ARM 指令集

寄存器和处理器模式

ARM 处理器有二十七个寄存器，其中一些是在一定条件下使用的，所以一次只能使用十六个...

- 寄存器 0 到寄存器 7 是通用寄存器并可以用做任何目的。不象 80x86 处理器那样要求特定寄存器被用做栈访问，或者象 6502 那样把数学计算的结果放置到一个累加器中，ARM 处理器在寄存器使用上是高度灵活的。

- 寄存器 8 到 12 是通用寄存器，但是在切换到 FIQ 模式的时候，使用它们的影子(shadow)寄存器。

- 寄存器 13 典型的用做 OS 栈指针，但可被用做一个通用寄存器。这是一个操作系统问题，不是一个处理器问题，所以如果你不使用栈，只要你以后恢复它，你可以在你的代码中自由的占用(corrupt)它。每个处理器模式都有这个寄存器的影子寄存器。

- 寄存器 14 专职持有返回点的地址以便于写子例程。当你执行带连接的分支的时候，把返回地址存储到 R14 中。同样在程序第一次运行的时候，把退出地址保存在 R14 中。R14 的所有实例必须被保存到其他寄存器中(不是实际上有效)或一个栈中。这

个寄存器在各个处理器模式下都有影子寄存器。一旦已经保存了连接地址，这个寄存器就可以用做通用寄存器了。

- 寄存器 15 是程序计数器。它除了持有指示程序当前使用的地址的二十六位数之外，还持有处理器的状态。

为更清晰一些... 提供下列图表:

```
User 模式  SVC 模式   IRQ 模式   FIQ 模式  APCS

R0 ------- R0 ------- R0 ------- R0        a1
R1 ------- R1 ------- R1 ------- R1        a2
R2 ------- R2 ------- R2 ------- R2        a3
R3 ------- R3 ------- R3 ------- R3        a4
R4 ------- R4 ------- R4 ------- R4        v1
R5 ------- R5 ------- R5 ------- R5        v2
R6 ------- R6 ------- R6 ------- R6        v3
R7 ------- R7 ------- R7 ------- R7        v4
R8 ------- R8 ------- R8         R8_fiq    v5
R9 ------- R9 ------- R9         R9_fiq    v6
R10 ------ R10 ------ R10        R10_fiq   sl
R11 ------ R11 ------ R11        R11_fiq   fp
R12 ------ R12 ------ R12        R12_fiq   ip
R13        R13_svc    R13_irq    R13_fiq   sp
R14        R14_svc    R14_irq    R14_fiq   lr
------------ R15 / PC ------------         pc
```

最右侧的列是 APCS 代码使用的名字，关于 APCS 的详情参见这里。

程序计数器构造如下:

```
  位   31  30  29  28  27  26  25----------2  1  0

       N   Z   C   V   I   F    程 序 计 数 器  S1 S0
```

对 R15 的详细解释，请参见 psr.html。

下面是你想知道的"模式"，比如上面提及的"FIQ"模式。

- 用户模式，运行应用程序的普通模式。限制你的内存访问并且你不能直接读取硬件设备。

- 超级用户模式(SVC 模式)，主要用于 SWI(软件中断)和 OS(操作系统)。这个模式有额外的**特权**，允许你进一步控制计算机。例如，你必须进入超级用户模式来读取一个插件(podule)。这不能在用户模式下完成。

- 中断模式(IRQ 模式)，用来处理发起中断的外设。这个模式也是有特权的。导致 IRQ 的设备有键盘、 VSync (在发生屏幕刷新的时候)、IOC 定时器、串行口、硬盘、软盘、等等...

- 快速中断模式(FIQ 模式)，用来处理发起快速中断的外设。这个模式是有特权的。导致 FIQ 的设备有处理数据的软盘，串行端口(比如在 82C71x 机器上的 A5000)和 Econet。

IRQ 和 FIQ 之间的区别是对于 FIQ 你必须尽快处理你事情并离开这个模式。IRQ 可以被 FIQ 所中断但 IRQ 不能中断 FIQ。为了使 FIQ 更快，所以有更多的影子寄存器。FIQ 不能调用 SWI。FIQ 还必须禁用中断。如果一个 FIQ 例程必须重新启用中断，则它太慢了并应该是 IRQ 而不是 FIQ。 Phew!

关于如果变更处理器的模式的详情请参照 psr.html。

32 位操作

这里的许多信息取自 ARM 汇编器手册。我现在没有 32 位处理器,就只能信任文档了... 这个文档中表述的 UMUL 和 UMLA 只能在 32bit 模式下进行是错误的。如果你的处理器(比如: StrongARM)可以这么做，则它可以在 32bit 或 26bit 下工作...

ARM2 和 ARM3 有一个 32 位数据总线和一个 26 位地址总线。在以后版本的 ARM 上，数据总线和地址总二者都是完全的 32 位宽。这解释了为什么一个“32 位处理器”被称为 26 位。数据宽度和指令/字大小是 32 位，并总是这样，但地址总线只是 24 位。因为 PC 总是字对齐的，一个地址中的低两位总是零，所以在 ARM2/ARM3 处理器上这些位持有处理器模式设置。尽管实际上只使用了 24 位，PC 的有效宽度仍是 26 位。

在老机器上这没有问题。4Mb 内存是基准的。一些人升级到 8Mb、和 16Mb 是理论上的限制。(Some people upgraded to 8Mb, and 16Mb was the theoretical limit.但是 RiscPC 使用一个 26 位程序计数器是不可能的，因为 RiscPC 允许安装 258Mb 内存，而 26 位只允许你寻址到 %11111111111111111111111100 (或 67108860 字节，或 64Mb)。这附带的解释了对应用任务的 28Mb 大小限制；就是希望系统与老的 RISC OS API 相容。

尽管这个汇编器站点的某些部分覆盖了 32 位模式(比如运行在 SVC32 下的一个简要的例子!)，但多数部分是关于 26 位模式操作的，这是为了与 RISC OS 的当前可获得的版本相兼容(就是 RISC OS 2 到 RISC OS 4)；我注意到部分例子不适用于 32 位。

RiscPC、Mico、RiscStation、A7000 等都有能力运行完全的 32 位操作系统；实际上 ARMLinux 就是这样的一个操作系统。RISC OS 不是，因为 RISC OS 需要，至少一个时期，保持与现存版本的兼容。这是个古老的两分问题(dichotomy)，有一个崭新的完全 32 位版本的 RISC OS 版本是美妙的,但当你发现许多你的现存软件不能继续运行(so much as load)就不那么美妙了!

RISC OS 不是完全的 26 位。一些处理程序(handler)需要工作在 32 位模式下；限制它的是金钱(就是说，谁为完全转换 RISC OS 付钱；谁为用来重建它们的代码的开发工具付钱(PD 在 RISC OS 上是强壮的))和必要性(就是说，很多人使用 Impression 而 CC 不再与我们同在；Impression 好象不能在更新的 RISC OS 上工作，所以如果人们需要的软件将不能工作，那么他们不会认为有升级的必要)。

为什么这如此重要？新的 ARM 处理器将不支持 26 位操作。尽管做了一些融合(ARM6、ARM7、StrongARM)，但气数就要尽了。你可以增加一个 26/32 位系统的复杂性，或者只用 32 位而得到更简单、更小的处理器。我们要么随波逐流，要么被甩下... 所以我们别无选择。

32 位体系

ARM 体系在 ARM6 系列中进行了重大变更。下面我将描述 26 位 和 32 位操作行为的不同之处。

在 ARM 6 中，程序计数器被扩展到完整的 32 位。结果是：

- PSR 从 PC 中分离到自己的寄存器 CPSR(当前的程序状态寄存器)中。
- 在改变处理器模式的时候，不再与 PC 一起保存 PSR；现在是每个有特权的模式都有一个额外的寄存器 - SPSR (保存的程序状态寄存器) - 用来持有前面模式的 PSR。
- 增加了使用这些新寄存器的指令。

除了允许 PC 使用完全的 32 位之外，还有进一步的变更，就是给 PSR 增加了额外的有特权的模式。这些模式用于处理未定义指令和异常终止例外：

- 未定义指令、异常终止、和超级用户不再共享同一个模式。去掉了在早期 ARM 上存在的对超级用户的那些限制。
- 在 ARM6 系列(和以后的其他兼容芯片)中通过设置片上某个控制寄存器来确定这些特征的可获得性。可以选择三个处理器配置中的一个：
    - 26 位程序和数据空间，这个配置强制 ARM 在 26 位地址空间中进行操作。在这个配置中只能获得四个 26 位模式(参照处理器模式描述)；不可能选择任何 32 位模式。在所有当前的 ARM6 和 7 系列上复位(reset)时被设置为这个模式。
    - 26 位程序空间和 32 位数据空间。除了禁止地址例外来允许数据传送操作访问完整的 32 位地址空间之外，与 26 位程序和地址空间配置相同。
    - 32 位程序和数据空间。这个配置把地址空间扩展成 32 位，并介入了对处理器模型的重大变更。在这个配置中你可以选择任何 26 位和 32 位处理器模式(参见下面的处理器模式)。

在配置成 32 位程序和数据空间的时候，ARM6 和 ARM7 系列支持十个有所重叠的处理器操作模式：

- 用户模式：正常的程序执行状态；
  或 User26 模式：一个 26 位版本。

- FIQ 模式：设计来支持一个数据传送或通道处理；
  或 FIQ26 模式：一个 26 位版本。
- IRQ 模式：用于通用中断处理；
  或 IRQ26 模式：一个 26 位版本。
- SVC 模式：用于操作系统的保护模式
  或 SVC26 模式：一个 26 位模式。
- 异常终止模式(ABT 模式)：在一个数据或指令预取异常终止(abort)的时候进入的模式。
- 未定义模式(UND 模式)：在执行了一个未定义的指令的时候进入的模式。

当在一个 26 位处理器模式中的时候，编程模型倒退成早期的 26 位 ARM 处理器。除了下列变动之外，它的行为与 ARM2aS 宏单元(macrocell)相同：

- 只在 ARM 被配置为 26 位程序和数据空间的时候，它才生成地址例外。在其他配置下 OS 仍然可以通过使用外部逻辑模拟地址例外的行为，比如用一个内存管理单元在超出 64Mbyte 范围的时候生成一个异常终止，并把这个异常终止转换成给这个应用程序的一个‘地址例外陷入’。
- 保持在通用寄存器和程序状态寄存器之间传送数据的新指令可操作。在调用了包含 26 位的 ARM 二进制代码的之后，操作系统可以使用这些新指令返回到一个 32 位模式。
- 当在一个 32 位程序和数据空间配置下的时候，所有例外(包括未定义指令和软件中断)把处理器返回到一个 32 位模式，所以必须修改操作系统来处理它们。
- 如果处理器尝试写到在 &0 和 &1F 之间包括二者(就是例外向量)的一个位置，则硬件将禁止写操作并生成一个数据异常终止。这允许操作系统来截获对例外向量的变动并把向量重定向到一些伪装(veneer)代码上。在调用 26 位例外处理程序之前，这些伪装代码应该把处理器置于一个 26 位模式中。

在所有其他方面，当在一个 26 位模式下进行操作的时候，ARM 表现的如同一个 26 位 ARM。CPSR 的相关的位将被组建(incorporated)回到 R15 中，来形成 I 和 F 位在位 27 和 26 的 PC/PSR。指令集表现的如同增加了 MRS 和 MSR 指令的 ARM2aS 宏单元。

在 ARM 6(和以后)的 32 位模式下可获得的寄存器有：

```
User26    SVC26     IRQ26     FIQ26       User      SVC       IRQ       ABT       UND
FIQ

R0 ----- R0 ----- R0 ----- R0 --    -- R0 ----- R0 ----- R0 ----- R0 ----- R0 -----
R1
R1 ----- R1 ----- R1 ----- R1 --    -- R1 ----- R1 ----- R1 ----- R1 ----- R1 -----
R2
R2 ----- R2 ----- R2 ----- R2 --    -- R2 ----- R2 ----- R2 ----- R2 ----- R2 -----
R2
```

```
R3 ----- R3 ----- R3 ----- R3 --   -- R3 ----- R3 ----- R3 ----- R3 ----- R3 -----
R3
R4 ----- R4 ----- R4 ----- R4 --   -- R4 ----- R4 ----- R4 ----- R4 ----- R4 -----
R4
R5 ----- R5 ----- R5 ----- R5 --   -- R5 ----- R5 ----- R5 ----- R5 ----- R5 -----
R5
R6 ----- R6 ----- R6 ----- R6 --   -- R6 ----- R6 ----- R6 ----- R6 ----- R6 -----
R6
R7 ----- R7 ----- R7 ----- R7 --   -- R7 ----- R7 ----- R7 ----- R7 ----- R7 -----
R7
R8 ----- R8 ----- R8       R8_fiq     R8 ----- R8 ----- R8 ----- R8 ----- R8
R8_fiq
R9 ----- R9 ----- R9       R9_fiq     R9 ----- R9 ----- R9 ----- R9 ----- R9
R9_fiq
R10 ---- R10 ---- R10      R10_fiq    R10 ---- R10 ---- R10 ---- R10 ---- R10
R10_fiq
R11 ---- R11 ---- R11      R11_fiq    R11 ---- R11 ---- R11 ---- R11 ---- R11
R11_fiq
R12 ---- R12 ---- R12      R12_fiq    R12 ---- R12 ---- R12 ---- R12 ---- R12
R12_fiq
R13      R13_svc  R13_irq  R13_fiq    R13      R13_svc  R13_irq  R13_abt
R13_und  R13_fiq
R14      R14_svc  R14_irq  R14_fiq    R14      R14_svc  R14_irq  R14_abt
R14_und  R14_fiq
--------- R15 (PC / PSR) ---------    ---------------------- R15 (PC)
----------------------
                                      ------------------------ CPSR
-----------------------
                                               SPSR_svc SPSR_irq SPSR_abt
SPSR_und SPSR_fiq
```

简要的说，32 位的与 26 位的不同是:

- PC 是完全的 32 位宽，并只用做程序计数器。
- PSR 包含在它自己的寄存器 CPSR 中。
- 每个有特权的模式都有一个专有的 SPSR 寄存器，用来保存 CPSR。
- 这里有两个新的特权模式，每个有特权的模式都有 R13 和 R14 的专有复件。

CPSR 和 SPSR 寄存器

CPSR 寄存器(和保存它的 SPSR 寄存器)中的位分配如下:

```
  31 30 29 28  ---   7   6   -   4   3   2   1   0
```

N	Z	C	V	I	F	M4	M3	M2	M1	M0	
						0	0	0	0	0	User26 模式
						0	0	0	0	1	FIQ26 模式
						0	0	0	1	0	IRQ26 模式
						0	0	0	1	1	SVC26 模式
						1	0	0	0	0	User 模式
						1	0	0	0	1	FIQ 模式
						1	0	0	1	0	IRQ 模式
						1	0	0	1	1	SVC 模式
						1	0	1	1	1	ABT 模式
						1	1	0	1	1	UND 模式

关于 N、Z、C、V 标志和 I、F 中断标志请参见(26 位) PSR。

这在实践中意味着什么?

多数 ARM 代码将正确的工作。唯一不能工作的是通过摆弄 R15 来设置处理器状态的那些操作。不幸的是，好象没有简便的方法修理这个问题。我检查了一个有潜在问题的 9K 程序(一个 MODE 7 teletext frame viewer，用 C 写的)，基本上查找:

- 用 R15 作为目的寄存器的 MOVS 指令。
- 以‘^’作为后缀并装载 R15 的 LDMFD 指令。

大约有 64 个指令被归入此类。

好象有没有什么方式来自动进行转换。基本上...

- 系统如何知道哪个是数据哪个是代码。实际上，一个灵巧的基于规则的程序能够可以做非常准确的猜测，但“非常准确的猜测”就足够了吗?
- 没有简单的指令替代。一个自动系统可以修补需要的指令并调整(jiggle)周围的代码，但这将导致不希望的副作用，比如一个 ADR 宏指令(directive)不在范围内(in range)。
- 需要难以置信的技巧(It is incredibly hacky)。当然，最好重新编译，或修改源代码。

这是很不容易的。这样小的变更，竟有如此严重(far-reaching)的后果。

程序状态寄存器

- MRS
- MSR

直接查看 R15/PSR 在 32-bit 模式下的详情

寄存器 15 (26-bit 模式):

R15 构造如下:

```
  Bit  31  30  29  28  27  26  25----------2  1  0

       N   Z   C   V   I   F    程 序 计 数 器  S1 S0
```

标志的意义:

N	Negative	如果结果是负数则置位
Z	Zero	如果结果是零则置位
C	Carry	如果发生进位则置位
O	Overflow	如果发生溢出则置位
I	IRQ	中断禁用
F	FIQ	快速中断禁用

S1 和 S0 是处理器模式标志:

S1	S0	模式
0	0	USR - 用户模式
0	1	FIQ - 快速中断模式
1	0	IRQ - 中断模式
1	1	SVC - 超级用户模式

在 R15 作为一个指令的第一个操作数的时候，只有程序计数器部分是可以获得的。所以，下列指令把 PC 复制到一个寄存器中并向这个目标寄存器加上 256:

```
  ADD     R0, R15, #256
```

(对于 BASIC 汇编器 R15 和 PC 的意思是相同的)

在 R15 作为第二个操作数的时候，所有 32 位都是可以获得的：程序计数器、标志、和状态。下列代码段将标识当前的处理器模式：

```
MOV     R0, #3          ; 装载一个位掩码(%11)到 R0 中
AND     R0, R0, PC; 把 R15 与 R0 做逻辑与并把结果放入 R0，来得到模式状态
CMP     R0, #3          ; 把模式与 '3' 相比较(SVC)
BEQ     svc             ; 如果等于 SVC 模式，分支到 'svc'
CMP     R0, #2          ; 把模式与 '2' 相比较 (IRQ)
BEQ     irq             ; 如果等于 IRQ 模式，分支到 'irq'
CMP     R0, #1          ; 把模式与 '1' 相比较(FIQ)
BEQ     fiq             ; 如果等于 FIQ 模式，分支到 'fiq'
CMP     R0, #0          ; 把模式与 '0' 相比较(USR)
BEQ     usr             ; 如果等于 USR 模式，分支到 'usr'
```

这个例子不遵从 32-bit 体系。如何在 32-bit 环境中读当前的模式请参照下面的章节。

改变处理器的状态：

要改变处理器模式、或者任何标志，我们需要用想要的标志 EOR(异或)状态标志，新状态 = 旧状态 EOR (1 << 28) 可以成为改变 oVerflow 标志的伪码。但是我们不能做这个简单的 EORS 操作，原因是这将导致随后的两个指令被跳过。不要担心，指令 TEQ 做一个假装的 EOR (结果不存储到任何地方)。把它与 P 后缀组合，则把结果的第 0、1、和 26 至 31 位直接写到 R15 的第 0、1、和 26 至 31 位，这是改变标志的一个简便的方法：

```
TEQP    R15, bit_mask
```

如果你处在允许你设置一个标志的一个模式中，则你只可以改变这个标志。

这个例子不遵从 32-bit 体系。如何在 32-bit 环境中改变模式请参照下面的章节。

可以被扩充它来改变处理器模式。例如，要进入 SVC 模式你可以：

```
MOV     R6, PC          ; 把 PC 的最初状态存储到 R6 中
ORR     R7, R6, #3      ; 设置 SVC 模式
TEQP    R7, #0          ; 把(在 R7 中的)模式标志写入 PC
```

而返回最初的模式是:

```
TEQP    R6, #0          ; 把(在 R6 中的)最初的模式写入 PC
```

在改变了模式之后，你应该进行一个空操作来允许这个寄存器安定下来。比如 MOV R0, R0 之类的东西就可以。废弃使用 NV 后缀的指令。

32 位 PSR

如同在 32 位操作中描述的那样，ARM 3 之后的处理器提供一个 32 bit 地址空间，它们把 PSR 移出 R15 并给予 R15 完整的 32 位位域，在其中存储当前位置的地址。目前，除了一些不太可能遇到的情况之外，RISC OS 工作在 26 位模式。

32 位模式是重要的,因为 26 位(在老的 PSR 中)把每个应用程序的可寻址内存的最大数量限制为 28Mb。这就是不管你安装了多少内存你不能拖动超过 28Mb 的下一个槽(drag the Next slot beyond 28Mb)的原因。

CPSR 寄存器(和保存它的 SPSR 寄存器)中的位分配如下:

31	30	29	28	---	7	6	-	4	3	2	1	0	
N	Z	C	V		I	F		M4	M3	M2	M1	M0	
								0	0	0	0	0	User26 模式
								0	0	0	0	1	FIQ26 模式
								0	0	0	1	0	IRQ26 模式
								0	0	0	1	1	SVC26 模式
								1	0	0	0	0	User 模式
								1	0	0	0	1	FIQ 模式
								1	0	0	1	0	IRQ 模式
								1	0	0	1	1	SVC 模式
								1	0	1	1	1	ABT 模式
								1	1	0	1	1	UND 模式

典型的，处理器将在 User26、FIQ26、IRQ26 和 SVC26 下操作。可以进入一个 32 位模式，但要格外小心。RISC OS 不希望这样，并且如果它发现自己在其中会非常生气!

操纵 32 位 PSR 的指令

你不能在 32 位模式中使用 MOVS PC, R14 或 LDMFD R13!, {registers, PC}^。也不能使用 ORRS PC, R14, #1<<28 来设置 V 标志。现在需要使用 MRS 和 MSR。

复制一个寄存器到 PSR 中

```
  MSR     CPSR, R0                ; 复制 R0 到 CPSR 中

  MSR     SPSR, R0                ; 复制 R0 到 SPSR 中

  MSR     CPSR_flg, R0            ; 复制 R0 的标志位到 CPSR 中

  MSR     CPSR_flg, #1<<28        ; 复制(立即值)标志位到 CPSR 中
```

复制 PSR 到一个寄存器中

```
  MRS     R0, CPSR                ; 复制 CPSR 到 R0 中

  MRS     R0, SPSR                ; 复制 SPSR 到 R0 中
```

指令格式

你有两个 PSR - CPSR 是当前的程序状态寄存器(Current Program Status Register)，而 SPSR 是保存的程序状态寄存器(Saved Program Status Register)(前面的处理器模式的 PSR)。每个有特权的模式都有自己的 SPSR，可获得的 PSR 有:

CPSR_all - 当前的

SPSR_svc - 保存的，SVC(32) 模式

SPSR_irq - 保存的，IRQ(32) 模式

SPSR_abt - 保存的，ABT(32) 模式

SPSR_und - 保存的，UND(32) 模式

SPSR_fiq - 保存的，FIQ(32) 模式

你不能显式的指定把 CPSR 保存到哪个 SPSR 中，比如 SPSR_fiq。而是必须变更到 FIQ 模式并接着保存到 SPSR。换句话说，你只能在你所在的模式中改变这个模式的 SPSR。使用 _flg 后缀允许你改变标志位而不影响控制位。

在 user(32) 模式中，保护 CPSR 的控制位，你只能改变条件标志。在其他模式中，可获得整个 CPSR。你不应该指定 R15 为一个源寄存器或一个目标寄存器。最后，在 user(32) 模式中，你不能尝试访问 SPSR，因为它不存在!

要设置 V 标志:

```
 MSR      CPSR_flg, #&10000000
```

这将设置 V 标志但不影响控制位。

要改变模式:

```
  MRS      R0, CPSR_all              ; 复制 PSR

  BIC      R0, R0, #&1F              ; 清除模式位

  ORR      R0, R0, #new_mode         ; 把模式位设置为新模式

  MSR      CPSR_all, R0              ; 写回 PSR，变更模式
```

现在我们要做的是进入 SVC32 模式并设置 Z 标志。接着我们返回 SVC26 模式并‘测试’是否设置了 Z。
RISC OS 不希望发现自己处在 32 位模式中，所以我们要禁止所有中断并保持它们这样(keep them that way)。尽管这些代码应该执行的非常快，但我们不应当冒任何风险...

你可能觉得 32 位模式不是非常有用。在当前版本的 RISC OS 下，这是事实。实际上，就我而言，32 位模式提供给你的只是:

访问大于 28Mb 的区域。在 RISC OS 上这不是真的很重要，在这个系统里 web 浏览器适合于 1 M 或 2 M，而重要的艺术程序为那些非常巨大的图象提供它们自己的虚拟内存系统。

本文档的最初版本，和最初的 ARM 汇编器指南包括...

StrongARM 提供了两个指令(UMUL 和 UMLA、IIRC)，它们处理 64 位乘法。这只能在 32 位模式下获得。

这是错误的。在 26 位模式下可以使用扩展的乘法；MP3 解码器就使用了它!

尽管 32 位模式的利益好象不是多的那么惊人，新近的处理器(比如 Xscale)不再支持 26 位模式，所以 RISC OS 和它的应用程序要在 32 位环境下工作则必须经过修改。听起来不是很多,但是如果所有补偿/改变 R15 中的 PSR 位的引用都必须被变更为对不在 R15 中的独立的 PSR 的引用，这就突然变成一个非常重大的问题了。还有你不能继续用一个指令来恢复 PSR 并分支回到调用者，现在这需要两个独立的指令。为此代码必须重写。你不能简单的用另一个指令来修补...

寄存器装载和存储

- LDM
- LDR
- STM
- STR
- SWP

它们可能是能获得的最有用的指令。其他指令都操纵寄存器，所以必须把数据从内存装载寄存器并把寄存器中的数据存储到内存中。

传送单一数据

使用单一数据传送指令(STR 和 LDR)来装载和存储单一字节或字的数据从/到内存。寻址是非常灵活的。

首先让我们查看指令格式:

```
LDR{条件}     Rd, <地址>
STR{条件}     Rd, <地址>
LDR{条件}B    Rd, <地址>
STR{条件}B    Rd, <地址>
```

指令格式

这些指令装载和存储 Rd 的值从/到指定的地址。如果象后面两个指令那样还指定了‘B’，则只装载或存储一个单一的字节；对于装载，寄存器中高端的三个字节被置零(zeroed)。

地址可以是一个简单的值、或一个偏移量、或者是一个被移位的偏移量。可以还可以把合成的有效地址写回到基址寄存器(去除了对加/减操作的需要)。

各种寻址方式的示例：

译注：下文中的 Rbase 是表示基址寄存器，Rindex 表示变址寄存器，index 表示偏移量，偏移量为 12 位的无符号数。用移位选项表示比例因子。标准寻址方式 - 用 AT&T 语法表示为 disp(base, index, scale)，用 Intel 语法表示为 [base + index*scale + disp]，中的变址(连带比例因子)与偏移量不可兼得。

```
STR     Rd, [Rbase]          存储 Rd 到 Rbase 所包含的有效地址。
```

STR Rd, [Rbase, Rindex] 存储 Rd 到 Rbase + Rindex 所合成的有效地址。

STR Rd, [Rbase, #index] 存储 Rd 到 Rbase + index 所合成的有效地址。
index 是一个立即值。
例如，STR Rd, [R1, #16] 将把 Rd 存储到 R1+16。

STR Rd, [Rbase, Rindex]! 存储 Rd 到 Rbase + Rindex 所合成的有效地址，
并且把这个新地址写回到 Rbase。

STR Rd, [Rbase, #index]! 存储 Rd 到 Rbase + index 所合成的有效地址，
并且并且把这个新地址写回到 Rbase。

STR Rd, [Rbase], Rindex 存储 Rd 到 Rbase 所包含的有效地址。
把 Rbase + Rindex 所合成的有效地址写回 Rbase。

STR Rd, [Rbase, Rindex, LSL #2]
存储 Rd 到 Rbase + (Rindex * 4) 所合成的有效地址。

STR Rd, place 存储 Rd 到 PC + place 所合成的有效地址。
你当然可以在这些指令上使用条件执行。但要注意条件标志要先于字节标志，所以如果你希望在结果是等于的时候装载一个字节，要用的指令是 LDREQB Rx, <address>(不是 LDRBEQ...)。

如果你指定预先变址寻址(这里的基址和变址都在方括号中)，用是否存在‘!’来控制写回操作。上面的第 4 和第 5 个例子中使用了这个标志。你可以使用它来在内存中自动正向或反向移动。一个字符串打印例程将变成:

```
.loop
  LDRB   R0, [R1, #1]!
  SWI    "OS_WriteC"
  CMP    R0, #0
  BNE    loop
```

而不是:

```
.loop
  LDRB   R0, [R1]
  SWI    "OS_WriteC"
  ADD    R1, R1, #1
```

```
CMP     R0, #0
BNE     loop
```

对于过后变址寻址‘!’是无效的(这里的变址在方括号外面，比如上面的例子6)，因为写回是暗含的。

如同你见到的那样，变址可以被移位来实现比例缩放。除此之外，可以从基址上减去偏移量。在这种情况下，你可以使用如下代码:

```
LDRB    R0, [R1, #-1]
```

尽管你可以存储或装载 PC，但你不可以用装载或存储指令来修改 PSR。要装载一个被存储的‘状态’并正确的恢复它，请使用:

```
LDR     R0, [Rbase]
MOVS    R15, R0
```

假如你在有特权的模式下，MOVS 将导致 PSR 的位被更改。
对 PC 使用 MOVS 不遵从 32-bit 体系，你需要使用 MRS 和 MSR 来处理 PSR 。

依照 ARM 汇编手册:

译注：下文所叙述内容针对的是小端字节序配置，对大端字节序配置在手册中另有专门叙述。

- 如果提供的地址在一个字边界上，则字节装载(LDRB)使用在 0 至 7 位上的数据，如果在一个字地址加上一个字节上，则使用 8 至 15 位，以此类推。选择的字节被放入目标寄存器的低端 8 位中，并把寄存器中其余的位用零填充。
- 字节存储(STRB)在数据总线上重复源寄存器的的低端 8 位 4 次。由外部的内存系统来激活适当的字节子系统来存储数据。
- 字装载(LDR)或字存储(STR)将生成一个字对齐的地址。使用一个非字对齐的地址将有不明显和未规定的结果。实际上提示的是你不能使用 LDR 从一个非对齐的地址装载一个字。

传送多个数据

使用多数据传送指令(LDM 和 STM) 来装载和存储多个字的数据从/到内存。

LDM/STM 的主要用途是把需要保存的寄存器复制到栈上。如我们以前见到过的 STMFD R13!, {R0-R12, R14}。

指令格式是:

```
xxM{条件}{类型}  Rn{!}, <寄存器列表>{^}
```

‘xx’是 LD 表示装载，或 ST 表示存储。

再加 4 种‘类型’就变成了 8 个指令:

栈	其他	
LDMED	LDMIB	预先增加装载
LDMFD	LDMIA	过后增加装载
LDMEA	LDMDB	预先减少装载
LDMFA	LDMDA	过后减少装载
STMFA	STMIB	预先增加存储
STMEA	STMIA	过后增加存储
STMFD	STMDB	预先减少存储
STMED	STMDA	过后减少存储

指令格式

汇编器关照如何映射这些助记符。注意 ED 不同于 IB；只对于预先减少装载是相同的。在存储的时候，ED 是过后减少的。

FD、ED、FA、和 EA 指定是满栈还是空栈，是升序栈还是降序栈。一个满栈的栈指针指向上次写的最后一个数据单元，而空栈的栈指针指向第一个空闲单元。一个降序栈是在内存中反向增长(就是说，从应用程序空间结束处开始反向增长)而升序栈在内存中正向增长。

其他形式简单的描述指令的行为，意思分别是过后增加(Increment After)、预先增加(Increment Before)、过后减少(Decrement After)、预先减少(Decrement Before)。

RISC OS 使用传统的满降序栈。在使用符合 APCS 规定的编译器的时候，它通常把你的栈指针设置在应用程序空间的结束处并接着使用一个 FD (满降序 - Full Descending)栈。如果你与一个高级语言(BASIC 或 C)一起工作，你将别无选择。栈指针(传统上是 R13)指向一个满降序栈。你必须继续这个格式，或则建立并管理你自己的栈(如果你是死硬派人士那么你可能喜欢这样做!)。

‘基址’是包含开始地址的寄存器。在传统的 RISC OS 下，它是栈指针 R13，但你可以使用除了 R15 之外的任何可获得的寄存器。

如果你想把复制操作后栈顶的当前的内存地址保存到栈指针中，可以寄存器按从最低到最高的编号次序与到从低端到高端的内存之间传送数据。并且因为用指令中的一个单一的位来表示是否保存一个寄存器，不可能指定某个寄存器两次。它的副作用是不能用下面这样的代码:

```
STMFD  R13!, {R0, R1}
LDMFD  R13!, {R1, R0}
```

来交换两个寄存器的内容。

提供了一个有用的简写。要包含一个范围的寄存器，可以简单的只写第一个和最后一个，并在其间加一个横杠。例如 R0-R3 等同与 R0, R1, R2, R3 只是更加整齐和理智而已...

在把 R15 存储到内存中的时候，还保存了 PSR 位。在重新装载 R15 的时候，除非你要求否则不恢复 PSR 位。要求的方法是在寄存器列表后跟随一个‘^’。

```
  STMFD   R13!, {R0-R12, R14}
  ...
  LDMFD   R13!, {R0-R12, PC}
```

这保存所有的寄存器，做一些事情，接着重新装载所有的寄存器。从 R14 装载 PC，它由一个 BL 或此类指令所设置。不触及 PSR 标志。

```
  STMFD   R13!, {R0-R12, R14}
  ...
  LDMFD   R13!, {R0-R12, PC}^
```

这保存所有的寄存器，做一些事情，接着重新装载所有的寄存器。从 R14 装载 PC，它由一个 BL 或此类指令所设置。变更 PSR 标志。

警告: 这些代码不遵从 32 bit 体系。你需要使用 MRS 和 MSR 来处理 PSR，你不能使用‘^’后缀。

注意在这两个例子中，R14 被直接装载到 PC 中。这节省了对 MOV(S) R14 到 R15 中的需要。

警告: 使用 MOVS PC, ... 不遵从 32 bit 体系。你需要使用 MRS 和 MSR 来处理 PSR。

SWP：单一数据交换

(Swap)

```
  SWP{条件}{B}  <dest>, <op 1>, [<op 2>]
```

指令格式

SWP 将:

- 从操作数 2 所指向的内存装载一个字并把这个字放置到目的寄存器中。
- 把寄存器操作数 1 的内容存储到同一个地址中。

如果目的和操作数 1 是同一个寄存器，则把寄存器的内容和给定内存位置的内容进行交换。

如果提供了 B 后缀，则将传送一个字节，否则传送一个字。

算术和逻辑指令

指令格式

ADC：带进位的加法

(Addition with Carry)

ADC{条件}{S} <dest>, <op 1>, <op 2>

dest = op_1 + op_2 + carry

ADC 将把两个操作数加起来，并把结果放置到目的寄存器中。它使用一个进位标志位，这样就可以做比 32 位大的加法。下列例子将加两个 128 位的数。

128 位结果: 寄存器 0 、1、2、和 3

第一个 128 位数: 寄存器 4 、5、6、和 7

第二个 128 位数: 寄存器 8 、9、10、和 11 。

```
ADDS    R0, R4, R8          ; 加低端的字
ADCS    R1, R5, R9          ; 加下一个字，带进位
ADCS    R2, R6, R10         ; 加第三个字，带进位
ADCS    R3, R7, R11         ; 加高端的字，带进位
```

如果如果要做这样的加法，不要忘记设置 S 后缀来更改进位标志。

ADD：加法

(Addition)

ADD{条件}{S} <dest>, <op 1>, <op 2>

```
                dest = op_1 + op_2
```
ADD 将把两个操作数加起来，把结果放置到目的寄存器中。操作数 1 是一个寄存器，操作数 2 可以是一个寄存器，被移位的寄存器，或一个立即值:
```
  ADD     R0, R1, R2              ; R0 = R1 + R2
  ADD     R0, R1, #256            ; R0 = R1 + 256
  ADD     R0, R2, R3,LSL#1        ; R0 = R2 + (R3 << 1)
```
加法可以在有符号和无符号数上进行。

AND：逻辑与

(logical AND)

```
  AND{条件}{S}  <dest>, <op 1>, <op 2>

                dest = op_1 AND op_2
```
AND 将在两个操作数上进行逻辑与，把结果放置到目的寄存器中；对屏蔽你要在上面工作的位很有用。 操作数 1 是一个寄存器，操作数 2 可以是一个寄存器，被移位的寄存器，或一个立即值:
```
  AND     R0, R0, #3              ; R0 = 保持 R0 的位 0 和 1，丢弃其余的位。
```
AND 的真值表(二者都是 1 则结果为 1):

Op_1	Op_2	结果
0	0	0
0	1	0
1	0	0
1	1	1

BIC：位清除

(Bit Clear)

```
  BIC{条件}{S}  <dest>, <op 1>, <op 2>

                dest = op_1 AND (!op_2)
```
BIC 是在一个字中清除位的一种方法，与 OR 位设置是相反的操作。操作数 2 是一个 32 位位掩码(mask)。如果如果在掩码中设置了某一位，则清除这一位。未设置的掩码位指示此位保持不变。

```
  BIC      R0, R0, #%1011           ; 清除 R0 中的位 0、1、和 3。保持其
余的不变。
```
BIC 真值表:

Op_1	Op_2	结果
0	0	0
0	1	0
1	0	1
1	1	0

译注：逻辑表达式为 Op_1 AND NOT Op_2

EOR：逻辑异或

(logical Exclusive OR)

```
  EOR{条件}{S}  <dest>, <op 1>, <op 2>

                dest = op_1 EOR op_2
```
EOR 将在两个操作数上进行逻辑异或，把结果放置到目的寄存器中；对反转特定的位有用。操作数 1 是一个寄存器，操作数 2 可以是一个寄存器，被移位的寄存器，或一个立即值:
```
  EOR      R0, R0, #3               ; 反转 R0 中的位 0 和 1
```
EOR 真值表(二者不同则结果为 1):

Op_1	Op_2	结果
0	0	0
0	1	1
1	0	1
1	1	0

MOV：传送

(Move)

```
  MOV{条件}{S}  <dest>, <op 1>

                dest = op_1
```
MOV 从另一个寄存器、被移位的寄存器、或一个立即值装载一个值到目的寄存器。你可以指定相同的寄存器来实现 NOP 指令的效果，你还可以专门移位一个寄存器:
```
  MOV      R0, R0                   ; R0 = R0... NOP 指令
```

```
  MOV      R0, R0, LSL#3              ; R0 = R0 * 8
```
如果 R15 是目的寄存器，将修改程序计数器或标志。这用于返回到调用代码，方法是把连接寄存器的内容传送到 R15:
```
  MOV      PC, R14                    ; 退出到调用者

  MOVS     PC, R14                    ; 退出到调用者并恢复标志位
                                        (不遵从 32-bit 体系)
```

MVN：传送取反的值

(Move Negative)

```
  MVN{条件}{S}  <dest>, <op 1>

                dest = !op_1
```
MVN 从另一个寄存器、被移位的寄存器、或一个立即值装载一个值到目的寄存器。不同之处是在传送之前位被反转了，所以把一个被取反的值传送到一个寄存器中。这是逻辑非操作而不是算术操作，这个取反的值加 1 才是它的取负的值:
```
  MVN      R0, #4                     ; R0 = -5

  MVN      R0, #0                     ; R0 = -1
```

ORR：逻辑或

(logical OR)

```
  ORR{条件}{S}  <dest>, <op 1>, <op 2>

                dest = op_1 OR op_2
```
OR 将在两个操作数上进行逻辑或，把结果放置到目的寄存器中；对设置特定的位有用。操作数 1 是一个寄存器，操作数 2 可以是一个寄存器，被移位的寄存器，或一个立即值:
```
  ORR      R0, R0, #3                 ; 设置 R0 中位 0 和 1
```
OR 真值表(二者中存在 1 则结果为 1):

Op_1	Op_2	结果
0	0	0
0	1	1
1	0	1
1	1	1

RSB：反向减法

(Reverse Subtraction)

RSB{条件}{S} <dest>, <op 1>, <op 2>

dest = op_2 - op_1

SUB 用操作数 **two** 减去操作数 **one**，把结果放置到目的寄存器中。操作数 1 是一个寄存器，操作数 2 可以是一个寄存器，被移位的寄存器，或一个立即值:

```
RSB     R0, R1, R2              ; R0 = R2 - R1
RSB     R0, R1, #256            ; R0 = 256 - R1
RSB     R0, R2, R3,LSL#1        ; R0 = (R3 << 1) - R2
```

反向减法可以在有符号或无符号数上进行。

RSC：带借位的反向减法

(Reverse Subtraction with Carry)

RSC{条件}{S} <dest>, <op 1>, <op 2>

dest = op_2 - op_1 - !carry

同于 SBC，但倒换了两个操作数的前后位置。

SBC：带借位的减法

(Subtraction with Carry)

SBC{条件}{S} <dest>, <op 1>, <op 2>

dest = op_1 - op_2 - !carry

SBC 做两个操作数的减法，把结果放置到目的寄存器中。它使用进位标志来表示借位，这样就可以做大于 32 位的减法。SUB 和 SBC 生成进位标志的方式不同于常规，如果需要借位则**清除**进位标志。所以，指令要对进位标志进行一个**非**操作 - 在指令执行期间自动的反转此位。

SUB：减法

(Subtraction)

SUB{条件}{S} <dest>, <op 1>, <op 2>

dest = op_1 - op_2

SUB 用操作数 **one** 减去操作数 **two**，把结果放置到目的寄存器中。操作数 1 是一个寄存器，操作数 2 可以是一个寄存器，被移位的寄存器，或一个立即值:

```
SUB     R0, R1, R2              ; R0 = R1 - R2
SUB     R0, R1, #256            ; R0 = R1 - 256
SUB     R0, R2, R3,LSL#1        ; R0 = R2 - (R3 << 1)
```

减法可以在有符号和无符号数上进行。

移 位

- LSL
- ASL
- LSR
- ASR
- ROR
- RRX

ARM 处理器组建了可以与数据处理指令(ADC、ADD、AND、BIC、CMN、CMP、EOR、MOV、MVN、ORR、RSB、SBC、SUB、TEQ、TST)一起使用的桶式移位器(barrel shifter)。你还可以使用桶式移位器影响在 LDR/STR 操作中的变址值。

译注：移位操作在 ARM 指令集中不作为单独的指令使用，它是指令格式中是一个字段，在汇编语言中表示为指令中的选项。如果数据处理指令的第二个操作数或者单一数据传送指令中的变址是寄存器，则可以对它进行各种移位操作。如果数据处理指令的第二个操作数是立即值，在指令中用8 位立即值和 4 位循环移位来表示它，所以对大于 255 的立即值，汇编器尝试通过在指令中设置循环移位数量来表示它，如果不能表示则生成一个错误。在逻辑类指令中，逻辑运算指令由指令中 S 位的设置或清除来确定是否影响进位标志，而比较指令的 S 位总是设置的。在单一数据传送指令中指定移位的数量只能用立即值而不能用寄存器。

下面是给不同的移位类型的六个助记符:

LSL 逻辑左移

ASL 算术左移
LSR 逻辑右移
ASR 算术右移
ROR 循环右移
RRX 带扩展的循环右移

ASL 和 LSL 是等同的，可以自由互换。

你可以用一个立即值(从 0 到 31)指定移位数量，或用包含在 0 和 31 之间的一个值的寄存器指定移位数量。

逻辑或算术左移

(Logical or Arithmetic Shift Left)

```
Rx, LSL #n    or
Rx, ASL #n    or
Rx, LSL Rn    or
Rx, ASL Rn
```

接受 Rx 的内容并按用'n'或在寄存器 Rn 中指定的数量向高有效位方向移位。最低有效位用零来填充。除了概念上的第 33 位(就是被移出的最小的那位)之外丢弃移出最左端的高位，如果逻辑类指令中 S 位被设置了，则此位将成为从桶式移位器退出时进位标志的值。

考虑下列:

```
MOV    R1, #12
MOV    R0, R1, LSL#2
```

在退出时，R0 是 48 。 这些指令形成的总和是 R0 = #12, LSL#2 等同于 BASIC 的 R0 = 12 << 2

逻辑右移

(Logical Shift Right)

```
Rx, LSR #n    or
Rx, LSR Rn
```

它在概念上与左移相对。把所有位向更低有效位方向移动。如果逻辑类指令中 S 位被设置了，则把最后被移出最右端的那位放置到进位标志中。它同于 BASIC 的 register = value >>> shift。

算术右移

(Arithmetic Shift Right)

```
  Rx, ASR #n     or
  Rx, ASR Rn
```

类似于 LSR，但使用要被移位的寄存器(Rx)的第 31 位的值来填充高位，用来保护补码表示中的符号。如果逻辑类指令中 S 位被设置了，则把最后被移出最右端的那位放置到进位标志中。它同于 BASIC 的 register = value >> shift。

循环右移

(Rotate Right)

```
  Rx, ROR #n     or
  Rx, ROR Rn
```

循环右移类似于逻辑右移，但是把从右侧移出去的位放置到左侧，如果逻辑类指令中 S 位被设置了，则同时放置到进位标志中，这就是位的'循环'。一个移位量为 32 的操作将导致输出与输入完全一致，因为所有位都被移位了 32 个位置，又回到了开始时的位置!

带扩展的循环右移

(Rotate Right with extend)

```
  Rx, RRX
```

这是一个 ROR#0 操作，它向右移动一个位置 - 不同之处是，它使用处理器的进位标志来提供一个要被移位的 33 位的数量。

Thanks to *Ian Jeffray* for the correction.

乘法指令

- MLA
- MUL

指令格式

这两个指令与普通算术指令在对操作数的限制上有所不同:

1. 给出的所有操作数、和目的寄存器必须为简单的寄存器。
2. 你不能对操作数 2 使用立即值或被移位的寄存器。
3. 目的寄存器和操作数 1 必须是不同的寄存器。
4. 最后，你不能指定 R15 为目的寄存器。

MLA：带累加的乘法

(Multiplication with Accumulate)

MLA{条件}{S} <dest>, <op 1>, <op 2>, <op 3>

dest = (op_1 * op_2) + op_3

MLA 的行为同于 MUL，但它把操作数 3 的值加到结果上。这在求总和时有用。

MUL：乘法

(Multiplication)

MUL{条件}{S} <dest>, <op 1>, <op 2>

dest = op_1 * op_2

MUL 提供 32 位整数乘法。如果操作数是有符号的，可以假定结果也是有符号的。

比较指令

- CMN
- CMP
- TEQ
- TST

指令格式

译注：CMP 和 CMP 是算术指令，TEQ 和 TST 是逻辑指令。把它们归入一类的原因是它们的 S 位总是设置的，就是说，它们总是影响标志位。

CMN： 比较取负的值

(Compare Negative)

CMN{条件}{P} <op 1>, <op 2>

status = op_1 - (- op_2)

CMN 同于 CMP，但它允许你与小负值(操作数 2 的取负的值)进行比较，比如难于用其他方法实现的用于结束列表的 -1 。这样与 -1 比较将使用:

```
CMN     R0, #1                  ; 把 R0 与 -1 进行比较
```

详情参照 CMP 指令。

CMP： 比较

(Compare)

CMP{条件}{P} <op 1>, <op 2>

status = op_1 - op_2

CMP 允许把一个寄存器的内容如另一个寄存器的内容或立即值进行比较，更改状态标志来允许进行条件执行。它进行一次减法，但不存储结果，而是正确的更改标志。标志表示的是操作数 1 比操作数 2 如何(大小等)。如果操作数 1 大于操作操作数 2，则此后的有 GT 后缀的指令将可以执行。

明显的，你不需要显式的指定 S 后缀来更改状态标志... 如果你指定了它则被忽略。

TEQ：测试等价

(Test Equivalence)

TEQ{条件}{P} <op 1>, <op 2>

Status = op_1 EOR op_2

TEQ 类似于 TST。区别是这里的概念上的计算是 EOR 而不是 AND 。这提供了一种查看两个操作数是否相同而又不影响进位标志(不象 CMP 那样)的方法。加上 P 后缀的 TEQ 还可用于改变 R15 中的标志(在 26-bit 模式中)。详情请参照 psr.html，在32-bit 模式下如何做请参见这里。

TST：测试位

(Test bits)

TST{条件}{P} <op 1>, <op 2>

Status = op_1 AND op_2

TST 类似于 CMP，不产生放置到目的寄存器中的结果。而是在给出的两个操作数上进行操作并把结果反映到状态标志上。使用 TST 来检查是否设置了特定的位。操作数 1 是要测试的数据字而操作数 2 是一个位掩码。经过测试后，如果匹配则设置 Zero 标志，否则清除它。象 CMP 那样，你不需要指定 S 后缀。

```
TST     R0, #%1                 ; 测试在 R0 中是否设置了位 0。
```

分支指令

- B
- BL

指令格式

B：分支

(Branch)

B{条件} <地址>

B 是最简单的分支。一旦遇到一个 B 指令，ARM 处理器将立即跳转到给定的地址，从那里继续执行。注意存储在分支指令中的实际的值是相对当前的 R15 的值的一个偏移量；而

不是一个绝对地址。它的值由汇编器来计算，它是24 位有符号数，左移两位后有符号扩展为 32 位，表示的有效偏移为 26 位(+/- 32 M)。

在其他处理器上，你可能经常见到这样的指令:

```
  OPT 1
  LDA &70
  CMP #0
  BEQ Zero
  STA &72
 .Zero RTS
```

(取自 Acorn Electron User Guide issue 1 page 213)

在 ARM 处理器上，它们将变成下面这些东西:

```
  OPT     1
  ADR     R1, #&70
  LDR     R0, [R1]
  CMP     #0
  BEQ     Zero
  STR     R0, [R1, #2]
 .Zero
  MOV     PC, R14
```

这不是一个很好的例子，但你可以构想如何更好的去条件执行而不是分支。另一方面，如果你有大段的代码或者你的代码使用状态标志，那么你可以使用条件执行来实现各类分支: 这样一个单一的简单条件执行指令可以替代在其他处理器中存在的所有这些分支和跳转指令。

```
  OPT     1
  ADR     R1, #&70
  LDR     R0, [R1]
  CMP     R0, #0
  STRNE   R0, [R1, #2]
  MOV     PC, R14
```

BL：带连接的分支

(Branch with Link)

```
  BL{条件}  〈地址〉
```

BL 是另一个分支指令。就在分支之前，在寄存器 14 中装载上 R15 的内容。你可以重新装载 R14 到 R15 中来返回到在这个分支之后的那个指令，它是子例程的一个基本但强力的实现。它的作用在屏幕装载器 2 (例子 4) 中得以很好的展现...

```
    .load_new_format
```

```
    BL      switch_screen_mode
    BL      get_screen_info
    BL      load_palette

.new_loop
    MOV     R1, R5
    BL      read_byte
    CMP     R0, #255
    BLEQ    read_loop
    STRB    R0, [R2, #1]!
```

...在这里我们见到在装载器循环之前调用了三个子例程。接着，一旦满足了条件执行就在循环中调用了 *read_byte* 子例程。

条件执行

条件执行

指令格式

ARM 处理器的一个非常特殊的特征是它的条件执行。我们指的不是基本的如果进位则分支，ARM 使这个逻辑阶段进一步深化为如果进位则 XXX - 这里的 XXX 是任何东西。

为了举例，下面是 Intel 8086 处理器分支指令的一个列表:

```
JA    Jump if Above
JAE   Jump if Above or Equal
JB    Jump if Below
JBE   Jump if Below or Equal
JC    Jump if Carry
JCXZ  Jump if CX Zero (CX is a register that can be used for loop counts)
JE    Jump if Equal
JG    Jump if Greater than
JGE   Jump if Greater than or Equal
JL    Jump if Less than
JLE   Jump if Less Than or Equal
JMP   JuMP
JNA   Jump if Not Above
JNAE  Jump if Not Above or Equal
JNB   Jump if Not Below
JNBE  Jump if Not Below or Equal
JNC   Jump if No Carry
```

```
  JNE   Jump if Not Equal
  JNG   Jump if Not Greater than
  JNGE  Jump if Not Greater than or Equal
  JNL   Jump if Not Less than
  JNLE  Jump if Not Less than or Equal
  JNO   Jump if Not Overflow
  JNP   Jump if Not Parity
  JNS   Jump if Not Sign
  JNZ   Jump if Not Zero
  JO    Jump if Overflow
  JP    Jump if Parity
  JPE   Jump if Parity Even
  JPO   Jump if Parity Odd
  JS    Jump if Sign
  JZ    Jump if Zero
```

80386 添加了:

```
  JECXZ Jump if ECX Zero
```

作为对比，ARM 处理器只提供了:

```
  B     分支
  BL    带连接的分支
```

但 ARM 提供了条件执行，你可以不受这个表面上不灵活的方式的限制:

```
  BEQ   Branch if EQual
  BNE   Branch if Not Equal
  BVS   Branch if oVerflow Set
  BVC   Branch if oVerflow Clear
  BHI   Branch if HIgher
  BLS   Branch if Lower or the Same
  BPL   Branch if PLus
  BMI   Branch if MInus
  BCS   Branch if Carry Set
  BCC   Branch if Carry Clear
  BGE   Branch if Greater than or Equal
  BGT   Branch if Greater Than
  BLE   Branch if Less than or Equal
  BLT   Branch if Less Than

  BLEQ  Branch with Link if EQual
  ....
  BLLT  Branch with Link if Less Than
```

还有两个代码，

- AL - ALways，缺省条件所以不须指定
- NV - NeVer，不是非常有用。你无论如何不要使用这个代码...

当你发现所有 Bxx 指令实际上是同一个指令的时候，紧要关头就到了。接着你会想，如果你可以在一个分支指令上加上所有这些条件，那么对一个寄存器装载指令能否加上它们？答案是可以。

下面是可获得的条件代码的列表：

EQ：等于
如果一次比较之后设置了 Z 标志。

NE：不等于
如果一次比较之后清除了 Z 标志。

VS：溢出设置
如果在一次算术操作之后设置了 V 标志，计算的结果不适合放入一个 32bit 目标寄存器中。

VC：溢出清除
如果清除了 V 标志，与 VS 相反。

HI：高于(无符号)
如果一次比较之后设置了 C 标志**并**清除了 Z 标志。

LS：低于或同于(无符号)
如果一次比较操作之后清除了 C 标志**或**设置了 Z 标志。

PL：正号
如果一次算术操作之后清除了 N 。出于定义‘正号’的目的，零是正数的原因是它不是负数...

MI：负号
如果一次算术操作之后设置了 N 标志。

CS：进位设置
如果一次算术操作或移位操作之后设置了 C 标志，操作的结果不能表示为 32bit 。你可以把 C 标志当作结果的第 33 位。

CC：进位清除
与 CS 相反。

GE：大于或等于(有符号)
如果一次比较之后...
设置了 N 标志**并**设置了 V 标志
或者...
清除了 N 标志**并**清除了 V 标志。

GT : 大于(有符号)

如果一次比较之后...
设置了 N 标志并设置了 V 标志
或者...
清除了 N 标志并清除了 V 标志
并且...
清除了 Z 标志。

LE : 小于或等于(有符号)

如果一次比较之后...
设置了 N 标志并清除了 V 标志
或者...
清除了 N 标志并设置了 V 标志
并且...
设置了 Z 标志。

LT : 小于(有符号)

如果一次比较之后...
设置了 N 标志并清除了 V 标志。
或者...
清除了 N 标志并设置了 V 标志。

AL : 总是

缺省条件，所以不用明显声明。

NV : 从不

不是特别有用，它表示应当永远不执行这个指令。是穷人的 NOP 。

包含 NV 是为了完整性(与 AL 相对)，你不应该在你的代码中使用它。

有一个在最后的条件代码 S，它以相反的方式工作。当用于一个指令的时候，导致更改状态标志。这不是自动发生的 - 除非这些指令的目的是设置状态。例如:

```
ADD     R0, R0, R1

ADDS    R0, R0, R1

ADDEQS  R0, R0, R1
```

第一个例子是一个基本的加法(把 R1 的值增加到 R0)，它不影响状态寄存器。

第二个例子是同一个加法，只不过它导致更改状态寄存器。

最后一个例子是同一个加法，更改状态寄存器。不同在于它是一个有条件的指令。只有前一个操作的结果是 EQ (如果设置了 Z 标志)的时候它才执行。

下面是条件执行的一个工作中的例子。你把寄存器 0 与存储在寄存器 10 中内容相比较。如果不等于 R10，则调用一个软件中断，增加它并分支回来再次做这些。否则清除 R10 并返回到调用它的那部分代码(它的地址存储在 R14)。

\ 条件执行的一个例子

```
.loop                                   ; 标记循环开始位置
CMP       R0, R10                       ; 把 R0 与 R10 相比较
SWINE     &40017                        ; 不等于: 调用 SWI &40017
ADDNE     R0, R0, #1                    ;          向 R0 加 1
BNE       loop                          ;          分支到 'loop'
MOV       R10, #0                       ; 等于   : 设置 R10 为零
LDMFD     R13!, {R0-R12,PC}             ;          返回到调用者
```

注解:

- SWI 编号就象我写的这样。在RISC OS 下，它是给 *Econet_DoImmediate* 的编号。不要字面的接受它，这只是一个例子!
- 你可能以前没见过 LDMFD，它从栈中装载多个寄存器。在这个例子中，我们从一个完全正式的栈中装载 R0 至 R12 和 R14 。关于寄存器装载和存储的更多信息请参阅 str.html。
- 我说要装载 R14 。那么为什么要把它放入 PC 中? 原因是此时 R14 存储的值包含返回地址。我们也可以采用:
  ```
  LDMFD     R13!, {R0-R12,R14}
  MOV       PC, R14
  ```
  但是直接恢复到 PC 中可以省略这个 MOV 语句。
- 最后，这些寄存器很有可能被一个 SWI 调用所占用(依赖于在调用期间执行的代码)，所以你最好把你的重要的寄存器压入栈中，以后在恢复它们。

SWI 指令

SWI : 软件中断

(Software Interrupt)

SWI{条件} <24 位编号>

指令格式

这是一个简单的设施，但可能是最常用的。多数操作系统设施是用 SWI 提供的。没有 SWI 的 RISC OS 是不可想象的。

Nava Whiteford 解释了 SWI 是如何工作的(最初在 Frobnicate issue 12)...

SWI 是什么?

SWI 表示 Software Interrupt。在RISC OS 中使用 SWI 来访问操作系统例程或第三方生产的模块。许多应用使用模块来给其他应用提供低层外部访问。

SWI 的例子有:

- 文件器 SWI，它辅助读写磁盘、设置属性等。
- 打印机驱动器 SWI，用来辅助使用打印并行端口。
- FreeNet/Acorn TCP/IP 协议栈 SWI，用TCP/IP 协议在 Internet 上发送和接收数据。

在以这种方式使用的时候，SWI 允许操作系统拥有一个模块结构，这意味着用来建立完整的操作系统的所需的代码可以被分割成许多小的部分(模块)和一个模块处理程序(handler)。

当 SWI 处理程序得到对特定的例程编号的一个请求的时候，它找到这个例程的位置并执行它，并传递(有关的)任何数据。

它是如何工作的?

首先查看一下如何使用它。一个 SWI 指令(汇编语言)看起来如下:

```
SWI &02
```

或

```
SWI "OS_Write0"
```

这些指令实际上是相同的，将被汇编成相同的指令。唯一的不同是第二个指令使用一个字符串来表示 SWI 编号 &02。在使用采用了字符串编号的程序的时候，在执行之前首先查找这个字符串。

在这里我们不想处理字符串，因为它不能给出它要进行什么的一个真实表示。它们通常用于增进一个程序的清晰程度，但不是实际执行的指令。

让我们再次看一下第一个指令:

```
SWI &02
```

这是什么意思? 字面的意思是进入 SWI 处理程序并传递值 &02。在RISC OS 中这意味着执行编号是 &02 的例程。

它是如何这么作的? 它如何传递 SWI 编号和进入 SWI 处理程序?

如果你查看内存的开始 32 字节(位于 0-&1C)并反汇编它们(查开实际的 ARM 指令)你将见到如下:

```
地址        内容                反汇编
00000000 : 0..å : E5000030 : STR     R0,[R0,#-48]
00000004 : .óŸå : E59FF31C : LDR     PC,&00000328
00000008 : .óŸå : E59FF31C : LDR     PC,&0000032C
0000000C : .óŸå : E59FF31C : LDR     PC,&00000330
00000010 : .óŸå : E59FF31C : LDR     PC,&00000334
00000014 : .óŸå : E59FF31C : LDR     PC,&00000338
00000018 : .óŸå : E59FF31C : LDR     PC,&0000033C
0000001C :  2?ã : E3A0A632 : MOV     R10,#&3200000
```

让我们仔细看一下。

除了第一个和最后一个指令之外(它们是特殊情况)你见到的都是把一个新值装载到 PC (程序计数器)的指令，它们告诉计算机到哪里去执行下一个指令。还展示了这个值是从内存中的一个地址接受来的。(你可以在 !Zap 主菜单上使用“Read Memory”选项去自己查看一下。)

这看起来好象与 SWI 没多少关系，下面做进一步的说明。

一个 SWI 所做的一切就是把模式改变成超级用户并设置 PC 来执行在地址 &08 处的下一个指令！把处理器转换到超级用户模式会切换掉两个寄存器 r13 和 r14 并用 r13_svc 和 r14_svc 替换它们。

在进入超级用户模式的时候，还把 r14_svc 设置为在这个 SWI 指令之后的地址。

这个实际上就象一个连接到地址 &08 的分支指令(BL &08)，但带有用于一些数据(SWI 编号)的空间。

象我说过的那样,地址 &08 包含跳转到另一个地址的一个指令,就是实际的 SWI 程序的地址!

此时你可能会想“稍等一会！还有 SWI 编号呢?”。实际上处理器忽略这个值本身。SWI 处理程序使用传递来的 r14_svc 的值来获取它。

下面是完成它的步骤(在存储寄存器 r0-r12 之后):

1. 它从 r14 中减去 4 来获得 SWI 指令的地址。
2. 把这个指令装载到一个寄存器。
3. 清除这个指令的高端 8 位，去掉了 OpCode 而只剩下的 SWI 编号。
4. 使用这个值来找到要被执行的代码的例程的地址(使用查找表等)。
5. 恢复寄存器 r0-r12 。
6. 使处理器离开超级用户模式。
7. 跳转到这个例程的地址。

容易吧!;)

下面是一个例子，来自 ARM610 datasheet:

```
0x08 B Supervisor

EntryTable
 DCD ZeroRtn
 DCD ReadCRtn
 DCD WriteIRtn

 ...

Zero   EQU 0
ReadC  EQU 256
WriteI EQU 512

; SWI 包含需要的例程在位 8-23 中和数据(如果有的话)在位 0-7 中。
; 假定 R13_svc 指向了一个合适的栈

STMFD R13, {r0-r2 , R14}
 ; 保存工作寄存器和返回地址。
LDR R0,[R14,#-4]
 ; 得到 SWI 指令。
BIC R0,R0, #0xFF000000
 ; 清除高端的 8 位。
MOV R1, R0, LSR #8
 ; 得到例程偏移量。
ADR R2, EntryTable
 ; 得到入口表(EntryTable)的开始地址。
LDR R15,[R2,R1,LSL #2]
 ; 分支到正确的例程

WriteIRtn
 ; 写 R0 中的位 0 - 7 中的字符。


.............
 LDMFD R13, {r0-r2 , R15}^
 ; 恢复工作空间，并返回、恢复处理器模式和标志。
```

这就是 SWI 指令的基本处理步骤。

来源:

The ARM610 datasheet by Advanced Risc Machines
The ARM RISC Chip - A programmers guide by van Someren Atack published by Addison Wesley

APCS 简介

(ARM 过程调用标准)

介绍

APCS，ARM 过程调用标准(ARM Procedure Call Standard)，提供了紧凑的编写例程的一种机制，定义的例程可以与其他例程交织在一起。最显著的一点是对这些例程来自哪里没有明确的限制。它们可以编译自 C 、 Pascal 、也可以是用汇编语言写成的。

APCS 定义了:

- 对寄存器使用的限制。
- 使用栈的惯例。
- 在函数调用之间传递/返回参数。
- 可以被'回溯'的基于栈的结构的格式，用来提供从失败点到程序入口的函数(和给予的参数)的列表。

APCS 不一个单一的给定标准，而是一系列类似但在特定条件下有所区别的标准。例如，APCS-R (用于 RISC OS)规定在函数进入时设置的标志必须在函数退出时复位。在 32 位标准下，并不是总能知道进入标志的(没有 USR_CPSR)，所以你不需要恢复它们。如你所预料的那样，在不同版本间没有相容性。希望恢复标志的代码在它们未被恢复的时候可能会表现失常...

如果你开发一个基于 ARM 的系统，不要求你去实现 APCS。但建议你实现它，因为它不难实现，且可以使你获得各种利益。但是，如果要写用来与编译后的 C 连接的汇编代码，则必须使用 APCS。编译器期望特定的条件，在你的加入(add-in)代码中必须得到满足。一个好例子是 APCS 定义 a1 到 a4 可以被破坏，而 v1

到 v6 必须被保护。现在我确信你正在挠头并自言自语“a 是什么? v 是什么?”。所以首先介绍 APCS-R 寄存器定义...

寄存器命名

APCS 对我们通常称为 R0 到 R14 的寄存器起了不同的名字。使用汇编器预处理器的功能，你可以定义 R0 等名字，但在你修改其他人写的代码的时候，最好还是学习使用 APCS 名字。

寄存器名字		
Reg #	APCS	意义
R0	a1	工作寄存器
R1	a2	"
R2	a3	"
R3	a4	"
R4	v1	必须保护
R5	v2	"
R6	v3	"
R7	v4	"
R8	v5	"
R9	v6	"
R10	sl	栈限制
R11	fp	桢指针
R12	ip	
R13	sp	栈指针
R14	lr	连接寄存器
R15	pc	程序计数器

译注：ip 是指令指针的简写。

这些名字不是由标准的 Acorn 的 objasm(版本 2.00)所定义的，但是 objasm 的后来版本，和其他汇编器(比如 Nick Robert 的 ASM)定义了它们。要定义一个寄存器名字，典型的，你要在程序最开始的地方使用 RN 宏指令(directive):

```
a1      RN      0
a2      RN      1
a3      RN      2
    ...等...
```

```
r13     RN      13
sp      RN      13
r14     RN      14
lr      RN      r14
pc      RN      15
```

这个例子展示了一些重要的东西:

1. 寄存器可以定义多个名字 - 你可以定义‘r13’和‘sp’二者。
2. 寄存器可以定义自前面定义的寄存器 - ‘lr’定义自叫做‘r14’的寄存器。
   (对于 objasm 是正确的，其他汇编器可能不是这样)

设计关键

- 函数调用应当快、小、和易于(由编译器来)优化。
- 函数应当可以妥善处理多个栈。
- 函数应当易于写可重入和可重定位的代码；主要通过把可写的数据与代码分离来实现。
- 但是最重要的是，它应当简单。这样汇编编程者可以非常容易的使用它的设施，而调试者能够非常容易的跟踪程序。

一致性

程序的遵循 APCS 的部分在调用外部函数时被称为“一致”。在程序执行期间的所有时候都遵循 APCS (典型的，由编译器生成的程序)被称为“严格一致”。协议指出，假如你遵守正确的进入和退出参数，你可以在你自己的函数范围内做你需要的任何事情，而仍然保持一致。这在有些时候是必须的，比如在写 SWI 伪装(veneers)的时候使用了许多给实际的 SWI 调用的寄存器。

栈

栈是链接起来的‘桢’的一个列表，通过一个叫做‘回溯结构’的东西来链接它们。这个结构存储在每个桢的高端。按递减地址次序分配栈的每一块。寄存器 sp 总是指向在最当前桢中最低的使用的地址。这符合传统上的满降序栈。在 APCS-R 中，寄存器 sl 持有一个栈限制，你递减 sp 不能低于它。在当前栈指针和当前栈之间，不应该有任何其他 APCS 函数所依赖的东西，在被调用的时候，函数可以为自己设置一个栈块。

可以有多个栈区(chunk)。它们可以位于内存中的任何地址，这里没有提供规范。典型的，在可重入方式下执行的时候，这将被用于为相同的代码提供多个栈；一个类比是 FileCore，它通过简单的设置‘状态’信息和并按要求调用相同部分的代码,来向当前可获得的 FileCore 文件系统(ADFS、RAMFS、IDEFS、SCSIFS 等)提供服务。

回溯结构

寄存器 fp (桢指针)应当是零或者是指向栈回溯结构的列表中的最后一个结构，提供了一种追溯程序的方式，来反向跟踪调用的函数。

回溯结构是:

地址高端

保存代码指针	[fp]	fp 指向这里	
返回 lr 值	[fp, #-4]		
返回 sp 值	[fp, #-8]		
返回 fp 值	[fp, #-12]	指向下一个结构	
[保存的 sl]			
[保存的 v6]			
[保存的 v5]			
[保存的 v4]			
[保存的 v3]			
[保存的 v2]			
[保存的 v1]			
[保存的 a4]			
[保存的 a3]			
[保存的 a2]			
[保存的 a1]			
[保存的 f7]			三个字
[保存的 f6]			三个字
[保存的 f5]			三个字
[保存的 f4]			三个字

地址低端

这个结构包含 4 至 27 个字，在方括号中的是可选的值。如果它们存在，则必须按给定的次序存在(例如，在内存中保存的 a3 下面可以是保存的 f4，但 a2-f5 则不能存在)。浮点值按‘内部格式’存储并占用三个字(12 字节)。

fp 寄存器指向当前执行的函数的栈回溯结构。返回 fp 值应当是零，或者是指向由调用了这个当前函数的函数建立的栈回溯结构的一个指针。而这个结构中的返回 fp 值是指向调用了调用了这个当前函数的函数的函数的栈回溯结构的一个指针；并以此类推直到第一个函数。

在函数退出的时候，把返回连接值、返回 sp 值、和返回 fp 值装载到 pc、sp、和 fp 中。

```
#include <stdio.h>

void one(void);
void two(void);
void zero(void);

int main(void)
{
   one();
   return 0;
}

void one(void)
{
   zero();
   two();
   return;
}

void two(void)
{
   printf("main...one...two\n");
   return;
}

void zero(void)
{
   return;
}
```

当它在屏幕上输出消息的时候，
APCS 回溯结构将是:

```
    fp ----> two_structure
             return link
```

```
                return sp
                return fp  ----> one_structure
                ...              return link
                                 return sp
                                 return fp  ----> main_structure
                                 ...              return link
                                                  return sp
                                                  return fp  ----> 0
                                                  ...
```

所以，我们可以检查 fp 并参看给函数‘two’的结构，它指向给函数‘one’的结构，它指向给‘main’的结构，它指向零来终结。在这种方式下，我们可以反向追溯整个程序并确定我们是如何到达当前的崩溃点的。值得指出‘zero’函数，因为它已经被执行并退出了，此时我们正在做它后面的打印，所以它曾经在回溯结构中，但现在不在了。值得指出的还有对于给定代码不太可能总是生成象上面那样的一个 APCS 结构。原因是不调用任何其他函数的函数不要求完全的 APCS 头部。

为了更细致的理解，下面是代码是 Norcroft C v4.00 为上述代码生成的...

```
        AREA |C$$code|, CODE, READONLY

        IMPORT  |__main|
|x$codeseg|
        B       |__main|

        DCB     &6d,&61,&69,&6e
        DCB     &00,&00,&00,&00
        DCD     &ff000008

        IMPORT  |x$stack_overflow|
        EXPORT  one
        EXPORT  main
main
        MOV     ip, sp
        STMFD   sp!, {fp,ip,lr,pc}
        SUB     fp, ip, #4
        CMPS    sp, sl
        BLLT    |x$stack_overflow|
        BL      one
        MOV     a1, #0
        LDMEA   fp, {fp,sp,pc}^

        DCB     &6f,&6e,&65,&00
        DCD     &ff000004
```

```
        EXPORT  zero
        EXPORT  two
one
        MOV     ip, sp
        STMFD   sp!, {fp,ip,lr,pc}
        SUB     fp, ip, #4
        CMPS    sp, sl
        BLLT    |x$stack_overflow|
        BL      zero
        LDMEA   fp, {fp,sp,lr}
        B       two

        IMPORT  |_printf|
two
        ADD     a1, pc, #L000060-.-8
        B       |_printf|
L000060
        DCB     &6d,&61,&69,&6e
        DCB     &2e,&2e,&2e,&6f
        DCB     &6e,&65,&2e,&2e
        DCB     &2e,&74,&77,&6f
        DCB     &0a,&00,&00,&00

zero
        MOVS    pc, lr

        AREA |C$$data|

|x$dataseg|

        END
```

这个例子不遵从 32 为体系。APCS-32 规定只是简单的说明了标志不需要被保存。所以删除 LDM 的‘^’后缀，并在函数 zero 中删除 MOVS 的‘S’后缀。则代码就与遵从 32-bit 的编译器生成的一样了。

保存代码指针包含这条设置回溯结构的指令(STMFD ...)的地址再加上 12 字节。记住，对于 26-bit 代码，你需要去除其中的 PSR 来得到实际的代码地址。

现在我们查看刚进入函数的时候:

- **pc** 总是包含下一个要被执行的指令的位置。
- **lr** (总是)包含着退出时要装载到 **pc** 中的值。在 26-bit 位代码中它还包含着 PSR 。
- **sp** 指向当前的栈块(chunk)限制，或它的上面。这是用于复制临时数据、寄存器和类似的东西到其中的地方。在 RISC OS 下，你有可选择的至少 256 字节来扩展它。

- **fp** 要么是零，要么指向回溯结构的最当前的部分。
- 函数实参布置成(下面)描述的那样。

实际参数

APCS 没有定义记录、数组、和类似的格局。这样语言可以自由的定义如何进行这些活动。但是，如果你自己的实现实际上不符合 APCS 的精神，那么将不允许来自你的编译器的代码与来自其他编译器的代码连接在一起。典型的，使用 C 语言的惯例。

- 前 4 个整数实参(或者更少!)被装载到 a1 - a4 。
- 前 4 个浮点实参(或者更少!)被装载到 f0 - f3 。
- 其他任何实参(如果有的话)存储在内存中，用进入函数时紧接在 sp 的值上面的字来指向。换句话说，其余的参数被压入栈顶。所以要想简单。最好定义接受 4 个或更少的参数的函数。

函数退出

通过把返回连接值传送到程序计数器中来退出函数，并且:

- 如果函数返回一个小于等于一个字大小的值，则把这个值放置到 a1 中。
- 如果函数返回一个浮点值，则把它放入 f0 中。
- sp、fp、sl、v1-v6、和 f4-f7 应当被恢复(如果被改动了)为包含在进入函数时它所持有的值。
  我测试了故意的破坏寄存器，而结果是(经常在程序完全不同的部分)出现不希望的和奇异的故障。
- ip、lr、a2-a4、f1-f3 和入栈的这些实参可以被破坏。

在 32 位模式下，不需要对 PSR 标志进行跨越函数调用的保护。在26 位模式下必须这样，并通过传送 lr 到 pc 中(MOVS、或 LDMFD xxx^) 来暗中恢复。必须从 lr 重新装载 N 、Z、C 和 V ，跨越函数保护这些标志不是足够的。

建立栈回溯结构

对于一个简单函数(固定个数的参数，不可重入)，你可以用下列指令建立一个栈回溯结构:

```
function_name_label
        MOV     ip, sp
        STMFD   sp!, {fp,ip,lr,pc}
        SUB     fp, ip, #4
```

这个片段(来自上述编译后的程序)是最基本的形式。如果你要破坏其他不可破坏的寄存器，则你应该在这个 STMFD 指令中包含它们。

下一个任务是检查栈空间。如果不需要很多空间(小于 256 字节)则你可以使用:

```
        CMPS    sp, sl
        BLLT    |x$stack_overflow|
```

这是 C 版本 4.00 处理溢出的方式。在以后的版本中，你要调用 |__rt_stkovf_split_small|。

接着做你自己的事情...

通过下面的指令完成退出:

```
        LDMEA   fp, {fp,sp,pc}^
```

还有，如果你入栈了其他寄存器，则也在这里重新装载它们。选择这个简单的 LDM 退出机制的原因是它比分支到一个特殊的函数退出处理器(handler)更容易和更合理。

用在回溯中的对这个协议的一个扩展是把函数名字嵌入到代码中。紧靠在函数(和 MOV ip, sp)的前面的应该是:

```
        DCD     &ff0000xx
```

这里的'xx'是函数名字符串的长度(包括填充和终结符)。这个字符串是字对齐、尾部填充的，并且应当被直接放置在 DCD &ff.... 的前面。

所以一个完整的栈回溯代码应当是:

```
        DCB     "my_function_name", 0, 0, 0, 0
        DCD     &ff000010
my_function_name
        MOV     ip, sp
        STMFD   sp!, {fp, ip, lr, pc}
        SUB     fp, ip, #4

        CMPS    sp, sl                  ; 如果你不使用栈
        BLLT    |x$stack_overflow|      ; 则可以省略

        ...处理...

        LDMEA   fp, {fp, sp, pc}^
```

要使它遵从 32-bit 体系，只须简单的省略最后一个指令的'^'。注意你不能在一个编译的 26-bit 代码中使用这个代码。实际上，你可以去除它，但这不是我愿意打赌的事情。

如果你不使用栈，并且你不需要保存任何寄存器，并且你不调用任何东西，则没有必要设置 APCS 块(但在调试阶段对跟踪问题仍是有用的)。在这种情况下你可以:

```
my_simple_function

        ...处理...

        MOVS    pc, lr
```

(再次，对 32 位 APCS 使用 MOV 而不是 MOVS，但是不要冒险与 26 位代码连接)。

APCS 标准

总的来说，有多个版本的 APCS (实际上是 16 个)。我们只关心在 RISC OS 上可能遇到的。

APCS-A

就是 APCS-Arthur；由早期的 Arthur 所定义。它已经被废弃，原因是它有不同的寄存器定义(对于熟练的 RISC OS 程序员它是某种异类)。它用于在 USR 模式下运行的 Arthur 应用程序。不应该使用它。

- sl = R13, fp = R10, ip = R11, sp = R12, lr = R14, pc = R15。
- PRM (p4-411) 中说“用 r12 作为 sp，而不是在体系上更自然的 r13，是历史性的并先于 Arthur 和 RISC OS 二者。”
- 栈是分段的并可按需要来扩展。
- 26-bit 程序计数器。
- 不在 FP 寄存器中传递浮点实参。
- 不可重入。标志必须被恢复。

APCS-R

就是 APCS-RISC OS。用于 RISC OS 应用程序在 USR 模式下进行操作；或在 SVC 模式下的模块/处理程序。

- sl = R10, fp = R11, ip = R12, sp = R13, lr = R14, pc = R15。
- 它是唯一的最通用的 APCS 版本。因为所有编译的 C 程序都使用 APCS-R 。
- 显式的栈限制检查。
- 26-bit 程序计数器。
- 不在 FP 寄存器中传递浮点实参。
- 不可重入。标志必须被恢复。

APCS-U

就是 APCS-Unix，Acorn 的 RISCiX 使用它。它用于 RISCiX 应用程序(USR 模式)或内核(SVC 模式)。

- sl = R10, fp = R11, ip = R12, sp = R13, lr = R14, pc = R15。
- 隐式的栈限制检查(使用 sl)。
- 26-bit 程序计数器。
- 不在 FP 寄存器中传递浮点实参。
- 不可重入。标志必须被恢复。

APCS-32

它是 APCS-2(-R 和 -U)的一个扩展，允许 32-bit 程序计数器，并且从执行在 USR 模式下的一个函数中退出时，允许标志不被恢复。其他事情同于 APCS-R。Acorn C 版本 5 支持生成 32-bit 代码；在用于广域调试的 32 位工具中，它是最完整的开发发行。一个简单的测试是要求你的编译器导出汇编源码(而不是制作目标代码)。你不应该找到:

```
MOVS PC, R14
```

或者

```
LDMFD R13!, {Rx-x, PC}^
```

对编码有用的东西

首先要考虑的是该死的 26/32 位问题。 简单的说，不转弯抹角绝对没有方法为两个版本的 APCS 汇编同一个通用代码。但是幸运的这不是问题。APCS 标准不会突然改变。RISC OS 的 32 位版本也不会立刻变异。所以利用这些，我们可以设计一种支持两种版本的方案。这将远远超出 APCS，对于 RISC OS 的 32 位版本你需要使用 MSR 来处理状态和模式位，而不是使用 TEQP。许多现存的 API 实际上不需要保护标志位。所以在我们的 32 版本中可以通过把 MOVS PC,... 变成 MOV PC,...，和把 LDM {...}^ 变成 LDM {...}，并重新建造来解决。objasm 汇编器(v3.00 和以后)有一个 {CONFIG} 变量可以是 26 或 32。可以使用它建造宏...

```
my_function_name
        MOV     ip, sp
        STMFD   sp!, {fp, ip, lr, pc}
        SUB     fp, ip, #4

        ...处理...

        [ {CONFIG} = 26
          LDMEA   fp, {fp, sp, pc}^
        |
          LDMEA   fp, {fp, sp, pc}
        ]
```

我未测试这个代码。它(或类似的东西)好象是保持与两个版本的 APCS 相兼容的最佳方式，也是对 RISC OS 的不同版本，26 位版本和将来的 32 位版本的最佳方法。

测试是否处于 32 位? 如果你要求你的代码有适应性，有一个最简单的方法来确定处理器的 PC 状态：

```
TEQ     PC, PC      ; 对于 32 位是 EQ；对于 26 位是 NE
```

使用它你可以确定:

- 26 位 PC，可能是 APCS-R 或 APCS-32。
- 32 位 PC，不能 APCS-R。所有 26-bit 代码(TEQP 等)面临着失败!
-

32 位规则

- 不要使用有 P 后缀的测试指令: TEQP、TSTP、CMPP、CMNP。
- 检查更改 PC、R14 的如 BIC 和 ORR 这样的指令，和把一个寄存器复制到 PC 的指令。例如，ORRS PC, R14, #1<<28 将不能工作。实际上，不要使用带 S 标志设置的写到 PC 的任何指令。
- 在数据处理操作中不要使用 R15 (PC) 作为移位寄存器。
- 在 LDR/STR 中，不要使用 PC 作为寄存器偏移量并不要写回到它。
- 在过后变址 LDR/STR 中，Rm(变址)和 Rn(基址)不能是同一个寄存器。类似的，对于涉及写回的任何指令，Rm 和 Rn 都应该是不同的寄存器。
- LDM/STM 在用户模式下不使用 S 位。这意味着，不要使用'^'后缀(例如 LDMFD R13!, {PC}^)。
- BL 不保存状态寄存器。这必须显式的进行，但是这样的代码将不能在 ARM2 或 ARM3 上运行，因为它们不支持 MRS/MSR 指令。
- 还要注意，**不可能**见到被调用者的标志，所以不应该恢复你不知道其状态的标志。你能做的最好的就是在进入的时候保护标志。

任何带 S 位设置的到 R15 的 32-bit 写(MOVS、ORRS、TEQP、LDM...^) 将传送当前模式的 SPSR 到 CPSR 中。例如，假定我们在 irq_32 模式下:

```
MOVS PC, R14
```

将复制 R14 到 PC，并接着复制 SPSR_IRQ32 到 CPSR。

这在 USR 模式下不是非常有用因为它没有 SPSR!

浮点指令

指令索引

STF	存储浮点值
SUF	减法
TAN	正切
URD	非规格化舍入
WFC	写 FP 控制寄存器
WFS	写 FP 状态寄存器

本文档部分内容取自 ARM 汇编器手册。

ARM 可以与最多 16 个协处理器相接口(interface)。ARM3 和以后的处理器在 ARM 内有虚拟的协处理器来处理内部控制功能。而可获得的第一个协处理器是浮点处理器。这个芯片处理 IEEE 标准的浮点运算。定义了一个标准的 ARM 浮点指令集，所以编码可以跨越所有 RISC OS 机器。如果不存在实际的硬件，则这些指令被截获并由浮点模拟器模块(FPEmulator)来执行。程序不需要知道是否存在 FP 协处理器。唯一不同的是执行速度。

RISC OS 的 BASIC 汇编器，作为标准，不支持任何真实的浮点指令。你可以转换整数到你的实现定义的‘浮点’并用它们进行(最普通的定点)基本数学运算，但你不能与浮点协处理器交互并以‘固有的’方式来做这些事情。但是，扩展汇编器功能的补丁中包含了 FP 指令。

ARM IEEE FP 系统由 8 个高精度 FP 寄存器(F0 到 F7)。寄存器的格式是无关紧要的，因为你不能直接访问这些寄存器，寄存器只在它被传送到内存或 ARM 寄存器时是‘可见的’。在内存中，一个 FP 寄存器占用三个字，但因为 FP 系统把它重新装载到自己的寄存器中，这三个字的格式是无关紧要的。还有一个 FPSR (浮点状态寄存器)，它类似于 ARM 自己的 PSR，持有应用程序可能需要的状态信息。可获得的每个标志都有一个‘陷阱’，这允许应用程序来启用或禁用与给定错误关联的陷阱。FPSR 还允许你得知在 FP 系统得不同实现之间的区别。还有一个 FPCR (浮点控制寄存器)。它持有应用程序不应该访问的信息，比如开启和关闭 FP 单元的标志。典型的，硬件有 FPCR 而软件没有。

FP 单元可以软件实现比如 FPEmulator 模块，硬件实现比如 FP 芯片(和支持代码)，或二者的组合。二者的最好的例子是 Warm Silence Software 补丁，它允许 ARM FP 操作利用配备在 PC 协处理器卡上的 80x87 作为作为一个浮点协处理器。

计算的结果如同有无限的精度，接着被舍入成要求的精度。舍入方式有就近舍入，向正无穷(P)舍入，向负无穷舍入(M)，或向零舍入。缺省的是就近舍入。如果不可抉择，则舍入到最近似的偶数。工作精度是 80 位，其组成是 64 位尾数，15 位指数，和一个符号位。在一些实现中对用单精度工作的指令提供了更好的性能 - 特别是完全基于软件的那些实现。

FPSR 包含 FP 系统所需的状态。总是提供 IEEE 标志，但只在一次 FP 比较操作之后才可获得结果标志。

浮点指令不应该用在 SVC 模式下。

FPSR 的低字节是例外标志字节。

	6	4	3	2	1	0
FPSR:	保留	INX	UFL	OFL	DVZ	IVO

当引发一个例外条件的时候，把在位 0 到 4 中的适当的累计(cumulative)例外标志设置为 1。如果设置了相关的陷阱位，则按操作系统指定的方式把一个例外递送给用户程序。(注意在下溢的情况下，陷阱启用位的状态决定在什么条件下设置下溢标志。) 只能用 WFS 指令清除这些标志。

IVO - invalid operation 无效操作
在进行的操作的一个操作数是无效时设置 IVO。无效操作有:

- 在一个捕获(trapping)的 NaN (not-a-number：非数)上进行任何操作。
- 无穷大幅值(magnitude)相减，例如 (+∞) + (-∞)。
- 乘法 0 * ∞。
- 除法 ∞/∞ 或 x/0 。
- x REM y 这里 x = ∞ 或 y = 0 。
  (REM 是浮点除法操作的余数。)
- 任何小于 0 的数的平方根。
- 在上溢或操作数是 NaN 的时候进行转换成整数或十进制数。
  如果上溢使转换不可能，则生成最大的正或负整数(依赖于操作数的符号)并通知(signal)一个 IVO 。
- 比较时有未对阶(Unordered)操作数例外。
- ACS、ASN、SIN、COS、TAN、LOG、LGN、POW、或 RPW 有无效/错误的参数。

DVZ - division by zero 除零
如果除数是零而被除数是一个有限的、非零的数则设置 DVZ 标志。如果禁用了陷阱则返回一个正确的有符号的无穷。还为 LOG(0) 和 LGN(0) 设置这个标志。如果禁用了陷阱则返回负无穷。

OFL - overflow 上溢
结果幅值超出目的格式最大的数的时候设置 OFL 标志，舍入的结果是指数范围无限大的(unbounded)。 因为在结果被舍入之后检测上溢，在一些操作之后是否发生上溢依赖于舍入模式。如果禁用了陷阱，要么返回一个有正确符号的无穷，要么返回这个格式的最大的有限数。这依赖于舍入模式和使用的浮点系统。

UFL - underflow 下溢
两个有关联的事件产生下溢:

- 极小值(tininess) - 微小的非零结果在幅值上小于这个格式的最小规格化数。

- 准确性损失 - 反规格化导致的准确性损失可能大于单独舍入导致的准确性损失。

依赖于 UFL 陷阱启用位的值，以不同的方式设置 UFL 标志。如果启用了陷阱，则不管是否有准确性损失极，在检测到极小值时就设置 UFL 标志。如果禁用了陷阱，则在检测到极小值和准确性损失二者时设置 UFL 标志(在这种情况下还设置 INX 标志)；否则返回一个有正确符号的零。因为在结果被舍入之后检测下溢，在一些操作之后是否发生下溢依赖于舍入模式。

INX - inexact 不精确
如果操作的舍入的结果是不精确的(不同于可用无穷精度计算的值)，或者在禁用 OFL 陷阱时发生上溢，或者在禁用 UFL 陷阱时发生了下溢，则设置 INX 标志。OFL 或 UFL 陷阱优先于 INX。在计算 SIN 或 COS 的时候也设置 INX 标志，但 SIN(0) 和 COS(1) 例外。老的 FPE 和 FPPC 系统在处理 INX 标志上可能不同。由于这个不一致性，我建议你不要启用 INX 陷阱。

精度:

- S - 单精度
- D - 双精度
- E - 双扩展精度
- P - 压缩(packed)十进制数
- EP - 扩展压缩十进制数

舍入模式:

- - 最近(不需要字符)
- P - 正无穷
- M - 负无穷
- Z - 零

LDF{条件}<精度> <fp 寄存器>, <地址>
装载浮点值。
地址可以是下列形式:

- [Rn]
- [Rn], #offset
- [Rn, #offset]
- [Rn, #offset]!

这个调用类似于 LDR 。
你的汇编器可能允许使用如下文字：LDFS F0, [浮点值]

STF{条件}<精度> <fp 寄存器>, <地址>

存储浮点值。
地址可以是下列形式:

- [Rn]
- [Rn], #offset
- [Rn, #offset]
- [Rn, #offset]!

这个调用类似于 STR 。
你的汇编器可能允许使用如下文字：STFED F0, [浮点值]

LFM and SFM
它们类似于 LDM 和 STM，但因为一些版本的 FPEmulator 不支持它们就不进行描述了。最新版本的 RISC OS 3.1x (2.87) 中的 FP 模块支持。如果你想让你的软件只在支持 SFM 的系统上操作就使用它吧。否则你需要用 STF 的一个序列来‘伪造’它。LFM/LDF 也是类似。

FLT{条件}<精度>{舍入} <fp 寄存器>, <寄存器>
FLT{条件}<精度>{舍入} <fp 寄存器>, #<值>
转换整数成浮点数，要么从一个 ARM 寄存器要么从一个绝对值。

FIX{条件}{舍入} <寄存器>, <fp 寄存器>
转换浮点数成整数。

WFS{条件} <寄存器>
用指定 ARM 寄存器的内容写浮点状态寄存器。

RFS{条件} <寄存器>
读浮点状态寄存器到指定的 ARM 寄存器中。

WFC{条件} <寄存器>
用指定 ARM 寄存器的内容写浮点控制寄存器。
专属超级用户模式，并只存在于支持它的硬件上。

RFC{条件} <寄存器>
读浮点控制寄存器到指定的 ARM 寄存器中。
专属超级用户模式，并只存在于支持它的硬件上。

浮点协处理器数据操作指令的格式是:

双目操作{条件}<精度>{舍入} <目的浮点寄存器>, <源浮点寄存器>, <源浮点寄存器>
双目操作{条件}<精度>{舍入} <目的浮点寄存器>, <源浮点寄存器>, #<值>
单目操作{条件}<精度>{舍入} <目的浮点寄存器>, <源浮点寄存器>
单目操作{条件}<精度>{舍入} <目的浮点寄存器>, #<值>

<值>常量应当是 0、1、2、3、4、5、10、或 0.5。

双目操作有...
ADF - 加法
DVF - 除法
FDV - 快速除法 - 只定义用单精度工作
FML - 快速乘法 - 只定义用单精度工作
FRD - 快速反向除法 - 只定义用单精度工作
MUF - 乘法
POL - 极化角
POW - 幂
RDF - 反向除法
RMF - 余数
RPW - 反向幂
RSF - 反向减法
SUF - 减法

单目操作有...
ABS - 绝对值
ACS - 反余弦
ASN - 反正弦
ATN - 反正切
COS - 余弦
EXP - 指数
LOG - 常用对数
LGN - 自然对数
MVF - 传送
MNF - 传送取负的值
NRM - 规格化
RND - 舍入到整数值
SIN - 正弦
SQT - 平方根
TAN - 正切
URD - 非规格化舍入

CMF{条件}<精度>{舍入} <fp 寄存器 1>, <fp 寄存器 2>
把 FP 寄存器 2 与 FP 寄存器 1 进行比较。
变体 CMFE 比较带有例外。

CNF{条件}<精度>{舍入} <fp 寄存器 1>, <fp 寄存器 2>
把 FP 寄存器 2 与 FP 寄存器 1 取负的值进行比较。
变体 CNFE 比较带有例外。

提供带例外和不带例外的比较，如果操作数是未对阶的(就是说它们中的一个或两个是非数)时可以引发这个例外。为了遵守 IEEE 754，CMF 指令只应用于测试等同(就是说以后使用 BEQ 或 BNE) 或测试未对阶(在 V 标志中)。应当对所有其他测试使用 CMFE 指令(以后用 BGT、BGE、BLT、BLE)。

当 FPSR 中的 AC 位清除了的时候，在比较之后，ARM 标志 N、Z、C、V 表示:
N = 小于
Z = 等于
C = 大于等于
V = 未对阶

当 FPSR 中的 AC 位设置了的时候，在比较之后，这些标志表示:
N = 小于
Z = 等于
C = 大于等于或未对阶
V = 未对阶

在使用 objasm 的 APCS 代码中，要存储一个浮点值，你可以使用宏指令(directive) DCF。对单精度添加‘S’，对双精度添加‘D’。

伪指令

RISC OS 的 BASIC 汇编器提供了一组伪指令。它们不是处理器实际上能理解的指令，但可以转换成它能理解的某种东西。它们的存在能使你的程序更加简单。

ADR：装载地址

(load Address)

ADR{后缀} <寄存器>, <标号>

它把参照的地址装载到给定寄存器中:

```
   00008FE4                     OPT     1%
   00008FE4 E28F0004            ADR     R0, text
   00008FE8 EF000002            SWI     "OS_Write0"
   00008FEC E1A0F00E            MOV     PC, R14
   00008FF0                     .text
   00008FF0                     EQUS    "Hello!" + CHR$13 + CHR$10 +
CHR$0
   00008FFC                     ALIGN
```

下列代码有完全相同的效果:

```
   00008FE4                     OPT     1%
   00008FE4 E28F0004            ADD     R0, R15, #4
   00008FE8 EF000002            SWI     "OS_Write0"
   00008FEC E1A0F00E            MOV     PC, R14
   00008FF0                     .text
   00008FF0                     EQUS    "Hello!" + CHR$13 + CHR$10 +
CHR$0
   00008FFC                     ALIGN
```

实际上，它们的反汇编将显示:

```
   *MemoryI 8FE4 +18
   00008FE4 :  E28F0004 : ..忏 : ADR     R0,&00008FF0
   00008FE8 :  EF000002 : ...? : SWI     "OS_Write0"
   00008FEC :  E1A0F00E : .馋? : MOV     PC,R14
   00008FF0 :  6C6C6548 : Hell : STCVSTL CP5,C6,[R12],#-&120 ; =288
   00008FF4 :  0A0D216F : o!.. : BEQ     &003515B8
   00008FF8 :  00000000 : .... : DCD     &00000000
```

ADR 是一个很有用的指令，你不需要关心相对 R15 的偏移量(例如，我们为什么只加 4?)，也不需要在一块代码上计算偏移量。可以简单的使用 ADR Rx, label 而汇编器将设法为你使用 ADD、SUB、MOV 或 MVN 中最恰当的那个指令。限制因素是你的引用范围只能是在 4096 字节中(不完全是真的，它典型的对 ADD 或 SUB 使用被循环右移的立即值，但是为了参数的一致性，我们假定范围是 4K)。

ADRL：装载长地址

(load Address Long)

ADRL{后缀} <寄存器>, <标号>

BASIC 汇编器不支持它，但一些扩展支持它。

ADRL 指令使用 ADR 和 ADD，或 ADR 和 SUB 的一个组合，来生成一个更广大的可以到达的地址范围。但是它总是使用两个指令，所以可以尝试更加可运做的布置来重新组织你的那些可以使用普通的 ADR 代码。

还有，在一些汇编器中，用使用三个指令的 ADRX 来定位更大的地址。

ALIGN：对齐指针

(ALIGN pointers)

ALIGN

ALIGN 指令设置 P%(如果需要的话还有 O%)来在一个字边界上对齐。通常要求它跟随着一个字符串或者一个或多个字节的数据，并切应当在更远的代码被汇编之前使用它。

BASIC 汇编器非常聪明并且有经验，如果你疏忽了，它能为你处理对齐问题...

```
00008FF4                        OPT     1%
00008FF4 E28F0004               ADR     R0, text
00008FF8 EF000002               SWI     "OS_Write0"
00008FFC EA000004               B       carryon
00009000                        .text
00009000                        EQUS    "unaligned text!!!" + CHR$0
00009012                        .carryon
00009014 E1A0F00E               MOV     PC, R14
```

DCx：初始化数据存储

DCx <值>

没有 DCx 指令。小'x'表示一个可能的范围。它们是:

DCB 预备一个字节(8 位值)
DCW 预备一个半字(16 位值)
DCD 预备一个字(32 位值)

DCS　　按给出的字符串的要求预备直到 255 个的字符

例如:

```
.start_counter
  DCB     1

.pointer
  DCD     0

.error_block
  DCD     17
  DCS     "Uh-oh! It all went wrong!" + CHR$0
  ALIGN
```

EQUx：初始化数据存储

EQUx〈值〉

没有 EQUx 指令，小'x'表示一个可能的范围。它们是:

EQUB　预备一个字节(8 位值)
EQUW　预备一个半字(16 位值)
EQUD　预备一个字(32 位值)
EQUS　按给出的字符串的要求预备直到 255 个的字符

简单的理解，除了名字不同之外与(上面的) DCx 完全一样。你可以使用'='作为 EQUB 的简写。

OPT：设置汇编器选项

(set assembler Options)

OPT〈值〉

它设置各种汇编器选项。

ARM 指令格式和时序

在整个文档中，‘字’指的是 32 位(4 字节)的内存。

目录

处理器模式

ARM 有一个用户模式和多个有特权的超级用户模式。它们是:

IRQ
: 在触发中断请求(IRQ)时进入。

FIQ
: 在触发快速中断请求(FIQ)时进入。

SVC
: 在指令一个软件中断(SWI)时进入。

Undef

在执行了一个未定义的指令时进入(不存在于 ARM 2 和 3，在这里进入 SVC 模式)。

Abt

在一个内存访问尝试被内存管理器(例如，MEMC 或 MMU) 所终止时进入，通常因为所做的尝试要访问不存在的内存或者在没有充足特权的模式下访问内存(不存在于 ARM 2 和 3，在这里进入 SVC 模式)。

在每种情况下还调用适当的硬件向量。

寄存器

ARM 2 和 3 有 27 个 32 位处理器寄存器，在任何给定时间只有其中的 16 个是可见的(是哪十六个取决于处理器模式)。它们被引用为 R0-R15。

ARM 6 和以后有 31 个 32 位处理器寄存器，在任何给定时间只有其中的 16 个是可见的。

R15 特别重要。在 ARM 2 和 3，其中的 24 位用做程序计数器，而余下的 8 位用于保持处理器模式、状态标志和中断模式。所以 R15 经常被称做 PC。

R15 = PC = NZCVIFpp pppppppp pppppppp ppppppMM

位 0-1 和 26-31 被称为 PSR (处理器状态寄存器)。位 2-25 给出被取回到指令流水线中的当前指令的(以字为单位)地址 (见后)。所以永远只能从字对齐的地址执行指令。

M 当前处理器模式

0 用户模式
1 快速中断处理模式(FIQ 模式)
2 中断处理模式(IRQ 模式)
3 超级用户模式(SVC 模式)

名字 意思

N 负数(Negative)标志
Z 零(Zero)标志
C 进位(Carry)标志
V 溢出(oVerflow)标志
I 中断(Interrupt)请求禁用
F 快速(Fast)中断请求禁用

R14、R14_FIQ、R14_IRQ、和 R14_SVC 由于它们在带连接的分支指令期间的行为而有时被称为‘连接’寄存器。

ARM 6 和以后的处理器核心支持 32 位地址空间。这些处理可以在 26 为和 32 位 PC 模式二者下操作。 在 26 位 PC 模式下，R15 表现如同在以前的处理器上，所以代码只能运行在地址空间的最低的 64M 字节中。在 32 位 PC 模式下，R15 所有 32 位用做程序计数器。使用独立的状态寄存器来存储处理器模式和状态标志。PSR 定义如下:

NZCVxxxx xxxxxxxx xxxxxxxx IFxMMMMM

注意在 32-bit 模式下 R15 的底端两位总是零 - 就是说你仍然只能得到字对齐的指令。忽略对这两位写非零的任何尝试。

当前定义了下列模式:

M	名字	意思
00000	usr_26	26 位 PC Usr 模式
00001	fiq_26	26 位 PC FIQ 模式
00010	irq_26	26 位 PC IRQ 模式
00011	svc_26	26 位 PC SVC 模式
10000	usr_32	32 位 PC Usr 模式
10001	fiq_32	32 位 PC FIQ 模式
10010	irq_32	32 位 PC IRQ 模式
10011	svc_32	32 位 PC SVC 模式
10111	abt_32	32 位 PC Abt 模式
11011	und_32	32 位 PC Und 模式

推测自上面的表，可能期望还定义了下列两个模式:

M	名字	意思
00111	abt_26	26 bit PC Abt Mode
01011	und_26	26 bit PC Und Mode

实际上未定义它们(如果你确实向模式位写了 00111 或 01011，结果的芯片状态不会是你所希望的 - 就是说不会是有适当的 R13 和 R14 被交换进来的一个 26-bit 特权模式。

下表展示在每个处理器模式下可获得那些的寄存器:

模式	可获得的寄存器																			
USR	R0	-	-	-	-	-	-	-	-	-	-	-	-	-	R14	R15

```
| FIQ  | R0 - R7 | R8_FIQ    -    R14_FIQ  R15 |
+------+---------+----+-----------------------+
| IRQ  | R0   -   R12 | R13_IRQ - R14_IRQ  R15 |
+------+-------------+-----------------------+
| SVC  | R0   -   R12 | R13_SVC - R14_SVC  R15 |
+------+-------------+-----------------------+
| ABT  | R0   -   R12 | R13_ABT - R14_ABT  R15 | (ARM 6和以后)
+------+-------------+-----------------------+
| UND  | R0   -   R12 | R13_UND - R14_UND  R15 | (ARM 6和以后)
+------+-------------------------------------+
```

在 ARM6 和以后的处理器上有六个状态寄存器。一个是当前处理器状态寄存器(CPSR)，持有关于当前处理器状态的信息。其它五个是保存的程序状态寄存器(SPSR)：每个特权模式都有一个，持有完成在这个模式下的例外处理时处理器必须返回的关于状态的信息。

分别使用 MSR 和 MRS 指令来设置和读取这些寄存器。

流水线

不同于微编码的处理器，ARM (保持它的 RISC 性)是完全硬布线的。

为了加速 ARM 2 和 3 的执行使用 3 阶段流水线。第一阶段持有从内存中取回的指令。第二阶段开始解码，而第三阶段实际执行它。故此，程序计数器总是超出当前执行的指令两个指令。(在为分支指令计算偏移量时必须计算在内)。

因为有这个流水线，在分支时丢失 2 个指令周期(因为要重新添满流水线)。所以最好利用条件执行指令来避免浪费周期。例如:

```
        ...
        CMP R0,#0
        BEQ over
        MOV R1,#1
        MOV R2,#2
over
        ...
```

可以写为更有效的:

```
        ...
        CMP R0,#0
```

```
MOVNE R1,#1
MOVNE R2,#2
...
```

时序

ARM 指令在时序上是 S、N、I 和 C 周期的混合。

S 周期是 ARM 在其中访问一个顺序的内存位置的周期。

N 周期是 ARM 在其中访问一个非顺序的内存位置的周期。

I 周期是 ARM 在其中不尝试访问一个内存位置或传送一个字到/从一个协处理器的周期。

C 周期是 ARM 在其中与一个协处理器之间在数据总线(对于无缓存的 ARM)或协处理器总线(对于有缓存的 ARM)上写传送一个字的周期。

各种类型的周期都必须至少与 ARM 的时钟周期一样长。内存系统可以伸展它们:对于典型的 DRAM 系统，结果是:

- N 周期变成最小长度的两倍(主要因为 DRAM 在内存访问是非顺序时要求更长的访问协议)。
- S 周期通常是最小长度，但偶尔也会被伸展成 N 周期的长度(在你从一个内存“行”的最后一个字移动到下一行的第一个字的时候[1])。
- I 周期和 C 周期总是最小长度。

对于典型的 SRAM 系统，所有类型的周期典型的都是最小长度。

在 Acorn Archimedes A440/1 使用的 8MHz ARM2 中，一个 S (顺序) 周期是 125ns 而一个 N (非顺序) 周期是 250ns。应当注意到这些时序不是 ARM 的属性，而是内存系统的属性。例如，一个 8MHz ARM2 可以与一个给出 125ns 的 N 周期的 RAM 系统相连接。处理器的速率是 8MHz 只是简单的意味着如果你使任何类型的周期在长度上小于 125ns 则它不保证能够工作。

有缓存的处理器: 所有给出的信息依据 ARM 所见到的时钟周期。它们不按固定的速率发生: 缓存控制逻辑在 cache 不中的时候改变提供给 ARM 的时钟周期来源。

典型的，有缓存的 ARM 有两个时钟输入: “快速时钟” FCLK 和“内存时钟”MCLK。 在 cache 命中的时候，ARM 的时钟使用 FCLK 的速度并且所有类型

的周期都是最小的长度：从这点上看 cache 在效果上是某种 SRAM。在 cache 不中发生的时候，ARM 的时钟同步为 MCLK，接着以 MCLK 速度进行 cache 行添充(依赖于在处理器中涉及的 cache 行的长度使用 N+3S 或 N+7S 个周期)，接着 ARM 的时钟被同步回到 FCLK。

在发生内存访问的时候，ARM 将守时操作(be clocked)：但是，可以使用一个叫 NWAIT 的输入来导致涉及到的 ARM 周期不做任何事情，直到正确的字从内存中到来，并在仍有余下的字到来的时候通常不做任何事情(为了避免在 cache 仍忙于重新填充 cache 行的时候得到进一步的内存请求)。有缓存的 ARM 可以被配置成使用 FCLK 和 MCLK 来相互同步(所以 FCLK 是准确的 MCLK 倍数，并且每个 MCLK 时钟周期与一个 FCLK 周期同时开始)或异步的(这种情况下 FCLK 和 MCLK 周期相互之间可以有任何关系)使情况更加复杂。

情况非常复杂。这些行为的近似的描述是，在一个 cache 行不中发生的时候，它所涉及的周期耗用以 MCLK 周期为单位的 cache 行重填充时间(例如，N+3S 或 N+7S)，对于 N 周期和 S 周期可能按 DRAM 所描述的那样被伸展，加上一些更多的周期用于重新同步阶段。要得到详情，你需要得到所涉及的处理器的 datasheet。

脚注 1：内存控制器意图使用这个简单的策略：如果请求一个 N 周期，则把访问作为不在同一行来对待；如果请求一个 S 周期，除非它效果上是这行的最后一个字(可以被快速检测出来)，否则把访问作为同行来对待。结果是一些 S 周期将持续与 N 周期相同的时间；如果我记得正确，在 Archimedes 上 S 周期所访问的内存被按 16 字节来分开。对于 Archimedes 代码的实际后果是：(a) 大约 4 个 S 周期中的 1 个变成一个 N 周期，为此，所有地址都是字地址并按 4 来分开；(b) 有时值得仔细关照对齐代码来避免这种效果并得到一些额外的性能。)

指令

每个 ARM 指令都是 32 位宽，下面给出详细的解释。对于每个指令类，我们给出指令位图(bitmap)，和典型汇编器使用的语法的例子。

一定要注意助记符的语法不是固定的；它是汇编器的特性，而不是 ARM 机器编码的。

条件代码

每个指令的顶端部分是一个条件代码，所以可以有条件的运行每个单一的 ARM 指令。

指令位图	条件编号	条件代码	所须标志:
0000xxxx xxxxxxxx xxxxxxxx xxxxxxxx	0	EQ(等于，Equal)	Z
0001xxxx xxxxxxxx xxxxxxxx xxxxxxxx	1	NE(不等于，Not Equal)	~Z
0010xxxx xxxxxxxx xxxxxxxx xxxxxxxx	2	CS(进位设置，Carry Set)	C
0011xxxx xxxxxxxx xxxxxxxx xxxxxxxx	3	CC(进位清除，Carry Clear)	~C
0100xxxx xxxxxxxx xxxxxxxx xxxxxxxx	4	MI(负号，MInus)	N
0101xxxx xxxxxxxx xxxxxxxx xxxxxxxx	5	PL(正号，PLus)	~N
0110xxxx xxxxxxxx xxxxxxxx xxxxxxxx	6	VS(溢出设置，oVerflow Set)	V
0111xxxx xxxxxxxx xxxxxxxx xxxxxxxx	7	VC(溢出清除，oVerflow Clear)	~V
1000xxxx xxxxxxxx xxxxxxxx xxxxxxxx	8	HI(高于，HIgher)	C and ~Z
1001xxxx xxxxxxxx xxxxxxxx xxxxxxxx	9	LS(低于或同于，Lower or Same)	~C and Z
1010xxxx xxxxxxxx xxxxxxxx xxxxxxxx	A	GE(大于等于，Greater or equal)	N = V
1011xxxx xxxxxxxx xxxxxxxx xxxxxxxx	B	LT(小于，Less Than)	N = ~V
1100xxxx xxxxxxxx xxxxxxxx xxxxxxxx	C	GT(大于，Greater Than)	(N = V) and ~Z
1101xxxx xxxxxxxx xxxxxxxx xxxxxxxx	D	LE(小于等于,Less or equal)	(N = ~V) or Z
1110xxxx xxxxxxxx xxxxxxxx xxxxxxxx	E	AL(总是，Always)	永真
1111xxxx xxxxxxxx xxxxxxxx xxxxxxxx	F	NV(从不，Never)	永假

在多数汇编器中，插入条件代码到紧随在助记符根代码(stub)的后面；省略条件代码缺省为使用 AL。

在一些汇编器中把 HS (高于或同于) 和 LO (低于) 分别用做 CS 和 CC 的同义词。

条件 GT、GE、LT、LE 被成为有符号比较，而 HS、HI、LS、LO 被称为无符号比较。

把一个条件代码与 1 进行异或得到相反的条件的代码。

NB: ARM 废弃使用 NV 条件代码 - 假定你使用 MOV R0,R0 作为一个空指令而不是以前推荐的 MOVNV R0,R0 。将来的处理器可能重新使用 NV 条件来做其他事情。

所须条件为假的指令执行 1S 周期，使一个指令有条件执行不招致时间处罚。

数据处理指令

```
xxxx000a aaaSnnnn ddddcccc ctttmmmm  寄存器形式
xxxx001a aaaSnnnn ddddrrrr bbbbbbbb  立即数形式
```

典型的汇编语法:

```
MOV     Rd, #0
ADDEQS Rd, Rn, Rm, ASL Rc
ANDEQ   Rd, Rn, Rm
TEQP    Pn, #&80000000
CMP     Rn, Rm
```

在操作 a 下，组合 Rn 的内容和 Op2，放置结果到 Rd 中。

如果使用寄存器形式，则 Op2 被设置为依据下面描述的 t 来移位的 Rm 的内容。如果使用立即数形式，则 Op2 = #b, ROR #2r。

t	汇编器	解释
000	LSL #c	逻辑左移
001	LSL Rc	逻辑左移
010	LSR #c for c != 0	逻辑右移
	LSR #32 for c = 0	
011	LSR Rc	逻辑右移
100	ASR #c for c != 0	算术右移
	ASR #32 for c = 0	
101	ASR Rc	算术右移

110	ROR #c	for c != 0	循环右移
	RRX	for c = 0	带扩展的循环右移一位
111	ROR Rc		循环右移

在寄存器形式中，用位 8-11 表示 Rc；如果使用 Rc 则位 7 必须清除。(如果你编码为 1，你将得到一个乘法、SWP 或未分配的指令而不是一个数据处理指令。)

还有，只使用了 Rc 的底端字节 - 如果 Rc = 256, 则移位将是零。

“MOV[S] Ra,Rb,RLX” 可以通过 ADC[S] Ra,Rb,Rb 来完成，这里的 RLX 意思是带扩展的循环左移一位。

多数汇编器允许使用 ASL 作为 LSL 的同义词。因为对算术左移是什么有不同的意见，最好使用术语 LSL。

通过在 MOV、MVN 或逻辑指令中设置 S 位，(在寄存器或立即数形式中)把进位标志设置为最后移出的那一位。

如果不做移位，则不影响进位标志。

如果立即数有可选择的多个形式(例如，#1 可以表示为 1 ROR #0、4 ROR #2、16 ROR #4 或 64 ROR #6)，则汇编器希望使用涉及零移位的那个形式，如果可获得的话。所以，如果 0 <= const <= 255 则 MOVS Rn,#const 将保持进位标志不受影响，否则将改变它。

aaaa	汇编器	意思	P-Code
0000	AND	逻辑与	Rd = Rn AND Op2
0001	EOR	逻辑异或	Rd = Rn EOR Op2
0010	SUB	减法	Rd = Rn - Op2
0011	RSB	反向减法	Rd = Op2 - Rn
0100	ADD	加法	Rd = Rn + Op2
0101	ADC	带进位的加法	Rd = Rn + Op2 + C
0110	SBC	带借位的减法	Rd = Rn - Op2 - (1-C)
0111	RSC	带借位的反向减法	Rd = Op2 - Rn - (1-C)
1000	TST	测试位	Rn AND Op2
1001	TEQ	测试等同	Rn EOR Op2
1010	CMP	比较	Rn - Op2
1011	CMN	比较取负	Rn + Op2
1100	ORR	逻辑或	Rd = Rn OR Op2
1101	MOV	传送值	Rd = Op2
1110	BIC	位清除	Rd = Rn AND NOT Op2
1111	MVN	传送取非	Rd = NOT Op2

注意 MVN 和 CMN 不是象表面上的那种关系；MVN 使用直接的逐位(bitwise)非操作，把

Rn 设置为 Op2 对 1 的补码(反码)。CMN 把 Rn 与 Op2 对 2 的补码进行比较。

这些指令可归入 4 个子集:

MOV, MVN

Rn 被忽略，并且应当是 0000 。如果设置了 S 位，则在结果上设置 N 和 Z 标志。并且如果使用了移位器，则 C 标志被设置为被移出的最后一位。不影响 V 标志。

CMN, CMP, TEQ, TST

Rd 不被指令所设置，并且应当是 0000 。必须设置 S 位(多数汇编器会自动完成;如果没有设置它，则这个指令将是 MRS 、MSR、或一个未分配的指令。)

算术操作(CMN, CMP)在结果上设置 N 和 Z 标志，从 ALU 得到 C 和 V 标志。

逻辑操作(TEQ, TST)在结果上设置 N 和 Z 标志，如果使用了移位器则从它得到 C 标志(在这种情况下它变成被移出的最后一位)，不影响 V 标志。

有一个特殊情况(对于 ARMs >= 6 只针对 26 位模式),dddd 字段是 1111 导致用结果的相应的位设置标志(在用户模式下)，或整个 26 位 PSR (在特权模式下)。这由给指令的 P 后缀来指示 - CMNP、CMPP、TEQP、TSTP。常用 TEQP PC,#(新模式编号) 来改变模式。在 32 位模式，应当使用 MSR 来替代(因为 TEQP 等不再工作)。

ADC, ADD, RSB, RSC, SBC, SUB

如果设置了 S 位，则在结果上设置 N 和 Z 标志，从 ALU 的得到 C 和 V 标志。

AND, BIC, EOR, ORR

如果设置了 S 位，则在结果上设置 N 和 Z 标志,如果使用了移位器则从它得到 C 标志(在这种情况下它变成被移出的最后一位)，不影响 V 标志。

可以使用 ADD 和 SUB 以与位置无关的方式使寄存器指向数据，例如 ADD R0,PC,#24。这很有用，一些汇编器有一个叫做 ADR 的特殊宏指令(directive)，它自动生成恰当的 ADD 或 SUB 指令。(ADR R0, fred典型的把 fred 的地址放置到 R0 中，假定 fred 在范围内)。

在 26-bit 模式下，在 R15 是使用的寄存器之一的时候发生一种特殊情况:

- 如果 Rn = R15 则使用的 R15 值屏蔽掉了所有 PSR 位。
- 如果 Op2 涉及 R15 ，则使用所有的 32 位。

在 32-bit 模式下，使用 R15 的所有的位。

在 26-bit 模式下，如果 Rd = R15 则:

- 如果未设置 S 位，则只设置 PC 的 24 位。
- 如果设置了 S 位，则覆写 PC 和 PSR 二者(除非在非用户模式下，否则不改变模式位、I 和 F 位。)

对于 32-bit 模式， 如果 Rd=15，则覆写 PC 的所有的位，不包括最低的那两个有效位，它们总是零。如果未设置 S 位，则只进行上面这些;如果设置了 S 位，把当前模式的 SPSR 复制到 CPSR 中。在 32-bit 用户模式下，你不应该执行把 PC 作为目的寄存器并设置了 S 位的指令，因为用户模式没有 SPSR。(顺便说一句，你这样做不会打断处理器 - 这样做的结果只是未定义而已，且在不同的处理器上可能不同。)

执行这些指令使用下列数目的周期: 1S + (1S 如果使用了寄存器控制的移位) + (1S + 1N 如果改变了 PC)

分支指令

xxxx101L oooooooo oooooooo oooooooo

典型的汇编语法:

```
BEQ  地址
BLNE 子例程
```

使用这些指令强制跳转到一个新地址，用相对于执行这个指令时 PC 值的以字为单位的偏移量给出这个新地址。

因为流水线的缘故，PC 总是超出存储这个指令的地址 2 个指令(8 字节)，所以分支的偏移量 = (位 0-23 的有符号扩展):

```
目的地址 = 当前地址 + 8 + (4 * 偏移量)
```

在 26-bit 模式下，清除目的地址的顶端 6 位。

如果设置了 L 位，则在进行这个分支之前把 PC 的当前内容复制到 R14。所以 R14 持有在这个分支后面的指令的地址，被调用的例程可以用 MOV PC,R14 返回。

在 26-bit 模式下，使用 MOVS PC,R14 来从一个带连接的分支返回，在返回时可以自动恢复 PSR 标志。 在 32-bit 模式下 MOVS PC,R14 的行为是不同的，并只适合于从例外返回。

执行分支和带连接的分支二者都使用 2S+1N 个周期。

乘法

xxxx0000 00ASdddd nnnnssss 1001mmmm

典型汇编语法:

```
MULEQS Rd, Rm, Rs
MLA    Rd, Rm, Rs, Rn
```

这些指令做两个操作数的乘法，并且可以选择加上第三个操作数，把结果放置到另一个寄存器中。

如果设置了 S 位，则在结果是设置 N 和 Z 标志，未定义 C 标志，不影响 V 标志。

如果设置了 A 位，则操作的效果是 Rd = Rm*Rs + Rn，否则是 Rd = Rm*Rs。

目的寄存器不应该与操作数寄存器 Rm 相同。R15 不应该用于操作数或目的寄存器。

执行这些指令在最坏的情况下使用 1S + 16I 个周期，并依赖于实际参数的值可以更小。实际时间依赖于 Rs 的值，依照下表:

Rs 的范围	周期数
&0 -&1	1S + 1I
&2 -&7	1S + 2I
&8 -&1F	1S + 3I
&20 -&7F	1S + 4I
&80 -&1FF	1S + 5I
&200 -&7FF	1S + 6I
&800 -&1FFF	1S + 7I
&2000 -&7FFF	1S + 8I
&8000 -&1FFFF	1S + 9I
&20000 -&7FFFF	1S + 10I
&80000 -&1FFFFF	1S + 11I
&200000 -&7FFFFF	1S + 12I
&800000 -&1FFFFFF	1S + 13I
&2000000 -&7FFFFFF	1S + 14I
&8000000 -&1FFFFFFF	1S + 15I
&20000000 -&FFFFFFFF	1S + 16I

这些乘法时序不适用于 ARM7DM。 ARM7DM 时序由下表给出:

Rs 的范围	MUL	MLA/ SMULL	SMLAL	UMULL	UMLAL
&0 -&FF	1S+1I	1S+2I	1S+3I	1S+2I	1S+3I
&100 -&FFFF	1S+2I	1S+3I	1S+4I	1S+3I	1S+4I
&10000 -&FFFFFF	1S+3I	1S+4I	1S+5I	1S+4I	1S+5I
&1000000 -&FEFFFFFF	1S+4I	1S+5I	1S+6I	1S+5I	1S+6I
&FF000000 -&FFFEFFFF	1S+3I	1S+4I	1S+5I	1S+5I	1S+6I
&FFFF0000 -&FFFFFEFF	1S+2I	1S+3I	1S+4I	1S+5I	1S+6I
&FFFFFF00 -&FFFFFFFF	1S+1I	1S+2I	1S+3I	1S+5I	1S+6I

长乘法(ARM7DM)

xxxx0000 1UAShhhh llllssss 1001mmmm

典型的汇编语法:

```
UMULL  Rl,Rh,Rm,Rs
UMLAL  Rl,Rh,Rm,Rs
SMULL  Rl,Rh,Rm,Rs
SMLAL  Rl,Rh,Rm,Rs
```

这些指令做寄存器 Rm 和 Rs 的值的乘法并获得一个 64-bit 乘积。

在清除了 U 位的时候乘法是无符号的(UMULL 或 UMLAL)，否则是有符号的(SMULL, SMLAL)。在清除了 A 位的时候，把结果的低有效的那一半存储在 Rl 中并把它的高有效的那一半存储到 Rh 中。在设置了 A 位的时候，转而把结果加到 Rh,Rl 的内容上。

不应该使用程序计数器 R15。Rh、Rl 和 Rm 应该不同。

如果设置了 S 位，则在64-bit 位结果是设置 N 和 Z 标志,未定义 C 和 V 标志。

它们的时序可以在上面的乘法段落中找到。

单一数据传送

```
xxxx010P UBWLnnnn ddddoooo oooooooo  Immediate form
xxxx011P UBWLnnnn ddddcccc ctt0mmmm  Register form
```

典型的汇编语法:

```
LDR  Rd, [Rn, Rm, ASL#1]!
STR  Rd, [Rn],#2
LDRT Rd, [Rn]
LDRB Rd, [Rn]
```

这些指令装载/存储内存的一个字从/到一个寄存器。在指定地址时使用的第一个寄存器在术语上叫做基址寄存器。

如果设置了 L 位，则进行装载，否则进行存储。

如果设置了 P 位，则使用预先变址寻址，否则使用过后变址寻址。

如果设置了 U 位，则给出的偏移量被加到基址寄存器上 - 否则从中减去偏移量。

如果设置了 B 位，传送内存的一个字节，否则传送一个字。这在汇编器中表示为给根助记符的加上后缀‘B’。

W 位的解释依赖于使用的地址模式:

- 对于预先变址寻址，设置 W 位强制把用做地址转换的最终地址写回基址寄存器中。(例如，传送的副作用是 Rn := Rn +/- offset。这在汇编器中表示为给指令加上后缀‘!’。)
- 对于过后变址寻址，地址总是写回，设置 W 位指示在进行传送之前强制地址转换。这在汇编器中表示为给指令加上后缀‘T’。

地址转换导致芯片告知内存系统这是一个用户模式传送，而不管此时芯片是处于用户模式中还是处于特权模式中。这是有用的，例如在写模拟器的时候：假如一个用户模式程序一个内存区域执行了一个 STF 指令，而用户模式代码不可以写这个内存区域。如果由一个 FPA 来指令它，它将异常终止。如果由一个 FPE 来执行它，它也应该异常终止。但是 FPE 运行在一个特权模式下，所以如果它使用普通存储指令，则它不会异常终止。为了使异常终止正确工作，在一个特权模式调用它时使用 STRT 替代普通存储指令，使其如同调用自用户模式。

如果使用这个指令的立即数形式，o 字段给出一个 12-bit 偏移量。如果使用了寄存器形式，则按对数据处理指令那样解码它，限制是不允许使用寄存器指令移位量。

如果 R15 被用做 Rd，不修改 PSR。PC 不应该被用在 Op2 中。

其他限制:

- 在基址寄存器是 PC 的时候不要使用写回或过后变址。
- 不要使用 PC 作为给 LDRB 或 STRB 的 Rd。
- 在使用带有寄存器偏移量的过后变址时，不要让 Rn 和 Rm 是同一个寄存器(这样做导致不可能从异常终止中恢复)。

装载使用 1S + 1N + 1I + (1S + 1N 如果改变了 PC)个周期，而存储使用 2N 个周期。

块数据传送

xxxx100P USWLnnnn 11111111 11111111

典型的汇编语法:

```
LDMFD   Rn!, {R0-R4, R8, R12}
STMEQIA Rn,    {R0-R3}
STMIB   Rn,    {R0-R3}^
```

使用这些指令来同时装载/存储多个寄存器从/到内存。使用的内存地址从在基址寄存器 Rn 中持有的值指定的内存地址要么增加要么减少地址，(可以存储基址寄存器自身)，并且最终的地址可以被写回到基址寄存器中。这些指令适合于实现栈，在进入/退出一个子例程时存储/恢复寄存器的内容。

U 位指示对每个寄存器地址将被 +4 (设置)所修改，还是被 -4 (清除)所修改。

W 位总是指示写回。

如果设置了 L 位，则指示进行一个装载操作，如果清除了，则指示存储。

使用 P 位指示在每次装载/存储之前还是之后增加/减少基址寄存器(参见下面表格)。

如果这个操作要装载/存储 R1 则设置位 1。

汇编器典型的用条件代码跟随助记符根，并随后用两个字母代码指示 U 和 W 位的设置。

根	意思	P U
DA	在每次存储/装载之后减少 Rn	0 0
DB	在每次存储/装载之前减少 Rn	1 0
IA	在每次存储/装载之后增加 Rn	0 1
IB	在每次存储/装载之前增加 Rn	1 1

在实现栈的时候有更清楚的同义词:

根	意思

EA　　空升序栈
ED　　空降序栈
FA　　满升序栈
FD　　满降序栈

在一个空栈中，栈指针指向下一个空位置。在一个满栈中栈指针指向最顶端满位置。升序栈向高位置增长，而降序栈向低位置增长。

存储的寄存器总是最低编号的寄存器在内存中在最低地址。这可以影响入栈和出栈代码。例如，如果我想把 R1-R4 压入栈中，接着每次把它们中的两个装载回来，要使它们回到原先的寄存器，对于降序栈我需要做类似下面的事:

```
STMFD R13!,{R1,R2,R3,R4}  ;放置 R1 在内存低端，就是说在栈顶
LDMFD R13!,{R1,R2}
LDMFD R13!,{R3,R4}
```

对于升序栈则是:

```
STMFA R13!,{R1,R2,R3,R4}  ; 放置 R4 在内存高端，就是说在栈顶
LDMFA R13!,{R3,R4}
LDMFA R13!,{R1,R2}
```

同义的代码如下:

代码	装载	存储
EA	DB	IA
ED	IB	DA
FA	DA	IB
FD	IA	DB

S 为控制两个特殊的功能，它们被汇编器指示为在指令的结束处放置“^”:

- 如果设置了 S 位，并且指令是 LDM 而 R15 在寄存器列表中，则:
  - 在 26-bit 特权模式下，装载 R15 的所有 32 位。
  - 在 26-bit 用户模式下，装载 R15 的 4 个标志位和 24 个 PC 位。忽略装载值的位 27 、26、1 和 0 。
  - 在 32-bit 模式下，装载 R15 的所有 32 位，但要注意底端的两位总是零，所以忽略装载到它们的任何东西。除此之外，把当前模式的 SPSR 传送到 CPSR；因为用户模式没有 SPSR ，这种类型的指令不应该用在 32-bit 用户模式下。
- 如果设置了 S 位，并且要么指令是 STM 要么 R15 不在寄存器列表中，则传送用户模式的寄存器而不是当前模式的寄存器。在用户模式下不应该使用这个指令。

特殊情况发生在基址寄存器存在于要传送的寄存器列表中的时候。

- 基址寄存器总是可以被装载而没有任何问题。但是，如果基址寄存器被装载则不能指定写回 - 你不能同时把写回值和装载值二者写到基址寄存器中!
- 如果不使用写回则可以存储基址寄存器而没有任何问题。
- 如果在存储包含基址寄存器的一个寄存器列表的时候使用了写回，则只有在基址寄存器是列表的第一个的情况下，才在写回之前写基址寄存器的值到内存。其他情况，写到内存中的值未被定义。

进一步的特殊情况发生在程序计数器存在于要装载和保存的寄存器列表中。

- (在 26 位模式下) PSR 总是与 PC 一起保存((在所有模式下) PC 总是超出当前执行的指令的地址 12 字节，而不是通常的 8 字节)。
- 在装载时，只有在设置了 S 位的时候，才影响 PSR 的在当前模式下可改变的位。

PC 不应该作为基址寄存器。

块装载使用 nS + 1N + 1I + (1S + 1N如果改变了 PC)个周期，而块存储使用 (n-1)S + 2N 个周期，这里的“n”是被传送的字的数目。

软件中断

xxxx1111 yyyyyyyy yyyyyyyy yyyyyyyy

典型的汇编语法:

```
SWI   "OS_WriteI"
SWINE &400C0
```

在遇到软件中断的时候，ARM 切换到 SVC 模式中，把 R15 的当前值保存到 R14_SVC 中，并跳转到内存中的位置 8，它假定在这里可以找到一个 SWI 处理例程来解码刚才执行的 SWI 指令的低 24 位，并以特定于操作系统的方式做与 SWI 编号有关的事情。

在 ARM 上写的操作系统典型的使用 SWI 来为编程者提供各种例程。

执行 SWI 指令使用 2S + 1N个周期(加上解码 SWI 编号和执行适当例程使用的时间)。

协处理器数据操作

xxxx1110 oooonnnn ddddpppp qqq0mmmm

典型的汇编语法:

```
CDP p, o, CRd, CRn, CRm, q
```

CDP p, o, CRd, CRn, CRm

把这个指令传递给协处理器 p，告诉它在协处理器寄存器 CRn 和 CRm 上，进行操作 o，并把结果放置到 Crd 中。

可以使用 qqq 提供与操作有关的补充信息。

这些指令的准确意思依赖于使用的特定的协处理器；上面只是推荐的位的用法(实际上 FPA 就不遵守它)。唯一强制的部分是 pppp 必须是协处理器编号：协处理器设计者自由的按需要分配 oooo、nnnn、dddd、qqq 和 mmmm。

如果协处理器以与推荐的不同的方式使用这些位，可能需要使用汇编器宏来对人有意义的指令语法转换成正确的 CDP 指令。对最常使用的协处理器如 FPA，许多汇编器有额外的内置助记符并自动做这个转换。(例如，汇编 MUFEZ F0,F1,#10 成等价的 CDP 1,1,CR0,CR9,CR15,3。)

当前定义的协处理器编号包括:

1 和 2 浮点单元
15 Cache 控制器

如果做了到协处理器的一个调用而协处理器未做响应(通常因为它不存在!)，则调用未定义指令向量(完全同于后面描述的未定义指令)。这用于在没有 FPA 的机器上透明的提供 FP 支持。

执行这些指令使用 1S + bI 个周期，这里的 b 是协处理器在接受指令之前导致 ARM 忙等的周期数目：这在协处理器控制之下。

协处理器数据传送和寄存器传送

xxxx110P UNWLnnnn DDDDpppp oooooooo LDC/STC
xxxx1110 oooLNNNN ddddpppp qqq1MMMM MRC/MCR

它们还是依赖于使用的特定协处理器 p。

N 和 D 表示协处理器寄存器编号，n 和 d 是 ARM 处理器编号。o 是协处理器使用的操作。 M 表示协处理器自由的按需使用的位。

在第一种形式中，如果 L=1 则表示 LDC，否则是 STC。这些指令分别的表现得象 LDR 或 STR，在带有立即数偏移量的各种情况下，有下列例外。

- 偏移量是 4*(oooooooo)，不是通常的 12-bit 常数。
- 如果指定了 P=0 (过后变址)，则 W 必须是 1，而 W 是 1 只是指示要求写回，而不是告诉内存系统 这是一个用户模式传送。为将来的扩充保留 P=0 和 W=0 的指令。

- 在装载或存储一个或更多协处理器寄存器时，协处理器从 DDDD 和 N 位确定要装载或存储多少和哪个寄存器: ARM 所做的就是传送一个字到/从指示的地址，接着传送另一个到/从指示的地址＋4，接着传送一个到/从指示的地址＋8，以此类推，直到协处理器告诉它停止。
- 作为惯例，DDDD 表示要装载或存储的(第一个)协处理器寄存器，而 N 以某种方式表示长度，使用 N=1 指示一个“长”形式。协处理器设计者可以自由的忽略它...
- 下面是汇编语法:
- LDC p,CRd,[Rn,#20] ;短形式 (N=0)，预先变址
- STCL p,CRd,[Rn,#-32]! ;长形式 (N=1)，带写回的预先变址
- LDCNEL p,CRd,[Rn],#-100 ;长形式 (N=1)，过后变址

在第二种形式中，如果 L=1 则表示 MRC，否则是 MCR。MRC 传送一个协处理器寄存器到一个 ARM 寄存器， MCR 做反方向传送(字母看起来象是写反了，记住在 ARM 汇编器中目的通常写在左边)。

MCR 传送 ARM 寄存器 Rd 的内容到协处理器。协处理器基于 ooo、dddd、qqq 和 MMMM 字段的值自由的做它想做的任何事情，尽管有一个“标准的”解释：把它写到协处理器寄存器 CRN，使用操作ooo，用CRM 和 qqq 提供可能的补充控制。汇编语法是:

```
MCR   p,o,Rd,CRN,CRM,q
```

给 MCR 指令的 Rd 不应该是 R15。

MRC 从协处理器传送一个单一的字并把它放置到 ARM 寄存器 Rd 中。协处理器使用与 MCR 相同的字段自由的以任何方式生成这个字，有一个标准的解释：它来自 CRN，使用操作 ooo，用 CRM 和 qqq 提供可能的补充控制。汇编语法是:

```
MRC   p,o,Rd,CRN,CRM,q
```

如果给 MRC 指令的 Rd 是 R15，使用传送的字的顶端 4 位来设置标志；丢弃余下的 28 位。(例如，这种机制用于浮点比较指令。)

执行 LDC 和 STC 使用 (n-1)S + 2N + bI 个周期， MRC 使用 1S+bI+1C 个周期，而 MCR 使用 1S + (b+1)I + 1C个周期，这里的 b 是协处理器在接受指令之前导致 ARM 忙等的周期数目: 这在协处理器控制之下，而 n 是传送的字的数目(注意这在协处理器而不是 ARM 的控制下)

单一数据交换(ARM 3 和以后，包括 ARM 2aS)

xxxx0001 0B00nnnn dddd0000 1001mmmm

典型的汇编语法:

```
SWP Rd, Rm, [Rn]
```

这些指令装载一个内存的一个字(用寄存器 Rn 给出地址)到一个寄存器 Rd 并存储寄存器 Rm 到相同的地址。Rm 和 Rd 可以是同一个寄存器，在这种情况下交换寄存器和内存位置的内容。装载和存储操作通过设置 LOCK 引角(pin)为高电平来在操作期间锁定在一起，这指示内存管理器它们应当被没有中断的完成。

如果设置了 B 位，则传送一字节的内存，否则传送一个字。

Rd、Rn、和 Rm 都不能是 R15。

执行这个指令使用 1S + 2N + 1I 个周期。

状态寄存器传送(ARM 6 和以后)

```
xxxx0001 0s10aaaa 11110000 0000mmmm  MSR  寄存器形式
xxxx0011 0s10aaaa 1111rrrr bbbbbbbb  MSR  立即数形式
xxxx0001 0s001111 dddd0000 00000000  MRS
```

典型的汇编语法:

```
MSR   SPSR_all, Rm          ;aaaa = 1001
MSR   CPSR_flg, #&F0000000  ;aaaa = 1000
MSRNE CPSR_ctl, Rm          ;aaaa = 0001
MRS   Rd, CPSR
```

设置 s 位时意味着访问当前特权模式的 SPSR，而不是 CPSR。只能在特权模式下执行这个命令的时候设置此位。

使用 MSR 来传送一个寄存器或常数到一个状态寄存器。

aaaa 位接受下列值:

值　　意思

0001　设置有关的 PSR 的控制位。
1000　设置有关的 PSR 的标志位。
1001　设置有关的 PSR 的控制位和标志位(就是说所有现存的位)。
其他的值为将来的扩充而保留。

在寄存器形式中，源寄存器是 Rm。在立即数形式中，来源是 #b, ROR #2r。

R15 不应该被指定为 MRS 指令的源寄存器。

使用 MRS 来传送处理器的状态到一个寄存器。

dddd 位存储目的寄存器的编号；Rd 一定不能是 R15。

N.B. 指令编码对应于对应与操作码(opcode)是 10xx 并清除了 S 位的数据处理指令(就是测试指令)。

执行这些指令总是使用 1S 个周期。

未定义指令

xxxx0001 yyyyyyyy yyyyyyyy 1yy1yyyy 专属 ARM 2
xxxx011y yyyyyyyy yyyyyyyy yyy1yyyy

这些指令目前未定义。在遇到未定义指令时，ARM 切换到 SVC 模式(在 ARM 3 和以后)或 Undef 模式(在 ARM 6 和以后)，把 R15 的旧有值放置到 R14_SVC (或 R14_UND)中并跳转到一个位置，在那里它希望找到解码这个未定义指令的代码并相应的执行它。

注意:

- 这些指令被文档为“未定义的”，原因是这种方式下它们进入未定义指令处理器陷阱。许多其他指令是以更宽松的方式未被定义的，而不说出它们做什么。 例如，下面形式的位模式(pattern):
- xxxx0000 01xxxxxx xxxxxxxx 1001xxxx

  与数据处理的指令、乘法、长乘法和 SWP 指令有关，但却不是其中一个的原因是:

  - 数据处理指令的位 25 = 0 和位 4 = 1 时有寄存器控制的移位，所以必须位 7 = 0。
  - 乘法指令的位 23:22 = 00 。
  - 长乘法指令的位 23:22 = 1U 。
  - SWP 指令的位 24 = 1 。

  这些指令只是简单的未定义做什么，但是上面列出的那些指令实际上**定义**为进入未定义指令陷阱，至少直到将来为它们找到某种用途。

- 注意“专属 ARM2 ”的未定义指令包括了在 ARM3/ARM2as 和以后成为 SWP 的那些指令。