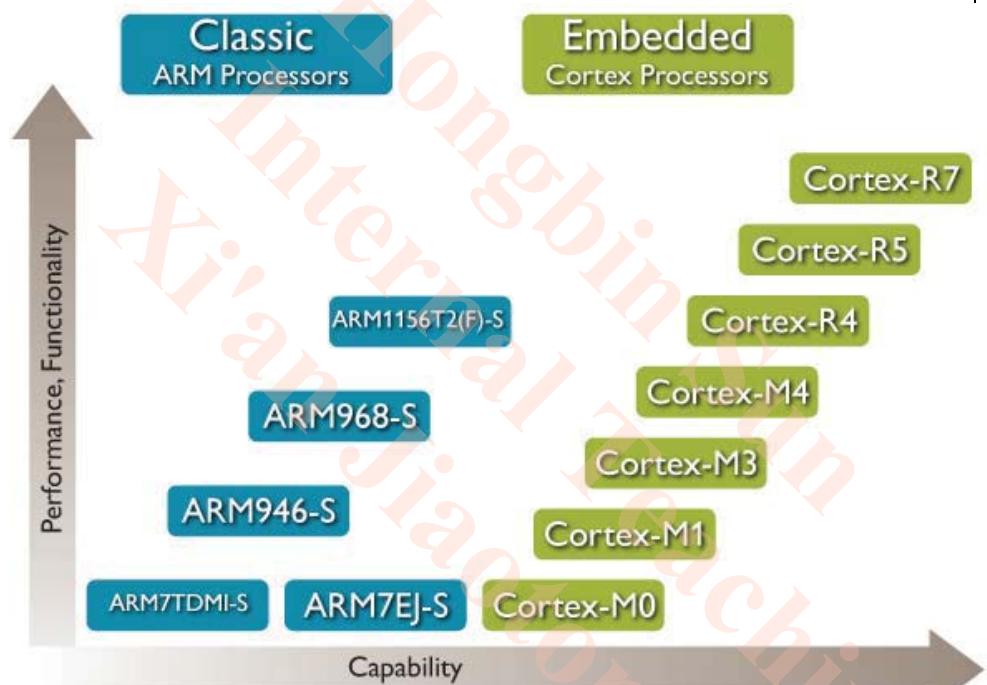
ARM, Advanced RISC Machine

- 1980年前后的一项处理器研究计划（RISC）
 - 斯坦福（Stanford）大学 → MIPS
 - 伯克利（Berkeley）大学 → ARM
- ARM仅有400名雇员的小公司。
- “**简单性**”的原则和理念成就了ARM。

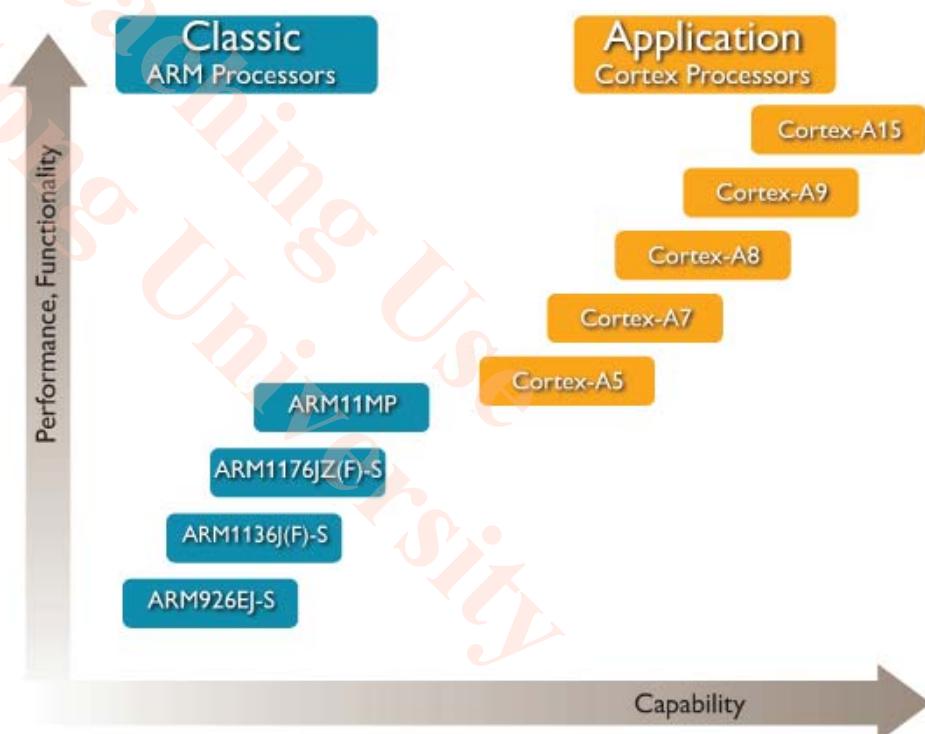




ARM体系结构的发展历程（1）



ARM体系结构的发展历程（2）





ARM处理器的主要特征

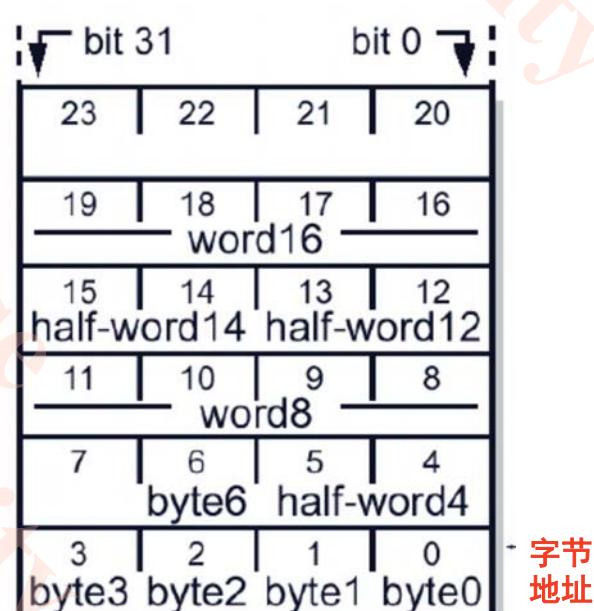
- 采用**Load-Store**结构
- 固定长度（**32bit**）指令
- 三操作数指令格式
 - ◆ 3-operand instruction format (used by ARM processor)

f bits	n bits	n bits	n bits
function	op 1 addr.	op 2 addr.	dest. addr.

- 条件执行所有指令
- 一条指令可以装载或存储多个寄存器
- **ALU**操作支持单周期**n-bit**移位

ARM存储器系统

- 标准的**ARM**字是**32**位长
- 每个字分为**4**个**8**位字节，或**2**个**16**位半字
- 字总是以**4**字节边界对准，半字则以偶数字节的边界对准。
- 允许按字节寻址，每个地址对应一个字节，而不是一个字。

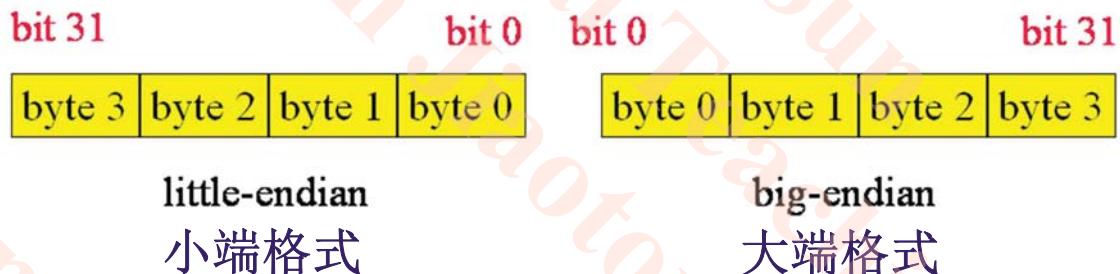


字节
地址



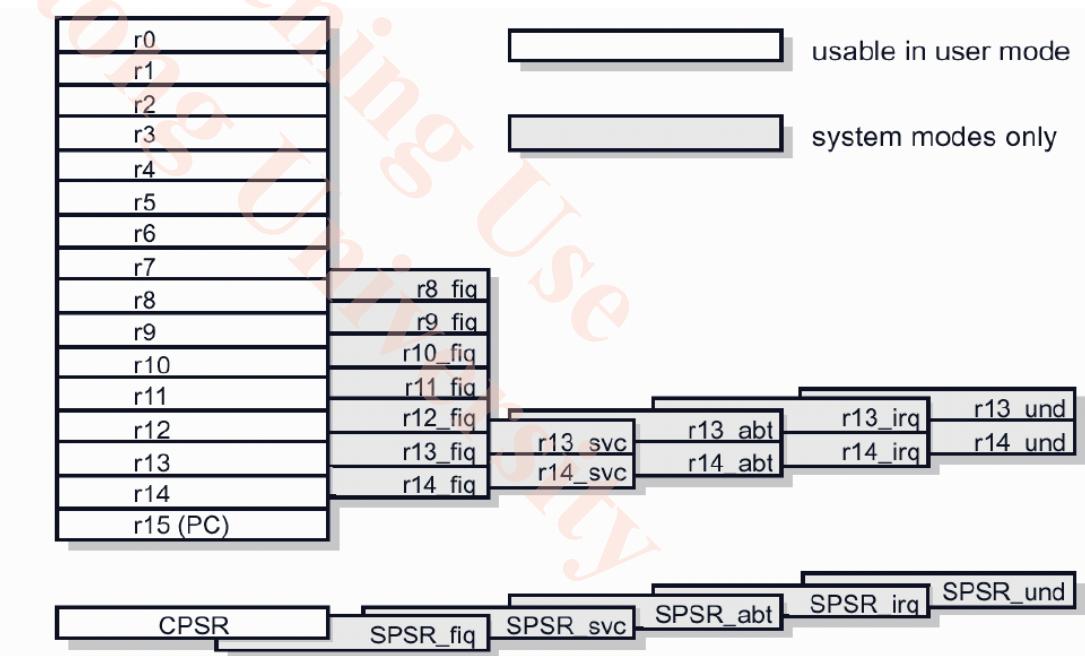
大端（低序）与小端（高序）

- 小端格式：在字的最低位中存放最低位字节（**ARM的常用格式**）
- 大端格式：在字的最高位中存放最低位字节。
- **ARM既可以配置为小端格式，也可以配置为大端格式。**



ARM编程模型 Programming Model

- 编程模型是所有用户可见寄存器的集合。





ARM编程模型

- 用户级程序：15个通用32位寄存器、程序计数器（r15）和当前状态寄存器（CPSR, Current Program Status Register）。
- CPSR在用户级编程时用于存储条件码（自动设置）。
 - N: 负数。改变标志位的最后一次ALU操作的结果为负。
 - Z: 零。改变标志位的最后一次ALU操作的结果为0。
 - C: 进位。改变标志位的最后一次ALU操作产生进位输出。
 - V: 溢出位。改变标志位的最后一次算术ALU操作产生符号位的溢出
 - IF: 中断使能。
 - T: 指令集选择。
 - Mode: 控制处理器的模式。

31	28 27	unused	8 7 6 5 4	0
N Z C V			IF T	mode



ARM编程模型

31	28 27	unused	8 7 6 5 4	0
N Z C V			IF T	mode

- N:** 当用两个补码表示的带符号数进行运算时，**N=1**表示运算的结果为负数；**N=0**表示运算的结果为正数或零。
- Z:** **Z=1**表示运算结果为零；**Z=0**表示运算结果为非零。
- C:** 可以有4种方法设置C的值
 - 加法运算（包括CMN）：产生进位**C=1**，否则**C=0**。
 - 减法运算（包括CMP）：**产生借位C=0, 否则C=1**。
 - 包含RRX的非加/减运算指令：C为移出值的最后一位。
 - 对于其他非加减运算指令，C值通常保持不变。
- V:** 当操作数和运算结果为二进制的补码表示的带符号数时，**V=1**表示符号位溢出。对于其他非加减运算，**V**的值通常保持不变。



示例：ARM中状态位的计算

- ARM字长为32位。在C语言表示中，一个十六进制的数是以0x开始的，如0xffffffff，它在一个32位字中是-1的二进制补码表示。
- $-1+1=0$; $0xffffffff+0x1=0x0$; NZCV=0110.
- $0-1=-1$; $0x0-0x1=0xffffffff$; NZCV=1000.
- $2^{31}-1+1 = -2^{31}$: $0x7fffffff+0x1=0x80000000$; NZCV = 1001.



ARM的Load-Store体系结构

- 指令集仅能处理寄存器中的值，而且总是将处理结果放回到寄存器中。
- 对存储器的操作仅能将存储器的值拷贝到寄存器中，或将寄存器的值拷贝到存储器中。
- ARM不支持“存储器-存储器”操作。



ARM的Load-Store体系结构

- ARM指令都属于下列三种类型之一：
 - **数据处理指令**: 只能使用或改变寄存器中的值。
 - 加、减、逻辑与、逻辑或
 - **数据传送指令**: 把存储器中的值拷贝到寄存器中，或者相反。
 - Load、Store、存储器和寄存器值交换（系统）
 - **控制流指令**: 使指令执行切换到不同的地址
 - 转移、转移和链接或陷入系统代码

汇编语言基础（1）

```
AREA Example, CODE, READONLY
ENTRY

start
    MOV r0, #15
    MOV r1, #20
    BL firstfunc
    SWI 0x11

firstfunc
    ADD r0, r0, r1
    MOV pc, lr

    END

; name this block of code
; mark first instruction
; to execute

; Set up parameters

; Call subroutine
; terminate
; Subroutine firstfunc
; r0 = r0 + r1
; Return from subroutine
; with result in r0
; mark end of file

label      标记
opcode     操作码
operands   操作数
comment    注释
```



汇编语言基础（2）

- 通常一条汇编语言包括以下几部分

`label <space> opcode <space> operands <space> ; comment`

- 汇编语言特征：

- 一条指令一行
- 给存储单元提供名字的标记从第一列开始
- 指令必须从第二列或能区分标记的地方开始
- 注释跟在指定的注释符后面（ARM使用“;”）直到行尾
- 每个域之间以空格分开

数据处理指令

- 应用于ARM数据处理指令的原则：

- 所有操作数都是32位宽，或来自寄存器，或在指令中定义的立即数；
- 结果也是32位宽，放在一个寄存器中；
- 3个操作数：2个作为输入，1个作为输出。

- 例子：**ADD r0, r1, r2 ; r0:=r1+r2**

- 可以计算无符号整数或有符号整数的2的补码值；
- 可能会产生进位或符号位溢出，但默认忽略；
- 结果寄存器可以和操作数寄存器相同。





数据处理指令 – 算术运算

ADD	r0, r1, r2	; r0 := r1 + r2
ADC	r0, r1, r2	; r0 := r1 + r2 + C
SUB	r0, r1, r2	; r0 := r1 - r2
SBC	r0, r1, r2	; r0 := r1 - r2 + C - 1
RSB	r0, r1, r2	; r0 := r2 - r1
RSC	r0, r1, r2	; r0 := r2 - r1 + C - 1

- **RSB**是反向减指令
- 操作数可以是无符号数，或是有符号的**2进制补码**
- “C”是当前状态寄存器**CPSR**中的进位位
- **ADC, SBC**和**RSC**用于运算超过32位长的数

ADC, SBC和RSC

- 举例：两个**64位数X, Y相加，结果为Z。**
- 每个数需要两个寄存器：
 - 假设X存储在r1:r0, Y在r3:r2，而结果Z在r5:r4。
 - ADDS r4, r0, r2
 - ADC r5, r1, r3
- “S”表明需要在**CPSR**中记录当前操作状态的标志位：**C, V, N**和**Z**。
- 类似的，减法的计算如下：
 - SUBS r4, r0, r2
 - SBC r5, r1, r3



数据处理指令 – 逻辑运算

AND	r0, r1, r2	; r0 := r1 and r2 (bit-by-bit for 32 bits)
ORR	r0, r1, r2	; r0 := r1 or r2
EOR	r0, r1, r2	; r0 := r1 xor r2
BIC	r0, r1, r2	; r0 := r1 and not r2

- **BIC**是按位清零指令。第二操作数的每个‘1’将对第一个操作数对应数据位清零。

r1:	0101 0011 1010 1111 1101 1010 0110 1011
r2:	1111 1111 1111 1111 0000 0000 0000 0000
r0:	0000 0000 0000 0000 1101 1010 0110 1011



数据处理指令 – 寄存器传送

MOV	r0, r2	; r0 := r2
MVN	r0, r2	; r0 := not r2

- 寄存器传送指令不用第一操作数，只是简单地将第二操作数（可能按位取反）传送到目的寄存器。
- **MVN**助记符意为“取反传送”，它是把源寄存器的每一位取反，将得到的值置入结果寄存器。

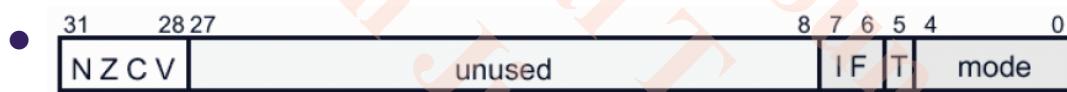
r2:	0101 0011 1010 1111 1101 1010 0110 1011
r0:	1010 1100 0101 0000 0010 0101 1001 0100



数据处理指令 – 比较操作

CMP	r1, r2	; set cc on r1 - r2
CMN	r1, r2	; set cc on r1 + r2
TST	r1, r2	; set cc on r1 and r2
TEQ	r1, r2	; set cc on r1 xor r2

- 如上“减，加，与和异或”的结果不存储于任何寄存器



- ❖ N = 1 if MSB of $(r1 - r2)$ is '1'
❖ Z = 1 if $(r1 - r2) = 0$
❖ C = 1 if carry-out of addition is 1
❖ V = 1 if carry-out of addition is different to carry-in to previous stage

寄存器寻址&立即数寻址

- 寄存器寻址: **ADD r0, r1, r2**
- 立即数寻址: **ADD r3, r3, #1 ;r3:=r3+1**
AND r8, r7, #&ff ;r8:=r7_[7:0]
- 如果希望寄存器和常数相加, 而不是两个寄存器相加, 可以用立即数取代第二操作数(前面加#)。
 - &表示以16进制的形式定义立即数。
- 立即数的范围: 在32位指令字内编码
 - 立即数 = $(0 \rightarrow 255) \times 2^{2n}$, 其中 $0 \leq n \leq 12$



数据传送指令 – 单寄存器L/S指令

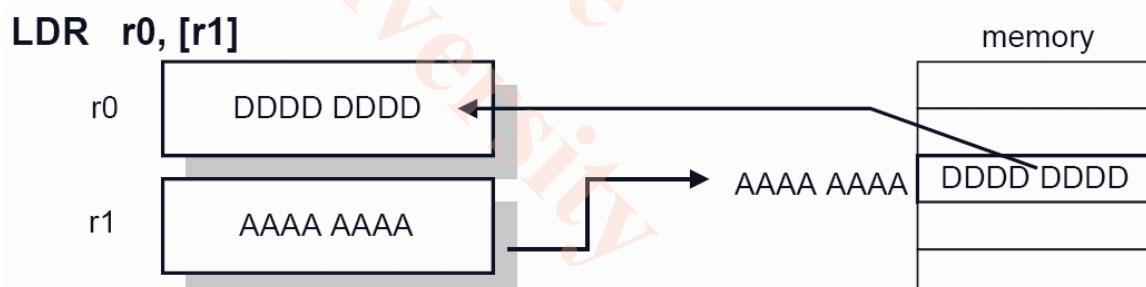
- ARM指令集中有3种基本的数据传送指令
 - 单寄存器**Load/Store**指令
 - 数据项可以是字节、16位半字和32位字。
 - 多寄存器**Load/Store**指令
 - 灵活性较低，但可以使大量数据有效传送。
 - 用于进程的进入和退出，保存和恢复工作寄存器，以及拷贝存储器中一块数据。
 - 单寄存器交换指令
 - 一条指令有效完成**Load**和**Store**操作。
 - 用户级编程很少使用，本课程不作进一步讨论。



寄存器间接寻址

- 寄存器间接寻址：利用一个寄存器的值（基址寄存器）作为存储器地址，或者从该地址取值存放到另一个寄存器，或者反之。

LDR r0, [r1]	; r0 := mem ₃₂ [r1]
STR r0, [r1]	; mem ₃₂ [r1] := r0





数据传送指令 – 初始化地址指针

- 如何使**ARM**寄存器包含特定存储器单元的地址？
- 使用**ADR**伪指令：在汇编源代码中，伪指令就像一般的指令，只是它没有直接对应于特定的**ARM**指令。使用时，汇编器都有一定的规则选择最适当的指令或几条相连指令。**ADR->ADD/SUB**

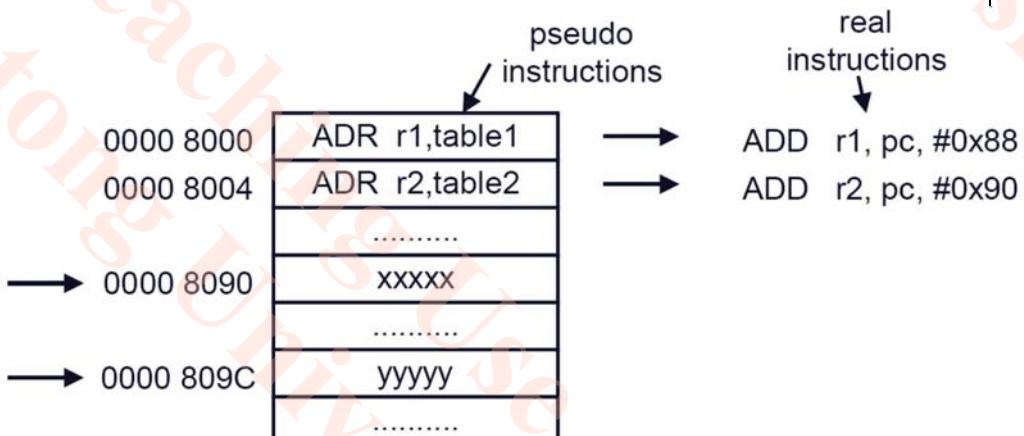
copy	ADR r1, TABLE1 ; r1 points to TABLE1
	ADR r2, TABLE2 ; r2 points to TABLE2
	LDR r0, [r1] ; load first value
	STR r0, [r2] ; and store it in TABLE2

TABLE1

TABLE2
	; <source of data>
	; <destination of data>



数据传送指令 – ADR伪指令



- ADR如何实现？32位地址很难用寄存器计算。
- 程序计数器（PC）r15的内容通常接近所需数据地址。
- ADR r1, table1被转换成PC（r15）加减一个常数。
- 程序计数器相对寻址（PC-relative addressing）



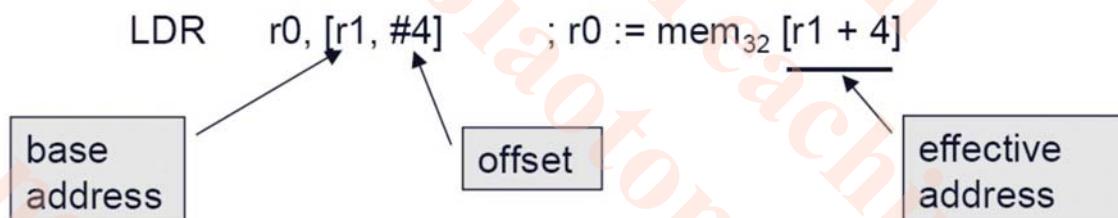
数据传送指令 – 基地址偏移寻址

```

copy      ADR  r1, TABLE1    ; r1 points to TABLE1
          ADR  r2, TABLE2    ; r2 points to TABLE2
          LDR  r0, [r1]       ; load first value ....
          STR  r0, [r2]       ; and store it in TABLE2
          ADD  r1, r1, #4     ; step r1 onto next word
          ADD  r2, r2, #4     ; step r2 onto next word
          LDR  r0, [r1]       ; load second value ...
          STR  r0, [r2]       ; and store it
          ....

```

- 前变址 (pre-indexed) 寻址模式



数据传送指令 – 前变址&自动变址

```

copy      ADR  r1, TABLE1    ; r1 points to TABLE1
          ADR  r2, TABLE2    ; r2 points to TABLE2
          LDR  r0, [r1]       ; load first value ....
          STR  r0, [r2]       ; and store it in TABLE2
          LDR  r0, [r1, #4]   ; load second value ...
          STR  r0, [r2, #4]   ; and store it
          ....

```

- 前变址不改变r1寄存器的值，在有些情况下不方便。
- 加“！”就可以成为前变址加自动变址。

```

LDR  r0, [r1, #4]! ; r0 := mem32[r1 + 4]
                     ; r1 := r1 + 4

```



数据传送指令 – 后变址

- 后变址 (**post-indexed**) 寻址：它允许基地址不加偏移即作为传送地址使用，而后再自动变址。

```
LDR r0, [r1], #4 ; r0 := mem32 [r1]  
; r1 := r1 + 4
```

copy	ADR	r1, TABLE1	; r1 points to TABLE1
	ADR	r2, TABLE2	; r2 points to TABLE2
loop	LDR	r0, [r1], #4	; get TABLE1 1st word
	STR	r0, [r2], #4	; copy it to TABLE2
	???		; if more, go back to loop
		
TABLE1		; < source of data >



数据传送指令总结

- 传送数据大小可以是无符号**8位字节**，而不是**32位字**。
传送地址可以对准任意字节，而不限制**4字节分界处**。
取出字节放在**r0**最低位字节，**r0**其余字节填充**0**。

```
LDRB r0, [r1] ; r0 := mem8 [r1]
```

- 寻址模式总结：

LDR	r0, [r1]	; register-indirect addressing
LDR	r0, [r1, # offset]	; pre-indexed addressing
LDR	r0, [r1, # offset]!	; pre-indexed, auto-indexing
LDR	r0, [r1], # offset	; post-indexed, auto-indexing
ADR	r0, address_label	; PC relative addressing



控制流指令

- 控制流指令确定下一步执行哪条指令。
- **转移指令**（无条件转移指令）

```
B      label ; unconditionally branch to label
.....
.....
label .....
```

- **条件转移指令：**可以用来控制循环

```
loop    MOV    r0, #10      ; initialize loop counted r0
        .....
        .....
        SUB    r0, r0, #1      ; decrement loop counter
        CMP    r0, #0          ; is it zero yet?
        BNE    loop           ; branch if r0 ≠ 0
```

S位

- 上一循环的例子可以简化为：

```
loop    MOV    r0, #10      ; initialize loop counted r0
        .....
        .....
        SUBS   r0, r0, #1      ; decrement loop counter
        BNE    loop           ; branch if r0 ≠ 0
```

- **SUBS**与**SUB**相同，只是前者更新**CPSR**的标志位
- **SUBS**指令后，**Z**状态位按照计算结果置位或清零
- 所有数据处理指令（**ADD, ADC**）都可以加**S**，表明标志位应该被更新
- 除了**BNE**之外，还有很多条件转移指令。



条件转移指令

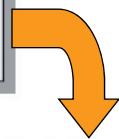
转移	解释	一般应用
B/BAL	无条件的/总是	总是执行转移
BEQ	相等	比较的结果为相等或零
BNE	不等	比较的结果为不等或非零
BPL	正	结果为正数或零
BMI	负	结果为负数
BCC/BLO	无进位/低于	算术操作未进位/无符号数比较, 结果为小于
BCS/BHS	有进位/高于或相等	算术操作进位/无符号数比较, 结果大于等于
BVC	无溢出	有符号整数操作, 未出现溢出
BVS	有溢出	有符号整数操作, 出现溢出
BGT	大于	有符号整数比较, 结果为大于
BGE	大于或相等	有符号整数比较, 结果为大于等于
BLT	小于	有符号整数比较, 结果为小于
BLE	小于或相等	有符号整数比较, 结果为小于等于
BHI	高于	无符号数比较, 结果为高于
BLS	低于或相等	无符号数比较, 结果为低于或相等



条件执行指令

- 条件执行指令：所有的ARM都可以条件执行

```
CMP r0, #5           ; if (r0 != 5) then  
BEQ BYPASS  
ADD r1, r1, r0       ;   r1 := r1 + r0 - r2  
SUB r1, r1, r2  
BYPASS .....
```



```
CMP r0, #5           ; if (r0 != 5) then  
ADDNE r1, r1, r0      ;   r1 := r1 + r0 - r2  
SUBNE r1, r1, r2  
BYPASS .....
```

ADDNE和SUBNE只有Z='0'才执行，即CMP指令是非零结果。



条件执行指令

- 有时巧妙地使用条件，有可能写出非常简练的代码

```
; if ( (a==b) && (c==d)) then e := e + 1;  
    CMP    r0, r1          ; r0 has a, r1 has b  
    CMPEQ r2, r3          ; r2 has c, r3 has d  
    ADDEQ r4, r4, #1      ; e := e+1
```

- 如果第一个比较发现操作数不同，则第二条和第三条指令都被跳过。
- 由于第二个比较指令使用了条件执行，从而实现了if语句中的逻辑“与”。
- 条件执行指令只有在条件序列小于等于3的时候才会达到高效，当大于时，要使用条件转移指令。



寄存器移位操作

- ARM一个巧妙特性：数据处理指令允许第二个寄存器操作数在同第一个操作数运算之前完成移位操作。

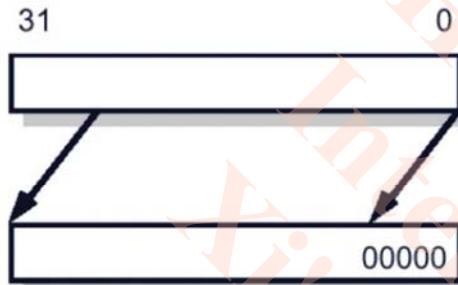
```
ADD    r3, r2, r1, LSL #3      ; r3 := r2 + 8 x r1
```

- LSL代表逻辑左移
- 注意它是一条ARM指令，在单个时钟周期内执行。
- 许多处理器采用独立的指令提供移位操作，但ARM将他们同基本的ALU操作合并在单条指令中。
- 同样可以使用寄存器值定义第二个操作数的移位位数

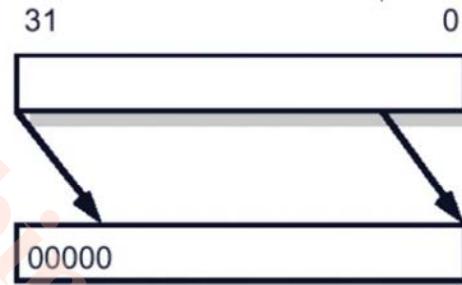
```
ADD    r5, r5, r3, LSL r2      ; r5 := r5 + r3 x 2**r1
```



ARM移位操作 – LSL and LSR



LSL #5

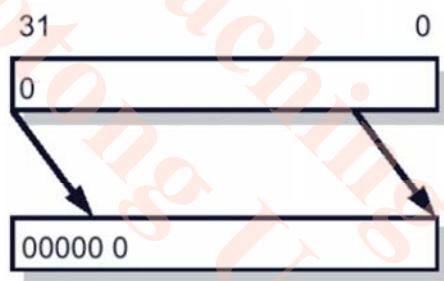


LSR #5

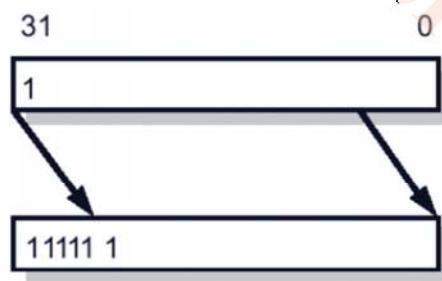
- **LSL (Logical Shift Left)** : 逻辑左移0~31位，空出的最低有效位用0填充；
- **LSR (Logical Shift Right)** : 逻辑右移0~32位，空出的最高有效位用0填充。



ARM移位操作 – ASL and ASR



ASR #5, positive operand

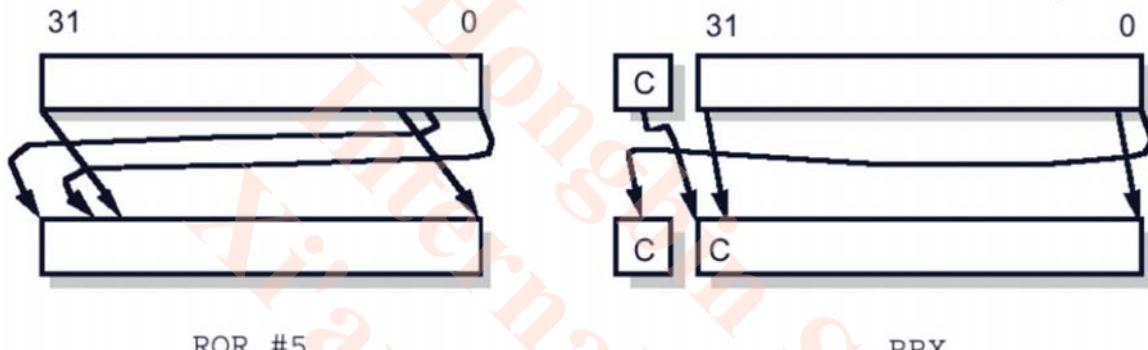


ASR #5, negative operand

- **ASL (Arithmetic Shift Left)** : 算术左移，与LSL同义；
- **ASR (Arithmetic Shift Right)** : 算术右移0~32位。如果源操作数是正数，则空出的最高有效位用0填充；如果是负数，则用1填充。



ARM移位操作 – ROR and RRX



- **ROR (Rotate Right)** : 循环右移0~32位，移出的字的最低有效位依次填入空出的最高有效位。
- **RRX (Rotate Right Extended by 1)** : 扩展1位的循环右移，空位（31）是用原来的标志位C填充，操作数右移1位。