ARM 汇编手册

版权声明

本手册为北京顶嵌开源科技有限公司内部培训资料，仅供本公司内部学习使用，在未经本公司授权的情况下，请勿用作任何商业用途。

目 录

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：

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...）。

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。

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。

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

- ADC
- ADD
- AND
- BIC
- EOR
- MOV
- MVN
- ORR
- RSB
- RSC
- SBC
- SUB

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

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。

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。

乘法指令

- MLA
- MUL

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。

程序状态寄存器访问指令

ARM 微处理器支持程序状态寄存器访问指令，用于在程序状态寄存器和通用寄存器之间传送
数据，程序状态寄存器访问指令包括以下两条：
— MRS 程序状态寄存器到通用寄存器的数据传送指令
— MSR 通用寄存器到程序状态寄存器的数据传送指令

MRS 指令

MRS 指令的格式为：
MRS{条件} 通用寄存器，程序状态寄存器（CPSR 或SPSR）
MRS 指令用于将程序状态寄存器的内容传送到通用寄存器中。该指令一般用在以下几种情况：
— 当需要改变程序状态寄存器的内容时，可用MRS 将程序状态寄存器的内容读入通用寄存
器，修改后再写回程序状态寄存器。
— 当在异常处理或进程切换时，需要保存程序状态寄存器的值，可先用该指令读出程序状态
寄存器的值，然后保存。
指令示例：

```
MRS R0，CPSR ；传送CPSR 的内容到R0
MRS R0，SPSR ；传送SPSR 的内容到R0
```

MSR 指令

MSR 指令的格式为：
MSR{条件} 程序状态寄存器（CPSR 或SPSR）_<域>，操作数
MSR 指令用于将操作数的内容传送到程序状态寄存器的特定域中。其中，操作数可以为通用寄存器或立即数。<域>用于设置程序状态寄存器中需要操作的位，32 位的程序状态寄存器可分为4 个域：
位[31：24]为条件标志位域，用f 表示；
位[23：16]为状态位域，用s 表示；
位[15：8]为扩展位域，用x 表示；
位[7：0]为控制位域，用c 表示；
该指令通常用于恢复或改变程序状态寄存器的内容，在使用时，一般要在MSR 指令中指明将要操作的域。
指令示例：

```
MSR CPSR，R0 ；传送R0 的内容到CPSR
MSR SPSR，R0 ；传送R0 的内容到SPSR
```

MSR CPSR_c，R0 ；传送 R0 的内容到 SPSR，但仅仅修改 CPSR 中的控制位域

异常产生指令

ARM 微处理器所支持的异常指令有如下两条：
— SWI 软件中断指令
— BKPT 断点中断指令

SWI 指令

SWI 指令的格式为：
SWI{条件} 24 位的立即数
SWI 指令用于产生软件中断，以便用户程序能调用操作系统的系统例程。操作系统在SWI 的异常处理程序中提供相应的系统服务，指令中24 位的立即数指定用户程序调用系统例程的类型，相关参数通过通用寄存器传递，当指令中24 位的立即数被忽略时，用户程序调用系统例程的类型由通用寄存器R0 的内容决定，同时，参数通过其他通用寄存器传递。
指令示例：

```
SWI 0x02 ；该指令调用操作系统编号位02 的系统例程。
```

BKPT 指令

BKPT 指令的格式为：
BKPT 16 位的立即数
BKPT 指令产生软件断点中断，可用于程序的调试。

协处理器指令

ARM 微处理器可支持多达16 个协处理器，用于各种协处理操作，在程序执行的过程中，每个协处理器只执行针对自身的协处理指令，忽略ARM 处理器和其他协处理器的指令。ARM 的协处理器指令主要用于ARM 处理器初始化ARM 协处理器的数据处理操作，以及在ARM 处理器的寄存器和协处理器的寄存器之间传送数据，和在ARM 协处理器的寄存器和存储器之间传送数据。ARM 协处理器指令包括以下5 条：

— CDP 协处理器数操作指令
— LDC 协处理器数据加载指令
— STC 协处理器数据存储指令
— MCR ARM 处理器寄存器到协处理器寄存器的数据传送指令
— MRC 协处理器寄存器到ARM 处理器寄存器的数据传送指令

CDP 指令

CDP 指令的格式为：
CDP{条件} 协处理器编码，协处理器操作码1，目的寄存器，源寄存器1，源寄存器2，协处理器操作码2。
CDP 指令用于ARM 处理器通知ARM 协处理器执行特定的操作,若协处理器不能成功完成特定的操作，则产生未定义指令异常。其中协处理器操作码1 和协处理器操作码2 为协处理器将要执行的操作，目的寄存器和源寄存器均为协处理器的寄存器，指令不涉及ARM 处理器的寄存器和存储器。
指令示例：

```
CDP P3，2，C12，C10，C3，4 ；该指令完成协处理器P3 的初始化
```

LDC 指令

LDC 指令的格式为：
LDC{条件}{L} 协处理器编码,目的寄存器，[源寄存器]
LDC 指令用于将源寄存器所指向的存储器中的字数据传送到目的寄存器中，若协处理器不能成功完成传送操作，则产生未定义指令异常。其中，{L}选项表示指令为长读取操作，如用于双精度数据的传输。
指令示例：

```
LDC P3，C4，[R0] ；将ARM 处理器的寄存器R0 所指向的存储器中的字数据传送到协处理
器P3 的寄存器C4 中。
```

STC 指令

STC 指令的格式为：
STC{条件}{L} 协处理器编码,源寄存器，[目的寄存器]

STC 指令用于将源寄存器中的字数据传送到目的寄存器所指向的存储器中，若协处理器不能成功完成传送操作，则产生未定义指令异常。其中，{L}选项表示指令为长读取操作，如用于双精度数据的传输。

指令示例：

```
STC P3，C4，[R0] ；将协处理器P3 的寄存器C4 中的字数据传送到ARM 处理器的寄存器
R0 所指向的存储器中。
```

MCR 指令

MCR 指令的格式为：

MCR{条件} 协处理器编码，协处理器操作码1，源寄存器，目的寄存器1，目的寄存器2，协处理器操作码2。

MCR 指令用于将ARM 处理器寄存器中的数据传送到协处理器寄存器中,若协处理器不能成功完成操作，则产生未定义指令异常。其中协处理器操作码1 和协处理器操作码2 为协处理器将要执行的操作，源寄存器为ARM 处理器的寄存器，目的寄存器1 和目的寄存器2 均为协处理器的寄存器。

指令示例：

```
MCR P3，3，R0，C4，C5，6 ；该指令将ARM 处理器寄存器R0 中的数据传送到协处理器P3 的寄
存器C4 和C5 中。
```

MRC 指令

MRC 指令的格式为：

MRC{条件} 协处理器编码，协处理器操作码1，目的寄存器，源寄存器1，源寄存器2，协处理器操作码2。

MRC 指令用于将协处理器寄存器中的数据传送到ARM 处理器寄存器中,若协处理器不能成功完成操作，则产生未定义指令异常。其中协处理器操作码1 和协处理器操作码2 为协处理器将要执行的操作，目的寄存器为ARM 处理器的寄存器，源寄存器1 和源寄存器2 均为协处理器的寄存器。

指令示例：

```
MRC P3，3，R0，C4，C5，6 ；该指令将协处理器 P3 的寄存器中的数据传送到 ARM 处理器寄存器
```

跳转指令

跳转指令用于实现程序流程的跳转，在ARM 程序中有两种方法可以实现程序流程的跳转：
— 使用专门的跳转指令。
— 直接向程序计数器PC 写入跳转地址值。
通过向程序计数器PC 写入跳转地址值，可以实现在4GB 的地址空间中的任意跳转，在跳转之前结合使用

```
MOV LR，PC
```

等类似指令，可以保存将来的返回地址值，从而实现在4GB 连续的线性地址空间的子程序调用。
ARM 指令集中的跳转指令可以完成从当前指令向前或向后的32MB 的地址空间的跳转，包括
以下4 条指令：
— B 跳转指令
— BL 带返回的跳转指令
— BLX 带返回和状态切换的跳转指令
— BX 带状态切换的跳转指令

B 指令

B 指令的格式为：
B{条件} 目标地址
B 指令是最简单的跳转指令。一旦遇到一个 B 指令，ARM 处理器将立即跳转到给定的目标地址，从那里继续执行。注意存储在跳转指令中的实际值是相对当前PC 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（参考寻址方式中的相对寻址）。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(前后32MB 的地址空间)。以下指令：

```
B Label ；程序无条件跳转到标号Label 处执行
CMP R1，＃0 ；当CPSR 寄存器中的Z 条件码置位时，程序跳转到标号Label 处执行
BEQ Label
```

BL 指令

BL 指令的格式为：
BL{条件} 目标地址
BL 是另一个跳转指令，但跳转之前，会在寄存器R14 中保存PC 的当前内容，因此，可以通过将R14 的内容重新加载到PC 中，来返回到跳转指令之后的那个指令处执行。该指令是实现子程序调用的一个基本但常用的手段。以下指令：

```
BL Label ；当程序无条件跳转到标号Label 处执行时，同时将当前的PC 值保存到R14 中
```

BLX 指令

BLX 指令的格式为：
BLX 目标地址
BLX 指令从ARM 指令集跳转到指令中所指定的目标地址，并将处理器的工作状态有ARM 状态切换到Thumb 状态，该指令同时将PC 的当前内容保存到寄存器R14 中。因此，当子程序使用Thumb 指令集，而调用者使用ARM 指令集时，可以通过BLX 指令实现子程序的调用和处理器工作状态的切换。
同时，子程序的返回可以通过将寄存器R14 值复制到PC 中来完成。

BX 指令

BX 指令的格式为：
BX{条件} 目标地址
BX 指令跳转到指令中所指定的目标地址，目标地址处的指令既可以是ARM 指令，也可以是Thumb
指令。

符号定义（Symbol Definition）伪指令

符号定义伪指令用于定义ARM 汇编程序中的变量、对变量赋值以及定义寄存器的别名等操作。
常见的符号定义伪指令有如下几种：
— 用于定义全局变量的GBLA、GBLL 和GBLS。
— 用于定义局部变量的LCLA、LCLL 和LCLS。
— 用于对变量赋值的SETA、SETL、SETS。
— 为通用寄存器列表定义名称的RLIST。

1、 GBLA、GBLL 和 GBLS

语法格式：
GBLA（GBLL 或GBLS） 全局变量名
GBLA、GBLL 和GBLS 伪指令用于定义一个ARM 程序中的全局变量，并将其初始化。其中：
GBLA 伪指令用于定义一个全局的数字变量，并初始化为0；
GBLL 伪指令用于定义一个全局的逻辑变量，并初始化为F（假）；
GBLS 伪指令用于定义一个全局的字符串变量，并初始化为空；
由于以上三条伪指令用于定义全局变量，因此在整个程序范围内变量名必须唯一。
使用示例：

```
GBLA Test1 ；定义一个全局的数字变量，变量名为Test1
Test1 SETA 0xaa ；将该变量赋值为0xaa
GBLL Test2 ；定义一个全局的逻辑变量，变量名为Test2
Test2 SETL {TRUE} ；将该变量赋值为真
GBLS Test3 ；定义一个全局的字符串变量，变量名为Test3
Test3 SETS “Testing” ；将该变量赋值为“Testing”
```

2、 LCLA、LCLL 和 LCLS

语法格式：
LCLA（LCLL 或LCLS） 局部变量名
LCLA、LCLL 和LCLS 伪指令用于定义一个ARM 程序中的局部变量，并将其初始化。其中：
LCLA 伪指令用于定义一个局部的数字变量，并初始化为0；
LCLL 伪指令用于定义一个局部的逻辑变量，并初始化为F（假）；
LCLS 伪指令用于定义一个局部的字符串变量，并初始化为空；
以上三条伪指令用于声明局部变量，在其作用范围内变量名必须唯一。
使用示例：

```
LCLA Test4 ；声明一个局部的数字变量，变量名为Test4
```

```
Test3 SETA 0xaa ；将该变量赋值为0xaa
LCLL Test5 ；声明一个局部的逻辑变量，变量名为Test5
Test4 SETL {TRUE} ；将该变量赋值为真
LCLS Test6 ；定义一个局部的字符串变量，变量名为Test6
Test6 SETS “Testing” ；将该变量赋值为“Testing”
```

3、 SETA、SETL 和 SETS

语法格式：
变量名 SETA（SETL 或SETS） 表达式
伪指令SETA、SETL、SETS 用于给一个已经定义的全局变量或局部变量赋值。
SETA 伪指令用于给一个数学变量赋值；
SETL 伪指令用于给一个逻辑变量赋值；
SETS 伪指令用于给一个字符串变量赋值；
其中，变量名为已经定义过的全局变量或局部变量，表达式为将要赋给变量的值。
使用示例：

```
LCLA Test3 ；声明一个局部的数字变量，变量名为Test3
Test3 SETA 0xaa ；将该变量赋值为0xaa
LCLL Test4 ；声明一个局部的逻辑变量，变量名为Test4
Test4 SETL {TRUE} ；将该变量赋值为真
```

4、 RLIST

语法格式：
名称 RLIST {寄存器列表}
RLIST 伪指令可用于对一个通用寄存器列表定义名称，使用该伪指令定义的名称可在ARM 指令LDM/STM 中使用。在LDM/STM 指令中，列表中的寄存器访问次序为根据寄存器的编号由低到高，而与列表中的寄存器排列次序无关。
使用示例：

```
RegList RLIST {R0-R5，R8，R10} ；将寄存器列表名称定义为RegList，可在ARM 指令LDM/STM
```

中通过该名称访问寄存器列表。

数据定义（Data Definition）伪指令

数据定义伪指令一般用于为特定的数据分配存储单元，同时可完成已分配存储单元的初始化。
常见的数据定义伪指令有如下几种：
— DCB 用于分配一片连续的字节存储单元并用指定的数据初始化。
— DCW（DCWU） 用于分配一片连续的半字存储单元并用指定的数据初始化。
— DCD（DCDU） 用于分配一片连续的字存储单元并用指定的数据初始化。
— DCFD（DCFDU）用于为双精度的浮点数分配一片连续的字存储单元并用指定的数据初始化。
— DCFS（DCFSU） 用于为单精度的浮点数分配一片连续的字存储单元并用指定的数据初始化。
— DCQ（DCQU） 用于分配一片以8 字节为单位的连续的存储单元并用指定的数据初始化。
— SPACE 用于分配一片连续的存储单元
— MAP 用于定义一个结构化的内存表首地址
— FIELD 用于定义一个结构化的内存表的数据域

1、 DCB

语法格式：
标号 DCB 表达式
DCB 伪指令用于分配一片连续的字节存储单元并用伪指令中指定的表达式初始化。其中，表达式可以为0～255 的数字或字符串。DCB 也可用“=”代替。
使用示例：

```
Str DCB “This is a test！” ；分配一片连续的字节存储单元并初始化。
```

2、 DCW（或 DCWU）

语法格式：
标号 DCW（或DCWU） 表达式
DCW（或DCWU）伪指令用于分配一片连续的半字存储单元并用伪指令中指定的表达式初始化。
其中，表达式可以为程序标号或数字表达式。。
用DCW 分配的字存储单元是半字对齐的，而用DCWU 分配的字存储单元并不严格半字对齐。
使用示例：

```
DataTest DCW 1，2，3 ；分配一片连续的半字存储单元并初始化。
```

3、 DCD（或 DCDU）

语法格式：
标号 DCD（或DCDU） 表达式
DCD（或DCDU）伪指令用于分配一片连续的字存储单元并用伪指令中指定的表达式初始化。
其中，表达式可以为程序标号或数字表达式。DCD 也可用“&”代替。
用DCD 分配的字存储单元是字对齐的，而用DCDU 分配的字存储单元并不严格字对齐。
使用示例：

```
DataTest DCD 4，5，6 ；分配一片连续的字存储单元并初始化。
```

4、 DCFD（或 DCFDU）

语法格式：
标号 DCFD（或DCFDU） 表达式
DCFD（或DCFDU）伪指令用于为双精度的浮点数分配一片连续的字存储单元并用伪指令中指定的表达式初始化。每个双精度的浮点数占据两个字单元。
用DCFD 分配的字存储单元是字对齐的，而用DCFDU 分配的字存储单元并不严格字对齐。
使用示例：

```
FDataTest DCFD 2E115，-5E7 ；分配一片连续的字存储单元并初始化为指定的双精度数。
```

5、 DCFS（或 DCFSU）

语法格式：
标号 DCFS（或DCFSU） 表达式
DCFS（或DCFSU）伪指令用于为单精度的浮点数分配一片连续的字存储单元并用伪指令中指定的表达式初始化。每个单精度的浮点数占据一个字单元。
用DCFS 分配的字存储单元是字对齐的，而用DCFSU 分配的字存储单元并不严格字对齐。
使用示例：

```
FDataTest DCFS 2E5，-5E－7 ；分配一片连续的字存储单元并初始化为指定的单精
度数。
```

6、 DCQ（或 DCQU）

语法格式：
标号 DCQ（或DCQU） 表达式
DCQ（或DCQU）伪指令用于分配一片以8 个字节为单位的连续存储区域并用伪指令中指定的表达式初始化。
用DCQ 分配的存储单元是字对齐的，而用DCQU 分配的存储单元并不严格字对齐。
使用示例：

```
DataTest DCQ 100 ；分配一片连续的存储单元并初始化为指定的值。
```

7、 SPACE

语法格式：
标号 SPACE 表达式
SPACE 伪指令用于分配一片连续的存储区域并初始化为0。其中，表达式为要分配的字节数。
SPACE 也可用“%”代替。
使用示例：

```
DataSpace SPACE 100 ；分配连续100 字节的存储单元并初始化为0。
```

8、 MAP

语法格式：
MAP 表达式{，基址寄存器}
MAP 伪指令用于定义一个结构化的内存表的首地址。MAP 也可用“^”代替。
表达式可以为程序中的标号或数学表达式，基址寄存器为可选项，当基址寄存器选项不存在时，表达式的值即为内存表的首地址，当该选项存在时，内存表的首地址为表达式的值与基址寄存器的和。
MAP 伪指令通常与FIELD 伪指令配合使用来定义结构化的内存表。
使用示例：

```
MAP 0x100，R0 ；定义结构化内存表首地址的值为0x100＋R0。
```

9、 FILED

语法格式：
标号 FIELD 表达式
FIELD 伪指令用于定义一个结构化内存表中的数据域。FILED 也可用“#”代替。
表达式的值为当前数据域在内存表中所占的字节数。
FIELD 伪指令常与MAP 伪指令配合使用来定义结构化的内存表。MAP 伪指令定义内存表的首地址，FIELD 伪指令定义内存表中的各个数据域，并可以为每个数据域指定一个标号供其他的指令引用。
注意MAP 和FIELD 伪指令仅用于定义数据结构，并不实际分配存储单元。
使用示例：

```
MAP 0x100 ；定义结构化内存表首地址的值为0x100。
A FIELD 16 ；定义A 的长度为16 字节，位置为0x100
B FIELD 32 ；定义B 的长度为32 字节，位置为0x110
S FIELD 256 ；定义S 的长度为256 字节，位置为0x130
```

汇编控制（Assembly Control）伪指令

汇编控制伪指令用于控制汇编程序的执行流程，常用的汇编控制伪指令包括以下几条：

— IF、ELSE、ENDIF
— WHILE、WEND
— MACRO、MEND
— MEXIT

1、 IF、ELSE、ENDIF

语法格式：
IF 逻辑表达式
指令序列1
ELSE
指令序列2
ENDIF
IF、ELSE、ENDIF 伪指令能根据条件的成立与否决定是否执行某个指令序列。当IF 后面的逻辑表达式为真，则执行指令序列1，否则执行指令序列2。其中，ELSE 及指令序列2 可以没有，此时，当IF 后面的逻辑表达式为真，则执行指令序列1，否则继续执行后面的指令。
IF、ELSE、ENDIF 伪指令可以嵌套使用。
使用示例：

```
GBLL Test ；声明一个全局的逻辑变量，变量名为Test
……
IF Test = TRUE
指令序列1
ELSE
指令序列2
ENDIF
```

2、 WHILE、WEND

语法格式：
WHILE 逻辑表达式
指令序列
WEND
WHILE、WEND 伪指令能根据条件的成立与否决定是否循环执行某个指令序列。当WHILE 后面的逻辑表达式为真，则执行指令序列，该指令序列执行完毕后，再判断逻辑表达式的值，若为真则继续执行，一直到逻辑表达式的值为假。
WHILE、WEND 伪指令可以嵌套使用。

使用示例：

```
GBLA Counter ；声明一个全局的数学变量，变量名为Counter
Counter SETA 3 ；由变量Counter 控制循环次数
……
WHILE Counter < 10
指令序列
WEND
```

3、 MACRO、MEND

语法格式：

$标号 宏名 $参数1，$参数2，……

指令序列

MEND

MACRO、MEND 伪指令可以将一段代码定义为一个整体，称为宏指令，然后就可以在程序中通过宏指令多次调用该段代码。其中，$标号在宏指令被展开时，标号会被替换为用户定义的符号，宏指令可以使用一个或多个参数，当宏指令被展开时，这些参数被相应的值替换。

宏指令的使用方式和功能与子程序有些相似，子程序可以提供模块化的程序设计、节省存储空间并提高运行速度。但在使用子程序结构时需要保护现场，从而增加了系统的开销，因此，在代码较短且需要传递的参数较多时，可以使用宏指令代替子程序。

包含在MACRO 和MEND 之间的指令序列称为宏定义体，在宏定义体的第一行应声明宏的原型（包含宏名、所需的参数），然后就可以在汇编程序中通过宏名来调用该指令序列。在源程序被编译时，汇编器将宏调用展开，用宏定义中的指令序列代替程序中的宏调用，并将实际参数的值传递给宏定义中的形式参数。

MACRO、MEND 伪指令可以嵌套使用。

4、 MEXIT

语法格式：

MEXIT

MEXIT 用于从宏定义中跳转出去。

其他常用的伪指令

还有一些其他的伪指令，在汇编程序中经常会被使用，包括以下几条：

— AREA
— ALIGN
— CODE16、CODE32
— ENTRY
— END
— EQU
— EXPORT（或GLOBAL）
— IMPORT
— EXTERN
— GET（或INCLUDE）
— INCBIN
— RN
— ROUT

1、 AREA

语法格式：

AREA 段名 属性1，属性2，……

AREA 伪指令用于定义一个代码段或数据段。其中，段名若以数字开头，则该段名需用“|”括

起来，如|1_test|。

属性字段表示该代码段（或数据段）的相关属性，多个属性用逗号分隔。常用的属性如下：

— CODE 属性：用于定义代码段，默认为READONLY。
— DATA 属性：用于定义数据段，默认为READWRITE。
— READONLY 属性：指定本段为只读，代码段默认为READONLY。
— READWRITE 属性：指定本段为可读可写，数据段的默认属性为READWRITE。
— ALIGN 属性：使用方式为ALIGN 表达式。在默认时，ELF（可执行连接文件）的代码段和数据段是按字对齐的，表达式的取值范围为0～31，相应的对齐方式为2 表达式次方。
— COMMON 属性：该属性定义一个通用的段，不包含任何的用户代码和数据。各源文件中同名的COMMON 段共享同一段存储单元。

一个汇编语言程序至少要包含一个段，当程序太长时，也可以将程序分为多个代码段和数据段。

使用示例：

```
AREA Init，CODE，READONLY
指令序列
；该伪指令定义了一个代码段，段名为Init，属性为只读
```

2、 ALIGN

语法格式：
ALIGN {表达式{，偏移量}}
ALIGN 伪指令可通过添加填充字节的方式，使当前位置满足一定的对其方式|。其中，表达式的值用于指定对齐方式，可能的取值为2 的幂，如1、2、4、8、16 等。若未指定表达式，则将当前位置对齐到下一个字的位置。偏移量也为一个数字表达式，若使用该字段，则当前位置的对齐方式为：2 的表达式次幂＋偏移量。
使用示例：

```
AREA Init，CODE，READONLY，ALIEN＝3 ；指定后面的指令为8 字节对齐。
指令序列
END
```

3、 CODE16、CODE32

语法格式：
CODE16（或CODE32）
CODE16 伪指令通知编译器，其后的指令序列为16 位的Thumb 指令。
CODE32 伪指令通知编译器，其后的指令序列为32 位的ARM 指令。
若在汇编源程序中同时包含ARM 指令和Thumb 指令时，可用CODE16 伪指令通知编译器其后的指令序列为16 位的Thumb 指令，CODE32 伪指令通知编译器其后的指令序列为32 位的ARM 指令。因此，在使用ARM 指令和Thumb 指令混合编程的代码里，可用这两条伪指令进行切换，但注意他们只通知编译器其后指令的类型，并不能对处理器进行状态的切换。
使用示例：

```
AREA Init，CODE，READONLY
……
CODE32 ；通知编译器其后的指令为32 位的ARM 指令
LDR R0，＝NEXT＋1 ；将跳转地址放入寄存器R0
BX R0 ；程序跳转到新的位置执行，并将处理器切换到Thumb 工作状态
……
CODE16 ；通知编译器其后的指令为16 位的Thumb 指令
NEXT LDR R3，＝0x3FF
……
END ；程序结束
```

4、 ENTRY

语法格式：
ENTRY
ENTRY 伪指令用于指定汇编程序的入口点。在一个完整的汇编程序中至少要有一个ENTRY（也可以有多个，当有多个ENTRY 时，程序的真正入口点由链接器指定），但在一个源文件里最多只能有一个ENTRY（可以没有）。

使用示例：

```
AREA Init，CODE，READONLY
ENTRY ；指定应用程序的入口点
……
```

5、 END

语法格式：

END

END 伪指令用于通知编译器已经到了源程序的结尾。

使用示例：

```
AREA Init，CODE，READONLY
……
END ；指定应用程序的结尾
```

6、 EQU

语法格式：

名称 EQU 表达式{，类型}

EQU 伪指令用于为程序中的常量、标号等定义一个等效的字符名称，类似于C 语言中的＃define。

其中EQU 可用“*”代替。

名称为EQU 伪指令定义的字符名称，当表达式为32 位的常量时，可以指定表达式的数据类型，可以有以下三种类型：

CODE16、CODE32 和DATA

使用示例：

```
Test EQU 50 ；定义标号Test 的值为50
Addr EQU 0x55，CODE32 ；定义Addr 的值为0x55，且该处为32 位的ARM 指令。
```

7、 EXPORT（或 GLOBAL）

语法格式：

EXPORT 标号{[WEAK]}

EXPORT 伪指令用于在程序中声明一个全局的标号，该标号可在其他的文件中引用。

EXPORT

可用GLOBAL 代替。标号在程序中区分大小写，[WEAK]选项声明其他的同名标号优先于该标号被

引用。

使用示例：

```
AREA Init，CODE，READONLY
EXPORT Stest ；声明一个可全局引用的标号Stest
……
```

```
END
```

8、 IMPORT

语法格式：

IMPORT 标号{[WEAK]}

IMPORT 伪指令用于通知编译器要使用的标号在其他的源文件中定义，但要在当前源文件中引用，而且无论当前源文件是否引用该标号，该标号均会被加入到当前源文件的符号表中。

标号在程序中区分大小写，[WEAK]选项表示当所有的源文件都没有定义这样一个标号时，编译器也不给出错误信息，在多数情况下将该标号置为0，若该标号为B 或BL 指令引用，则将B 或BL指令置为NOP 操作。

使用示例：

```
AREA Init，CODE，READONLY
IMPORT Main ；通知编译器当前文件要引用标号Main，但Main 在其他源文件中
定义
……
END
```

9、 EXTERN

语法格式：

EXTERN 标号{[WEAK]}

EXTERN 伪指令用于通知编译器要使用的标号在其他的源文件中定义，但要在当前源文件中引用，如果当前源文件实际并未引用该标号，该标号就不会被加入到当前源文件的符号表中。

标号在程序中区分大小写，[WEAK]选项表示当所有的源文件都没有定义这样一个标号时，编译器也不给出错误信息，在多数情况下将该标号置为0，若该标号为B 或BL 指令引用，则将B 或BL指令置为NOP 操作。

使用示例：

```
AREA Init，CODE，READONLY
EXTERN Main ；通知编译器当前文件要引用标号Main，但Main 在其他源
文件中定义
……
END
```

10、 GET（或 INCLUDE）

语法格式：

GET 文件名

GET 伪指令用于将一个源文件包含到当前的源文件中，并将被包含的源文件在当前位置进行汇编处理。可以使用INCLUDE 代替GET。

汇编程序中常用的方法是在某源文件中定义一些宏指令，用EQU 定义常量的符号名称，用

MAP和FIELD 定义结构化的数据类型，然后用GET 伪指令将这个源文件包含到其他的源文件中。使用方法与C 语言中的“include”相似。

GET 伪指令只能用于包含源文件，包含目标文件需要使用INCBIN 伪指令

使用示例：

```
AREA Init，CODE，READONLY
GET a1.s ；通知编译器当前源文件包含源文件a1.s
GE T C：\a2.s ；通知编译器当前源文件包含源文件C：\ a2.s
……
END
```

11、 INCBIN

语法格式：

INCBIN 文件名

INCBIN 伪指令用于将一个目标文件或数据文件包含到当前的源文件中，被包含的文件不作任何变动的存放在当前文件中，编译器从其后开始继续处理。

使用示例：

```
AREA Init，CODE，READONLY
INCBIN a1.dat ；通知编译器当前源文件包含文件a1.dat
INCBIN C：\a2.txt ；通知编译器当前源文件包含文件C：\a2.txt
……
END
```

12、 RN

语法格式：

名称 RN 表达式

RN 伪指令用于给一个寄存器定义一个别名。采用这种方式可以方便程序员记忆该寄存器的功能。其中，名称为给寄存器定义的别名，表达式为寄存器的编码。

使用示例：

```
Temp RN R0 ；将R0 定义一个别名Temp
```

13、 ROUT

语法格式：

{名称} ROUT

ROUT 伪指令用于给一个局部变量定义作用范围。在程序中未使用该伪指令时，局部变量的作用范围为所在的AREA，而使用ROUT 后，局部变量的作为范围为当前ROUT 和下一个ROUT 之

间。

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 s1]		
[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，下面是代码是编译器生成的...

```
        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。

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。

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}^
```