MM 的读书笔记系列 I

ARM 体系结构与编程

——METAL MAX

原书作者：杜春雷

笔记：METAL MAX

2008 年 5 月 1 日

目　录

前言

这是 2008 春节回家期间阅读《ARM 体系结构与编程》这部作品的时候记载下来的。小小的东西凝聚了我不少的心血(汗……书又不是我写的，我只是负责抄写了一遍,而且有些内容给省略了……)，让我体会到要真真正正做个像样的东西很不容易。但是，我相信点点滴滴的积累。正所谓是：不积跬步,无以至千里；不积小流，无以成江海！

由于刚接触 ARM 架构处理器，并不熟悉其中的一些细节问题，所有在做笔记的时候对有些东西理解不是很透彻，甚至有错误。在原书的基础上有些地方加入了自己的一些东西。由于本人太菜的缘故，排版不工整，存在错别字等问题，见谅！

本文档您可以任意的修改(严禁恶搞！^_^)和传播，引用文档的部分和全部内容请保留本文件的文件头，如果您是有心人修改或完善了其中的部分内容，请一定保留您的修改记录(修改日期、版本、修改人以及修改点等)，并将其归入文件头中。

由于春节回家计算机里只有简单的编写程序时使用的编辑器，所有的表格均采用制表符完成，编辑时使用了三号新宋体字体以及演示版的 Editplus2。

非常喜欢我的事业，愿与勤奋的人分享自己的努力，一同学习进步！

Email : liyangbbs@126.com

QQ: 249456711

修改历史记录页

日期	版本	修改人	备注
01/25/2008	V0.08.0125	Metal Max	Editplus 下完成笔记
05/01/2008	V0.08.0501	Metal Max	Word 下初步完成重排版
05/11/2008	V0.08.0511	Metal Max	修改错误

1. 嵌入式系统的概述

嵌入式系统： 是指以应用为中心，以计算机技术为基础，软件和硬件可以裁剪，适应应用系统对功能的、可靠性、体积和功耗严格要求的专用计算机系统。

ARM： ACRON RISC COMPUTER ←→ADVANCED RISC COMPUTER

ARM 技术的发展历程：第一片 ARM 处理器是在 1983 年 10 月到 1985 年 4 月位于英国剑桥的 ACRON COMPUTER 公司开发的。于 1985 年 4 月 26 日在 ACRON 公司进行了首批 ARM 样片测试并成功运行了测试程序。

1990 年 11 月 ARM 公司在英国剑桥的一个谷仓里成立最初只有 12 人。

2. ARM 体系的结构和特征

2.1 ARM 芯片特点

2.1.1 RISC 体系特点

(1) 具有大量的寄存器；
(2) 绝大部分的操作就在寄存器中进行，通过 LOAD,STORE 的体系结构在内存和寄存；
(3) 器之间传递数据；
(4) 寻找方式简单；
(5) 采用固定长度的指令格式。

2.1.2 ARM 技术

(1) 在同一条数据处理指令中包含 算术逻辑处理单元 和 移位处理；
(2) 使用地址自动增加（减少）来优化程序中循环处理；
(3) LOAD/STORE 指令可以批量传输数据，从而提高数据传输的效率；
(4) 所有指令都可以根据前面指令的执行结果，决定是否执行，以提高指令的执行效率。

2.2 ARM 版本和命名

迄今为止，ARM 体系结构共定义了 6 个版本，版本号分别为 V1-V6。

2.2.1 ARM 体系版本：

(1) 版本 1：V1 体系
- 处理乘法指令以外的基本数据处理指令；
- 基于字节、字和多字的 LOAD/STORE 指令；
- 包括子程序 BL 在内的跳转指令；
- 供操作系统使用的软件中断指令：SWI；
- 本版本总地址空间是 26 位（2^26 = 64MB），目前已经不在使用。

(2) 版本 2：V2 体系
- 乘法指令和乘加指令；
- 支持协处理器的指令；
- 对于 FIQ 模式，提供额外的两个备份寄存器；
- SWP 指令及 SWPB 指令；
- 本版本总地址空间是 26 位，目前已经不在使用。

(3) 版本 3：V3 体系
- 地址扩展到 32 位，除去 3G 版本以外的其他版本是向前兼容的，支持 26 位地址空间；
- 程序状态寄存器 CPSR（Current Program Status Register）从原来的 R15 移动到新的
- 专用寄存器；
- 增加了 SPSR（Saved Program Status Register）用于异常中断程序时，保存被中断程序的状态；
- 增加了两种处理器模式，是操作系统代码可以方便地使用数据访问；
- 中止异常、指令预取异常和未定义指令异常；
- 增加了 MSR 和 MRS，用于访问 CPSR,SPSR 寄存器；
- 修改了原来版本中从异常返回的指令。

(4) 版本 4：V4 体系

- 半字的读取和写入指令；
- 读取(LOAD)带符号的字节和半字数据的指令；
- 增加了 T 变种，可以使处理器切换到 Thumb 状态；
- 增加了处理器的特权模式。在该模式下使用的是用户模式下的寄存器；
- 版本 4 中明确定义了哪些指令会引起未定义指令异常，版本 4 不再强制；
- 要求于以前的 26 位地址空间兼容。

(5) 版本 5：V5 体系

- 提高了 T 变种中 ARM/THUMB 混合使用的效率；
- 对 T 变种的指令和非 T 变种的指令使用相同的代码生成技术；
- 增加了前导零计数(COUNT LEADING ZEROS)指令，该指令可以使整数除法；
- 和中断优先级排队操作的更为有效；
- 增加了软件断点指令；
- 为协处理器提供了更多的可选择的指令；
- 更加严格的定义了乘法指令堆条件标志位的影响。

(6) 版本 6：V6 体系

- 2001 年发布，其主要特点是增加了 SIMD 功能扩展。它适合用电池供电的高性能便携式设备。

2.2.2 ARM 体系变种

(1) T(THUMB)。
(2) M(长乘法指令)。
(3) E(增强型 DSP 指令)。
(4) J(JAVA 加速器 JAZELLE)。
(5) SIMD(ARM 多媒体扩展)。

2.2.3 ARM 命名格式

表示 AMR/THUMB 体系版本的字符串是由下面几个部分组成的：

```
ARM v5   T   exP
 |    |   |   |
 |    |   |   └要排除的功能
 |    |   └变种名称
 |    └版本号
 └ARM 字符串
```

(1) ARM 字符串；
(2) ARM 版本号；
(3) X 表示排除某种功能。

2.3 ARM 处理器系列

当前主要是有以下几个系列：

(1) ARM7

ARM7TDMI,ARM7TDMI-S,ARM7EJ-S,ARM720T-4

(2) ARM9

(3) ARM9 系列处理器使用了 ARM9TDMI 内核，其中包含了 16 位的 THUMB 指令集。

(4) ARM9E

ARM9E 系列处理器使用单一的处理器内核提供了微控制器、DSP、JAVA 应用系统的解决方案，从而极大的减小了芯片的大小及复杂程度，降低了功耗缩短了产品面世的时间。

(5) ARM10E

ARM10E 系列处理器有高性能和低功耗的特点。其采用了新的节能模式，提供了 64 位的读取和写入体系，包含向量操作的满足 IEEE754 的浮点运算协处理器，系统集成更加方便，拥有完整的硬件和软件可开发工具。包含：ARM1020E,ARM1022E,ARM1026EJ-S

(6) SECURCORE

提供了基于高性能的 32 位 RISC 技术的安全解决方案。包含：SECURCORE SC100, SECURCORE SC110, SECURCORE SC200,SECURCORE SC210。

2.4 ARM 处理器模式

7 种运行模式：

(1) 用户模式 USR(USER);
(2) 快速中断模式 FIQ(FAST INTERRUPT RQUEST);
(3) 中断模式 IRQ(INTERRUPUT RQUEST);
(4) 管理模式 SVC(SUPERVISOR);
(5) 中止模式 ABT(ABORT);
(6) 未定义模式 UND(UNDEFINED);
(7) 系统模式 SYS(SYSTEM)。

2.5 ARM 寄存器

2.5.1 37 个寄存器：

第一类：31 个通用寄存器（32bit），包括 PC 在内。

第二类：6 个状态寄存器（32bit）,目前只使用了一部分的位。

表格 2.5.1 ARM 内核寄存器列表

用户模式	特权模式					
	系统模式	异常模式				
usr	sys	svc	abt	und	irq	fiq
R0 – R7						
R8-R12						R8-R12_fiq
R13		R13_svc	R13_abt	R13_und	R13_irq	R13_fiq
R14		R14_svc	R14_abt	R14_und	R14_irq	R14_fiq
PC						
CPSR						
----------		SPSR_svc	SPSR_abt	SPSR_und	SPSR_irq	SPSR_fiq

表格中寄存器的讲解：

(1) R0 - R7： 未备份寄存器。通过上表可以看出，这几个寄存器在不同模式下使用的都是同样的物理寄存器所以对它们进行操作时都要注意备份保存。

(2) R8-R12：快速中断拥有自己私有的 R8-R12。

(3) R13：这个寄存器一般用来做堆栈的指针。个种异常模式都拥有自己的堆栈指针，所以在进行处理器初始化的时候要对每种模式的堆栈区域进行。

(4) R14(LR)：顾名思义，连接寄存器。用来保护程序返回地址。可以通过下面的两种方式来实现子程序的返回：

① MOV PC, LR

② BX LR(注意：BX 指令会根据 PC 最后一位来确定是否要进入 THRUMB 状态)

③ R15(PC)：

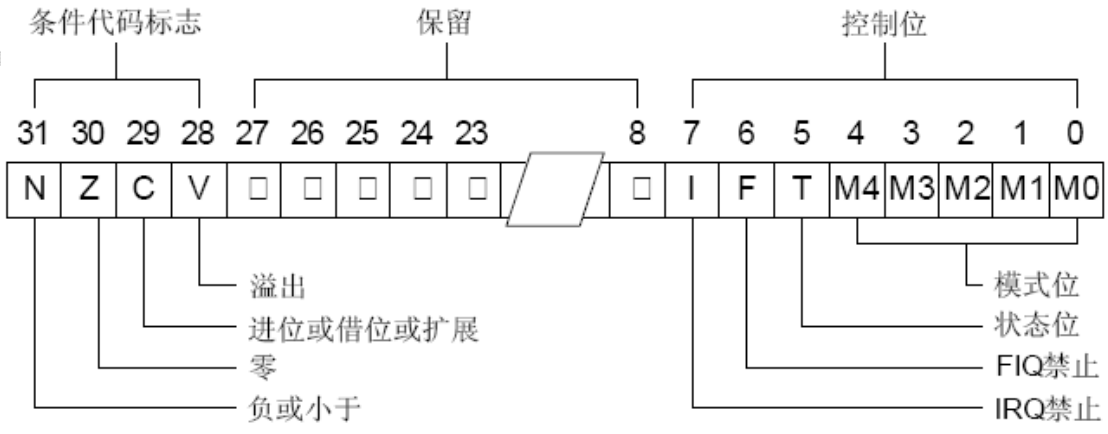
- 由于 ARM 处理器采用了流水线机制。当前的 PC 值和当前执行的指令有一定的偏移。在 ARM7 中，3 级流水，PC+8。
- 由于具体的芯片的不同，在使用 STM/STR 保存 R15 的时候，有可能保存的是当前的 PC+8,也有肯保存的是 PC+12.对于同一的芯片这个是个常量，但是如果要进行程序移植就有可能带来错误，所以要么避免使用 STM/STR 来保存 R15，要么测量出到底是 PC+8,还是 PC+12，或者其他。

测试 STM/STR 保存的 PC 的偏移量的示例：

```
PC-8--> SUB     R1, PC, #4;获取 STR 指令的地址。
PC-4--> STR     PC, [R0]   ;保存 PC + X 的值，也就是把偏移值读出来。
PC   --> LDR     R0, [R0]   ;保存偏移值到 R0 中
PC+4--> SUB     R0, R0, R1;求出偏移值。
PC+8--> XXX     X, X, X
PC+12-> XXX     X, X, X
```

- 当成功的向 R15 中写入值，则程序将跳转到该地址继续执行。由于 ARM 指令是字对齐的，所以地址的最后两位应该为 0.同理 THUMB 状态下指令是以半字对齐的，所以最后一位应该为 0.
- BX 指令对 R15 的要求。BX 指令测试 R15 的最后一位来决定是否进入到 THUMB 状态。
- MOV PC, PC 这种读取 PC 和写入 PC 不对称的指令要特别注意，由于流水线的影响，读取的 PC 值有一定的偏移。

(5) CPSR：



图表 2.5.1 CPSR 寄存器

① 以下指令会影响 CPSR 中的标志位：

- 比较指令：CMP,CMN,TEQ,TST；

- 当一些算术指令和逻辑指令的目标寄存器不是 R15 时，这些指令会影响 CPSR 中的条件标志位；
- MSR 指令可以向 CPSR 写入新的值；
- MRC 指令将 R15 作为目标寄存器时，可以把协处理器产生的条件标志位的值传送到 ARM 处理器；
- 一些 LDM 指令的变种指令可以将 SPSR 的值复制到 CPSR 中，这种操作主要用于从异常中断程序中返回；
- 一些带‘位设置’的算术和逻辑指令的变种指令，也可以将 SPSR 的值复制到 CPSR 中，这种操作主要用于从异常中断程序中返回。

(6) CPSR 中的【控制位】：I, F, T, M

M的组合：

图表 2.5.2

M[4:0]	模式	可见的 Thumb 状态寄存器	可见的 ARM 状态寄存器
10000	用户	r0~r7, SP, LR, PC, CPSR	r0~r14,PC, CPSR
10001	FIQ	r0~r7,SP_fiq,LR_fiq,PC,CPSR, SPSR_fiq	r0~r7,r8_fiq~r14_fiq,PC,CPSR, SPSR_fiq
10010	IRQ	r0~r7,SP_irq,LR_irq,PC,CPSR, SPSR_fiq	r0~r12,r13_irq,r14_irq,PC,CPSR, SPSR_irq
10011	超级用户	r0~r7,SP_svc,LR_svc,PC,CPSR, SPSR_svc	r0~r12,r13_svc,r14_svc,PC,CPSR, SPSR_svc
10111	中止	r0~r7,SP_abt,LR_abt,PC,CPSR, SPSR_abt	r0~r12,r13_abt,r14_abt,PC,CPSR, SPSR_abt
11011	未定义	r0~r7,SP_und,LR_und,PC,CPSR, SPSR_und	r0~r12,r13_und,r14_und,PC,CPSR, SPSR_und
11111	系统	r0~r7,SP,LR,PC,CPSR	

注：向 M[4:0]写入非法值将导致处理器进入不可恢复的状态。图表 2.5.3 M 组合值对应系统模式

2.6 ARM 体系的异常中断

2.6.1 ARM 中异常中断种类

表格 2.6.1 ARM 异常优先级及入口地址

优先级	异常	入口地址
1	上电复位（RESET）	0x0000 0000
2	数据中止（DAT_ABT）	0x0000 0010
3	快速中断（FIQ）	0x0000 001c
4	中断（IRQ）	0x0000 0018
5	指令预取（PRE_ABT）	0x0000 000c
6	未定义（UND）	0x0000 0004
6	软件中断（SWI）	0x0000 0008

2.6.2 ARM 异常中断响应过程

(1) ARM 处理器对异常中断的响应过程伪指令表示：

- 中断响应过程：

```
R14_<excepution_mode>     =   return link          ;保存返回地址
SPSR_<excepution_mode>    =   CPSR                 ;保护当前 CPSR
CPSR[5]                   =   0(T=0)               ;中断自动回到 ARM 状态，且仅在 ARM 状态
If(<execption_mode> ==    reset or FIQ)   then     ;禁止 FIQ,IRQ
CPSR[6]                   =   1                    ;如果时 FIQ 异常才禁止
CPSR[7]                   =   1                    ;响应异常就禁止
PC                        =   exception vector address    ;跳转到异常服务程序的地址
```

- 中断返回过程：

```
CPSR                      =   SPSR_<exception_mode>   ;恢复 CPSR
PC                        =   Return link             ;恢复执行地址
```

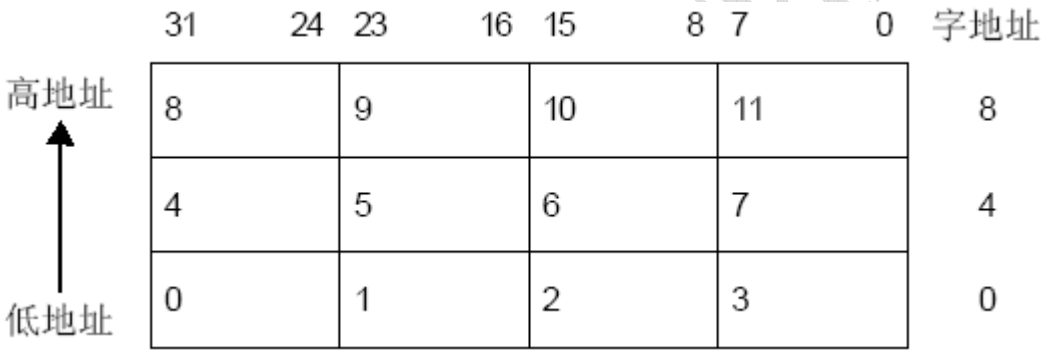
2.7 ARM 体系存储系统

2.7.1 ARM 体系存储空间。

ARM 体系使用 FLAT 模式的地址空间。可以为 2^32 字节(即 4 GB)。

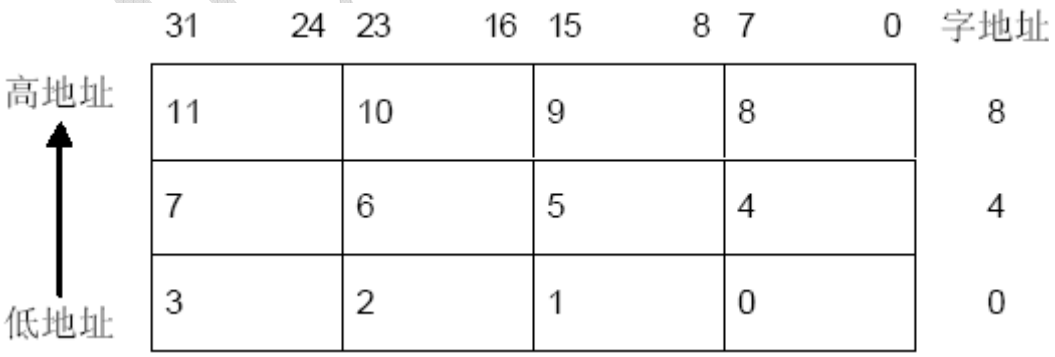
2.7.2 ARM 体系存储器格式

(1) 大端格式（字节 0 连接到 31~24 数据线）：



图表 2.7.1 存储器大端格式

(2) 小端格式（字节 0 连接到 7~0 数据线）：



图表 2.7.2 存储器小端格式

2.7.3 非对齐的存储器访问

(1) 非对齐的指令预取操作：

在 ARM 状态下非对齐即地址的最末两位不为 00，要么指令执行的结果不可预知

要么最后两位就会被忽略，同理可得 THUMB 状态的情况。

(2) 非对齐的数据预取操作：

① 执行的结果不可预知；

② 忽略字单元的最低两位值。即 Address and 0xfffffffc；

③ 忽略半字单元的最低位值。即 Address and 0xfffffffe；

④ 忽略字、半字单元的低位无效值。由存储器系统完成。

2.7.4 指令预取和自修改代码

自修改代码可能带来指令的误执行。当被修改的指令已经进入指令流水线时，虽然改指令已经被修改但是任有可能被执行一次，导致错误。慎用！

3. ARM 指令分类及寻址方式

3.1 ARM 指令集概要

3.1.1 ARM 指令的分类

(1) 跳转指令；
(2) 数据处理指令；
(3) 程序状态寄存器传输指令；
(4) LOAD/STORE 指令；
(5) 协处理器指令；
(6) 异常中断产生指令。

3.1.2 ARM 指令的编码格式

由于各种指令有很多种编码格式且有一定的重复性，具体的内容可以参加原著！

??

3.1.3 ARM 指令的条件码域

编码	助记符	含义	CPSR 中的条件标志
0000	EQ	相等	Z = 1
0001	NE	不相等	Z = 0
0010	CS/HS	无符号数大于等于	C = 1
0011	CC/LO	无符号数小于	C = 0
0100	MI	负数	N = 1
0101	PL	非负数	N = 0
0110	VS	上溢出	V = 1
0111	VC	无上溢出	V = 0
1000	HI	无符号数高于	C = 1 且 Z = 0
1001	LS	无符号数低于等于	C = 0 或 Z = 1
1010	GE	有符号数大于等于	N=1 且 V=1 或 N=0 且 V=0
1011	LT	有符号数小于	N=1 且 V=0 或 N=0 且 V=1
1100	GT	有符号数大于	Z=0 且 N=V
1101	LE	有符号素小于等于	Z=1 或 N !=V
1110	AL	无条件执行	
1111	NV	从不执行	

3.2 ARM 指令寻址方式

3.2.1 数据处理指令的操作数的寻址方式

(1) 立即数方式；
(2) 寄存器方式；
(3) 寄存器移位方式。

3.2.2 字及无符号字节的 LOAD/STORE 指令的寻址方式

(1) 各种类型的 LOAD/STORE 指令寻址方式由两部分组成。
- 基址寄存器；
- 地址偏移量。

(2) 地址偏移量可以由 3 种格式：
- 立即数；
- 寄存器；
- 寄存器移位。

3.2.3 杂类 LOAD/STORE 指令的寻址方式

LDR | STR <H/SH/SB/D> <Rd>,<Address_mode>

4. ARM 指令集

4.1 跳转指令

(1) B 及 BL 指令

表格 4.1.1 B 和 BL 指令格式

bit[31]	bit[27:25]	bit[24]	bit[23:0]
Condition	101	L	Signed_immed24

注：有 L 表示保存当前 PC 寄存器的值位于 LR 中，跳转的范围大致为：-32MB ~ +32MB

子程序的返回：

- BX R14
- MOV PC,R14
- STMFD R13!,{<registers>,R14}
- LDMFD R13!,{<registers>,PC}

Signed_immed24 的来历

将 PC 寄存器的值作为本跳转指令的基地址值，从目的地址减去基地址形成偏移地址。当上面的偏移地址超过 33554432~33554430 时，程序需要做相应的处理。否则，将指令编码字中的 Signed_immed24 设置成上述字节偏移量的。

注意:当指令跳转地址越过 0 或 32 位地址空间最高地址时，将产生不可预测的结果。

(2) BLX(1)指令

该指令完成跳转、保存 PC 到 LR 的同时切换处理器到 THUMB 状态。

bit[31:28]	bit[27:25]	bit[24]	bit[23:0]
1111	101	H	Signed_immed24

- 从程序中返回：

① BX R14

② PUSH {<registers>,R14}

③ POP {<registers>,PC}

Signed_immed24 的来历:

将 PC 寄存器的值作为本跳转指令的基地址值从目的地址减去基地址形成偏移地址，当上面的偏移地址超过 33554432~33554430 时，程序需要做相应的处理。否则，将指令编码字中的 Signed_immed24 设置成上述字节偏移量的 bits[25:2]。但是可知，现在转入了 THUMB 状态，所以要将 H 位设置成上述字节偏移量的 bit[1]。

注意：本指令时无条件执行的。指令中的 bit[24]被作为目标地址的 bit[0]。

(3) BLX(2)指令

该指令完成跳转，目标地址处可以时 ARM,THUMB 指令。目标地址放在 Rm 中，该地址的 bit[0]为 0，目标地址处的指令类型有 CPSR 中的 T 位决定。指令的伪代码：

```
if Cond passed then
LR = address of instruction after the BLX instruction
T = Rm[0]
PC = Rm AND 0xfffffffe
```

(4) BX 指令

和以上的 BLX(2)指令类似，只是不对 LR 操作。指令的伪代码为：

```
if cond passed then
T = Rm[0]
PC = Rm    AND 0xfffffffe
```

4.2 数据处理指令

4.2.1 数据传送指令

(1) MOV 指令

指令操作的伪代码：

```
if cond passed then
Rd = shifter_operand
if S==1 and Rd==R15 then
CPSR = SPSR
else if S==1 then
N = Rd[31]
Z = if Rd == 0 then 1
else 0
C = shifter_carry_out
V = unaffected
```

MOV 指令完成的功能：

- 将数据从一个寄存器传送到另外一个寄存器；
- 将一个常数传送到另外一个寄存器中；
- 实现单纯的位移操作。左移操作可以实现将操作数乘以 2^n；
- PC 作为目标寄存器可以实现程序跳转。用于实现子程序调用和返回；
- PC 作为目标寄存器且 S 位有效，则可以实现从某些异常中断中返回。

(2) MVN 指令

本指令的伪代码：

```
if cond passed then
Rd = NOT shifter_oprand     <——取反传送
if S==1 and Rd==R15 then
CPSR = SPSR
else if S==1 then
N = Rd[31]
Z = if Rd == 0 then 1
else 0
C = shifter_carry_out
V = unaffected
```

MVN 指令完成的功能：

- 向寄存器中传送一个负数；
- 生成位掩码；
- 求一个数的反码。

(3) ADD 指令

本指令的伪代码：

```
if cond passed then
Rd = Rn + shifter_oprand
if S == 1 and Rd == R15 then
CPSR = SPSR
else if S == 1 then
N = Rd[31]
Z = if Rd == 0 then 1
else 0
C = carryfrom(Rn + shifter_oprand)
V = overflowfrom(Rn + shifter_oprand)
```

ADD 指令完成的功能：

- 完成两个操作数相加。

(4) ADC 指令

本指令的伪代码：

```
if cond passed then
Rd = Rn + shifter_oprand + C
if S == 1 and Rd == R15 then
CPSR = SPSR
else if S == 1 then
N = Rd[31]
Z = if Rd == 0 then 1
else 0
C = carryfrom(Rn + shifter_oprand + C)
V = overflowfrom(Rn + shifter_oprand + C)
```

ADC 指令完成的功能：

- ADC 和 ADD 指令联合使用可以实现两个 64 位的操作数相加。

如：

```
ADD     R0,R2           ；累加低 32 位  具体的放置位置 R1R0,R3R2
ADC     R1,R3           ；带进位累加高 32 位
```

(5) SUB 指令

本指令的伪代码：

```
if cond passed then
Rd = Rn - shifter_oprand
if S == 1 and Rd==R15 then
CPSR = SPSR
else if S == 1 then
N = Rd[31]
Z = if Rd == 0 then 1
else 0
C = NOT borrowfrom(Rn - shifter_oprand)<-----进位标志取反
```

```
V = overflowfrom(Rn - shifter_oprand)
```

SUB 指令完成的功能：

- 实现两个操作数相减；
- SUBS 指令和条件跳转指令实现循环控制；

注意：在 SUBS 指令中，如果发生了借位操作，C 标志位设置成 0，反之，C 标志位设置为 1。这和 ADDS 指令中的进位指令正好相反。这主要是为了适应 SBC 指令的操作需要。

(6) SBC 指令

本指令的伪代码：

```
if cond passed then
Rd = Rn - shifter_oprand - NOT C<-----进位标志取反
if S==1 and Rd==R15 then
CPSR = SPSR
else if S == 1 then
N = Rd[31]
Z = if Rd == 0 then 1
else 0
C = NOT borrowfrom(Rn - shifter_oprand - NOT C)
V = overflowfrom(Rn - shifter_oprand - NOT C)
```

SBC 指令完成的功能：

- SUB 和 SBC 指令联合使用可以实现两个 64 位的操作数相减。

如：

```
SUBS   R4,R0,R2 ; 具体源操作数的放置位置  R1R0,R3R2
SBC    R5,R1,R3 ; 结果操作数放置位置  R5R4
```

注意：在 SBCS 中，如果发生了借位操作，CPSR 寄存器中的 C 标志位设置成 0。如果没有发生借位操作，CPSR 寄存器中的 C 标志位设置成 1。这个于 ADDS 指令中的进位指令正好相反。

(7) RSB 逆向减法指令

指令操作的伪代码：

```
if cond passed then
Rd = shifter_oprand - Rn
if S == 1 and Rd==R15 then
CPSR = SPSR
else if S == 1 then
N = Rd[31]
Z = if Rd == 0 then 1
else 0
C = NOT borrowfrom(shifter_oprand - Rn)
V = overflowfrom(shifter_oprand - Rn)
```

RSB 指令完成的功能：

```
RSB      Rd,Rx,#0; Rd = 0 - Rx   <---------逆向相减
RSB      Rd,Rx,Ry,LSL#n          ; Rd = RY*2^n - Rx
```

注意：RSBS 指令中如果发生借位，C 位将被设置为 0，反之设置为 1。这个和 ADDS 指令发生进位 C 的设置是相反的。主要是为了配合 SBC 指令的需要。

(8) RSC 带借位逆向减法指令

指令操作的伪代码：

```
if cond passed then
Rd = shifter_oprand - Rn - NOT C
if S == 1 and Rd == R15 then
CPSR = SPSR
else if S == 1 then
N = Rd[31]
Z = if Rd == 0 then 1
else 0
C = NOT borrowfrom(shifter_oprand - Rn - NOT C)
V = overflowfrom(shifter_oprand - Rn - NOT C)
```

RSB 指令完成的功能：

- 下面的指令序列可以求一个 64 位数值的负数。

```
RSBS    R4,R2,R0    ；减数：R3R2,被减数：R1R0
RSC     R5,R3,R1    ；结果：R5R4
```

注意：RSCS 指令中如果发生借位，C 位将被设置为 0，反之设置为 1。这个和 ADDS 指令发生进位 C 的设置是相反的。主要是为了配合 SBC 指令的需要。

4.2.2 算术逻辑运算指令概述

(1) AND 逻辑与操作指令

指令操作的伪代码：

```
if cond passed then
Rd = Rn AND shifter_oprand
if S == 1 and Rd==R15 then
CPSR = SPSR
else if S == 1 then
N = Rd[31]
Z = if Rd == 0 then 1
else 0
C = shifter_carry_out
V = unaffected
```

AND 指令完成的功能

- 提取某些位，也就是保留某些位。要保留的用 1 与，要清除的用 0 与。

(2) ORR 逻辑或指令

指令的伪代码：

```
if cond passed then
Rd = Rn OR shifter_oprand
if S == 1 and Rd == R15 then
CPSR = SPSR
else if S == 1 then
N = Rd[31]
```

```
Z = if Rd == then 1
else 0
C = shifter_carry_out
V = unaffected
```

ORR 指令完成的功能：

- 将某些位置 1。要置 1 的位用 1 或，保留不变的用 0 或。

(3) EOR 逻辑或指令

指令的伪代码：

```
if cond passed then
Rd = Rn EOR shifter_oprand
if S==1 and Rd==R15 then
CPSR = SPSR
else if S == 1 then
N = Rd[31]
Z = if Rd == then 1
else 0
C = shifter_carry_out
V = unaffected
```

EOR 指令完成的功能：

- 把某些位取反，要取反的就和 1 异或，保留的用 0 异或。

(4) BIC 位清除指令

指令的伪代码：

```
if cond passed then
Rd = Rn AND NOT shifter_oprand
if S == 1 and Rd==R15 then
CPSR = SPSR
else if S == 1 then
N = Rd[31]
Z = if Rd == then 1
else 0
C = shifter_carry_out
V = unaffected
```

ORR 指令完成的功能：

- 将某些位置 0。要置 0 的位用 1，保留不变的用 0。

4.2.3 比较指令概述：

(1) CMP 比较指令

指令的伪代码：

```
if cond passed then
alu_out = Rn - shifter_oprand      <——不保存结果
if S == 1 and Rd == R15 then
CPSR = SPSR
```

```
else if S == 1 then
N = alu_out[31]
Z = if alu_out    == then 1
else 0
C = NOT borrowfrom(Rn - shifter_oprand)
V = overflowfrom(Rn - shifter_oprand)
```

CMP 指令完成的功能：

- 只影响 CPSR 中的标志位，结果不保存。和 SUBS 类似。

(2) CMN 比较指令

指令的伪代码：

```
if cond passed then
alu_out = Rn + shifter_oprand       <——不保存结果
N = alu_out[31]
Z = if alu_out == then 1
else 0
C = NOT carryfrom(Rn + shifter_oprand)
V = overflowfrom(Rn + shifter_oprand)
```

CMN 指令完成的功能：

- CMN 指令将寄存器 Rn 中的值加上 shifter_oprand 表示的数值，根据加法操作的结果设置 CPSR 中相应的条件标志位。

注意：寄存器 Rn 中的值减去 shifter_oprand 表示的数值对 CPSR 中条件标志位的影响，与 Rn 中的值加上 shifter_oprand 表示的数值对 CPSR 中条件标志位的影响有细微的差别。当第 2 个操作数为 0 或者为 0x80000000 时二者结果不同，如：

```
CMP    Rn, #0                ; C=1
CMN    Rn, #0                ; C=0
```

(3) TST 位测试指令

指令操作的伪代码：

```
if cond passed then
alu_out = Rn AND shifter_oprand
N flag = alu_out[31]
Z flag = if alu_out == 0 then 1
else 0
C = shifter_carry_out
V = unaffected
```

TST 指令完成的功能：

- 测试某些位是 1 还时 0。

(4) TEQ 相等测试指令

```
if cond passed then
alu_out = Rn EOR shifter_oprand
N = alu_out[31]
Z = if alu_out==0 then 1
else 0
```

```
C = shifter_carry_out
V = unaffected
```

TEQ 指令完成的功能：

- 比较两个数是否相等，这种比较操作通常不影响 CPSR 寄存器中 V 位和 C 位。
- TEQ 指令可以用于比较两个操作数的符号是否相同，该指令执行后，CPSR 寄存器中 N 位为两个操作数符号位异或操作的结果。

4.2.4 乘法指令概述

① MUL 32 位乘法指令
② MLA 32 位带加数的乘法指令
③ SMULL 64 位有符号数乘法指令
④ SMLAL 64 位带加数的有符号数乘法指令
⑤ UMULL 64 位无符号数乘法指令
⑥ UMLAL 64 位带加数的无符号数乘法指令

以下是改系列指令详细介绍：

(1) MUL 指令

```
if cond passed then
Rd = (Rm * Rs)[31:0]
if S == 1 then
N = Rd[31]
Z = if Rd == 0 then 1
else 0
C = unaffected
V = unaffected
```

MUL 指令的使用：

由于 32 位的数相乘结果是 64 位的，而 MUL 指令仅仅保存了 64 位结果的低 32 位，所以对于带符号的和无符号的操作数来说 MUL 指令执行的结果相同。对于 ARM v5 及以上的版本，MULS 指令不影响 CPSR 寄存器中的 C 条件标志位。对于以前的版本，MULS 指令执行后，CPSR 寄存器中的 C 条件标志位数值是不确定的。寄存器 Rm，Rn，Rd 为 R15 时，指令执行的结果不可预测。

```
MUL     R0,R1,R2 ; R0=R1*R2
MULS    R0,R1,R2 ; R0=R1*R2,调试设置 CPSR 中的 N、Z 位
```

(2) MLA 指令

```
if cond passed then
Rd = (Rm * Rs)[31:0]
if S == 1 then
N = Rd[31]
Z = if Rd==0 then 1
else 0
C = unaffected
V = unaffected
```

MLA 指令的使用：

由于 32 位的数相乘结果是 64 位的，而 MUL 指令仅仅保存了 64 位结果的低 32 位，所以对于带符号的和无符号的操作数来说 MUL 指令执行的结果相同。对于 ARM v5 及以上的版本，MLA 指令不影响 CPSR 寄存器中的 C 条件标志位。对于以前的版本，MLA 指令执行后，CPSR 寄存器中的 C 条件标志位数值是不确定的。寄存器 Rm，Rn，Rd 为 R15 时，指令执行的结果不可预测。

```
MLA     R0,R1,R2,R3                     ; R0=R1*R2+R3
```

(3) SMULL 指令：

```
if cond passed then
RdHI = (Rm * Rs)[63:32]
RdLO = (Rm * Rs)[31:0]
if S==1 then
N = RdHI[31]
Z = if (RdHI == 0) and (RdLO==0)then 1
else 0
C = unaffected
V = unaffected
```

SMULL 指令的使用：

对于 ARM v5 及以上的版本，SMULL 指令不影响 CPSR 寄存器中的 C 条件标志位。对于以前的版本，SMULL 指令执行后，CPSR 寄存器中的 C 条件标志位数值是不确定的。寄存器 Rm，Rn，Rd 为 R15 时，指令执行的结果不可预测。

```
SMULL     R1,R2,R3,R4                   ; R1=(R3*R4)的低 32 位，R2=(R3*R4)的高 32 位。
```

(4) SMLAL 指令：

```
if cond passed then
RdLO = (Rm * Rs)[31:0]
RdHI = (Rm * Rs)[63:32]+RdHI+carryfrom((Rm*Rs)[31:0]+RdLO)
if S==1 then
N = RdHI[31]
Z = if (RdHI == 0) and (RdLO == 0)then 1
else 0
C = unaffected
V = unaffected
```

SMLAL 指令的使用：

- 对于 ARM v5 及以上的版本，SMLAL 指令不影响 CPSR 寄存器中的 C 条件标志位。
- 对于以前的版本，SMLAL 指令执行后，CPSR 寄存器中的 C 条件标志位数值是不确定的。
- 寄存器 Rm，Rn，Rd 为 R15 时，指令执行的结果不可预测。

(5) UMULL 指令：

```
if cond passed then
RdHI = (Rm*Rs)[63:32]
RdLO = (Rm*Rs)[31:0]
if S==1 then
N = RdHI[31]
Z = if (RdHI == 0) and (RdLO == 0) then 1
```

```
else 0
C = unaffected
V = unaffected
```

UMULL 指令完成的功能：

- 对于 ARM v5 及以上的版本，UMULLS 指令不影响 CPSR 中 C 条件标志和 V 条件标志
- 对于以前的版本 UMULLS 指令执行后，CPSR 寄存器中的 C 条件标志位数值是不确定的。
- 寄存器 Rm，Rn，RdLO,RdHI 位 R15 时指令的执行结果不可预测。

(6) UMLAL 指令：

```
if cond passed then
RdLO = (Rm*Rs)[31:0] + RdLO
RdHI = (Rm*Rs)[63:32] + RdHI + carryfrom((Rm*Rs)[31:0]+RdLO)
if S == 1 then
N = RdHI[31]
Z = if (RdHI == 0) and (RdLO == 0) then 1
else 0
C = unaffected
V = unaffected
```

UMLAL 指令完成的功能：

- 对于 ARM v5 及以上的版本，UMLALS 指令不影响 CPSR 中 C 条件标志和 V 条件标志
- 对于以前的版本 UMLALS 指令执行后，CPSR 寄存器中的 C 条件标志位数值是不确定的。
- 寄存器 Rm，Rn，RdLO,RdHI 位 R15 时指令的执行结果不可预测。

4.2.5 杂类的算术指令概述

(1) CLZ 指令

本条指令主要用于计算操作数最高端 0 位的个数。主要用于两种场合：

- 计算操作数规范化(使其最高位为 1)时所需要左移的位数。
- 确定一个优先级掩码中最高优先级(最高位的优先级)。

指令的语法格式：

```
CLZ <cond> <Rd>,<Rm>
```

指令操作的伪代码：

```
if Rm == 0
Rd == 32
else
Rd = 31-(bit position of most significant "1" in Rm)
```

指令的使用：

```
CLZ     Rd,Rm
MOVS    Rm,Rm,LSL Rdl
```

4.2.6 状态寄存器访问指令概述

```
MRS     Register <--- CPSR
MSR     CPSR       <--- Register
```

(1) MRS 指令：

本指令的伪代码：

```
if cond passed then
if R == 1 then
Rd = SPSR
else
Rd = CPSR
```

本指令应用场合：

- 通常通过"读-修改-写"操作序列修改状态寄存器的内容。
- 当异常中断允许嵌套时，需要在进入异常中断之后，嵌套中断发生之前处理器模式对应的 SPSR。这时需要通过 MRS 指令读出 SPSR 的值，再用其他指令将 SPSR 值保存起来。
- 在进程切换时也需要保存当前状态寄存器的值。

(2) MSR 指令

本指令的伪代码：

```
if cond passed then
if opcode[25] == 1
operand = 8_bit_immediate rotate_right (rotate_imm * 2)
else
operaand = Rm
if R == 0 then
if field_mask[0] == 1 and InAPrivilegedMode() then
CPSR[7:0] = operand[7:0]
if field_mask[1] == 1 and InAPrivilegedMode() then
CPSR[15:8] = operand[7:0]
if field_mask[2] == 1 and InAPrivilegedMode() then
CPSR[23:16] = operand[7:0]
if field_mask[3] == 1 and InAPrivilegedMode() then
CPSR[31:24] = operand[7:0]
else
if field_mask[0] == 1 and CurrentModeHasSPSR() then
CPSR[7:0] = operand[7:0]
if field_mask[1] == 1 and CurrentModeHasSPSR() then
CPSR[15:8] = operand[7:0]
if field_mask[2] == 1 and CurrentModeHasSPSR() then
CPSR[23:16] = operand[7:0]
if field_mask[3] == 1 and CurrentModeHasSPSR() then
CPSR[31:24] = operand[7:0]
```

本指令应用场合：

- 本指令通常用于恢复 PSR 的内容和改变 PSR 的内容。
- 当退出异常中断时，如果事先保存了 PSR 的内容通常通过 MSR 指令将事先保存的状态寄存器恢复到 PSR 中。
- 当需要修改 PSR 的内容时，通过"读-修改-写"指令序列完成。
- 考虑到指令的执行效率，通常在 MSR 指令中指定指令将要修改的位域。但是，当进程切换到应用场合，应指定 SPSR_fsxc,这样将来 ARM 扩展了当前未用的一些位后，程序还可以正常的运行。
- 当需要修改 PSR 位域包含未分配的位时，最好不要使用带立即数方式的 MSR 指令。

4.2.7 LOAD/STORE 内存访问指令概述

① LDR 字加载
② LDRB 字节加载
③ LDRBT 用户模式的字节数据加载
④ LDRH 半字加载
⑤ LDRSB 有符号字节数据加载
⑥ LDRSH 有符号半字数据加载
⑦ LDRT 用户模式的字数据加载
⑧ STR 字写入
⑨ STRB 字节写入
⑩ STRBT 用户模式的字节写入
⑪ STRH 半字写入
⑫ STRT 用户模式的字写入

(1) LDR 指令：

本指令伪代码：

```
if cond passed then
if address[1:0] == 0b00 then
value = Memory[address, 4]
else if address[1:0] == 0b01 then
value = Memory[address,4] rotate_right 8
else if address[1:0] == 0b10 then
value = Memory[address,4] rotate_right 16
else
value = Memory[address,4] rotate_right 24
if (Rd is R15) then
if (architecture version 5 or above) then
PC = value AND 0xfffffffe
T    = value[0]
else
PC = value AND 0xfffffffc
else
Rd = value
```

本指令的使用：

用于从内存中读取 32 位字数据到通用寄存器中，然后可以在该寄存器中对该数据进行一定的操作。

- 当 PC 作为指令中的目标寄存器时，指令可以实现程序跳转的功能。
- 当 PC 作为 LDR 指令的目标寄存器时，指令从内存中读取到的字数据当被当作目标地制值。指令执行后，将从目标地址处开始执行。在 ARMv5 及其以上版本中，地址值的 bit[0]用来指示目标地址处程序的状态，当 bit[0]为 1 时目标地址的指令为 THUMB 指令，同理，当 bit[0]为 0 时目标地址处的指令为 ARM 指令。在 ARMv5 以上的版本中地址值的 bit[0]将被忽略，程序将继续执行在 ARM 状态下。

指令示例：

```
LDR     R0,[R1,#4]            ;R0 = [R1+4]
LDR     R0,[R1,-#4]           ;R0 = [R1-4]
LDR     R0,[R1,R2]            ;R0 = [R1+R2]
LDR     R0,[R1,R2,LSL #2] ;R0 = [R1+R2*4]

LDR     R0,[R1,#4]!           ;R0 = [R1+4],R1 = R1+4
LDR     R0,[R1,-#4]           ;R0 = [R1-4],R1 = R1-4
LDR     R0,[R1],#4            ;R0 = [R1],R1 = R1+4
LDR     R0,[R1],R2            ;R0 = [R1],R1 = R1+R2
LDR     R0,[R1],R2,LSL #2 ;R0 = [R1],R1 = R1+R2*4
```

(2) LDRB 指令：

本指令伪代码：

```
if cond passed then
Rd = Memory[address,1]
```

本指令的使用：

- 用于从内存中读取 8 位字节数据到通用寄存器中，然后可以在该寄存器中对该数据进行一定的操作。
- 当 PC 作为指令中的目标寄存器时，指令可以实现程序跳转的功能。

指令示例：

```
LDRB    R0,[R2,#3]            ;R0[7:0] = [R2+3],R0[31:8] = 0
```

(3) LDRBT 指令：

本指令伪代码：

```
if cond passed then
Rd = Memory[address,1]
```

本指令的使用：

- 用于从内存中读取 8 位字节数据到通用寄存器中，然后可以在该寄存器中对该数据进行一定的操作。
- 当在特权级别的处理器模式下使用该指令，内存系统将该操作当作时一般用户模式下的内存访问操作。

指令示例：（注意模式）

```
LDRBT   R0,[R2,#3]            ;R0[7:0] = [R2+3],R0[31:8] = 0
```

(4) LDRH 指令：

本指令伪代码：

```
if cond passed then
if address[0] = 0
data = Memory[address,2]
else
data = UNPREDICTABLE
Rd = data
```

本指令的使用：

- 用于从内存中读取 16 位半字数据到通用寄存器中，然后可以在该寄存器中对该数据进行一定的操作。
- 当 PC 作为指令中的目标寄存器时，指令可以实现程序跳转的功能。

指令示例：

```
LDRH    R0,[R2,#3]          ;R0[15:0] = [R2+3],R0[31:16] = 0
```

(5) LDRSB 指令：

本指令伪代码：

```
if cond passed then
data = Memory[address,1]
Rd = SignExtend(data)
```

本指令的使用：

- 用于从内存中读取 8 位有符号的字节数据到通用寄存器，然后在该寄存器中用该 8 位符号数的符号位填充高 24 位，将其扩展成 32 位有符号数。
- PC 作为目标寄存器时，指令可以实现程序跳转的功能。

指令示例：

```
LDRSB    R0,[R2,#3]          ;R0[7:0] = [R2+3],R0[31:8] =  字节符号
```

(6) LDRSH 指令：

本指令伪代码：

```
if cond passed then
if address[0] = 0
data = Memory[address,2]
else
data = UNPREDICTABLE
Rd = SignExtend(data)
```

本指令的使用：

- 用于从内存中读取 16 位有符号的半字数据到通用寄存器，然后在该寄存器中用该 16 位符号数的符号位填充高 16 位，将其扩展成 32 位有符号数。
- PC 作为目标寄存器时，指令可以实现程序跳转的功能。

指令示例：

```
LDRSH    R0,[R2,#3]          ;R0[15:0] = [R2+3],R0[31:16] =  字节符号
```

(7) LDRT 指令：

本指令伪代码：

```
if cond passed then
if address[1:0] == 0b00
Rd = Memory[address,4]
else if address[1:0] == 0b01
Rd = Memory[address,4] rotate_right 8
else if address[1:0] == 0b10
Rd = Memory[address,4] rotate_right 16
else
Rd = Memory[address,4] rotate_right 24
```

本指令的使用：

- 用于从内存中读取一个 32 位的字数据到目标寄存器中，如果指令中的寻址方式确定的地址不时字对齐的，则从内存中读出的数值要进行循环右移操作。移位的位数为寻址方式确定的地址 bits[1:0]的 8 倍。这样对于 little-endian 的内存模式想要读取的字节数据存放在目标寄存器的低 8 位；对于 big-endian 的内存模式想要读取的字节数据存放在目标寄存器的 bits[32:24](寻址方式确定的地址 bit[0]为 0)或者存放在 bits[15:8](寻址方式确定的地址 bit[0]为 1)当处在特权级处理器模式下使用本指令时，内存系统将该操作当作一般用户模式下的内存操作。

(8) STR 指令

本指令的伪代码：

```
if cond passed then
Memory[address,4] = Rd
```

本指令的使用：

- 用于将一个 32 位的字数据写入到指令指定的内存单元。

指令示例：

```
STR     R0,[R1,#0X100]                  ; R0 ——> [R1+0X100]
STR     R0,[R1],#8                      ; R0 ——> [R1], R1 = R1+8
```

(9) STRB 指令

本指令的伪代码：

```
if cond passed then
Memory[address,1] = Rd[7:0]
```

本指令的使用：

- 用于将一个 32 位的字数据写入到指令指定的内存单元。

指令示例：

```
STR     R0,[R1,#0X100] ; R0 ---> [R1+0X100]单元的低 8 位
STR     R0,[R1],#8      ; R0 ---> [R1]单元低 8 位, R1 = R1+8
```

(10) STRH 指令

本指令的伪代码：

```
if cond passed then
if address[0] = 0
data = Rd[15:0]
else
```

```
data = UNPREDICTABLE
Memory[address,2] = data
```

本指令的使用：

- 用于将寄存器低 16 位半字数据写入到指令指定的内存单元。

指令示例：

```
STR     R0,[R1,#0X100] ; R0 ——> [R1+0X100]
STR     R0,[R1],#8      ; R0 ——> [R1], R1 = R1+8
```

(11) STRT 指令

本指令的伪代码：

```
if cond passed then
Memory[address,4] = Rd
```

本指令的使用：

- 异常中断程序在特权级的处理器模式下执行，这时如果需要按照用户模式的权限内存访问操作。

指令示例：

```
STR     R0,[R1,#0X100] ; R0 ——> [R1+0X100]
STR     R0,[R1],#8      ; R0 ——> [R1], R1 = R1+8
```

4.2.8 批量 LOAD/STORE 内存访问指令概述

① LDM(1) 批量内存字数据读取指令

② LDM(2) 用户模式的批量内存字数据

③ LDM(3) 带状态寄存器的批量内存字数据读取指令

④ STM(1) 批量内存字数据写入指令

⑤ STM(2) 用户模式的批量内存字数据写入指令

(1) LDM 指令

本指令的格式：

```
LDM{<cond>}<addressing_mode><Rn>{!},<registers>
```

本指令的伪代码：

```
if cond passed then
address = start_address
for i = 0 to 14
if register_list[i] == 1 then
Ri = Memory[address,4]
address = address + 4
if register_list[15] == 1 then
value = Memory[address,4]
if (architecture version 5 or above) then
PC = value AND 0xfffffffe
T  = value[0]
else
PC = value AND 0xfffffffc
```

```
address = address + 4
assert end_address = address - 4
```

指令的使用：

- 如果指令中基址寄存器 Rn 在寄存器列表 registers 中，而且指令中寻址方式指定指令执行后更新基址寄存器 Rn 的值，则指令执行会产生不可预知的结果。

(2) LDM(2)指令

本指令格式：

```
LDM{<cond>}<addressing_mode><Rn>,<registers_without_pc>^
```

本指令的伪代码：

```
if cond passed then
address = start_address
for i = 0 to 14
if register_list[i] == 1 then
Ri_usr = Memory[address,4]
address = address + 4
assert end_address = address - 4
```

指令的使用：

- 本指令后面不能紧跟访问备份寄存器(bank registers)的指令，最好跟一条 NOP 指令。
- 用户模式和系统模式下使用本指令会产生不可预知的结果。
- 指令中的基址寄存器是指令执行时的当前处理器模式独有的无聊寄存器，而不是用户模式对应的寄存器。
- 本指令忽略指令中内存地址的低 2 位，而不像 LDM(1)指令那样进行数据的循环右移操作。
- 异常中断程序是在特权级的处理器模式下执行的，这时如果需要按照用户模式的权限访问内存，可以是使用 LDM(2)指令。

(3) LDM(3)指令

本指令格式：

```
LDM{<cond>}<addressing_mode><Rn>,<registers_and_pc>^
```

本指令的伪代码：

```
if cond passed then
address = start_address
for i = 0 to 14
if register_list[i] == 1 then
Ri = Memory[address,4]
address = address + 4
CPSR = SPSR
value = Memory[address,4]
if (architecture version 4T, 5 or above) and (T==1) then
PC = value AND 0xfffffffe
else
PC = value AND 0xfffffffc
address = address + 4
```

```
assert end_address = address - 4
```

指令的使用：

- 如果指令中基址寄存器 Rn 在寄存器列表 registers 中，而且指令中寻址方式指定指令执行后更新基址寄存器 Rn 的值，则指令执行会产生不可预知的结果。
- 本指令主要用于从异常中断模式下返回，如果在用户模式和系统模式下使用该指令，会产生不可预知的结果。

(4) STM(1)指令

本指令格式：

```
STM{<cond>}<address_mode><Rn>{!},<registers>
```

本指令的伪代码：

```
if cond passed then
address = start_address
for i = 0 to 15
if register_list[i] == 1
Memory[address,4] = Ri
address = address + 4
assert end_address == address - 4
```

本指令的使用：

- 将指令列表中的各寄存器数值写入到连续的内存单元中。它主要用于数据块的写入、数据栈操作已经进入子程序时保存相关的寄存器的操作。如果指令中基址寄存器在寄存器列表中，而且在指令中寻址方式指定指令执行后更新基址寄存器的值，则当基址寄存器在寄存器列表中的编号最小的寄存器时，则将基址寄存器的值保存到内存中，否则，指令执行将产生不可预知的后果。

(5) STM(2)指令

本指令格式：

```
STM{<cond>}<address_mode><Rn>,<registers>^
```

本指令的伪代码：

```
if cond passed then
address = start_address
for i = 0 to 15
if register_list[i] == 1
Memory[address,4] = Ri_usr
address = address + 4
assert end_address == address - 4
```

本指令的使用：

- 本指令主要用于从异常中断模式下返回，如果在用户模式或系统模式下使用该指令，会产生不可预知的结果。
- 本指令后面不能紧跟访问备份寄存器的指令，最好跟一条 NOP 指令。
- 指令中的基址寄存器时指令执行时的当前处理器模式对应的物理寄存器，而不是用户模式对应的寄存器。

4.2.9 信号量操作指令概述

信号量用于进程间的同步和互斥，对信号量的操作通常要求是一个原子操作，即在一条指令中完成信号量的读取和修改操作。ARM 提供了如下两条指令完成信号量的操作。

① SWP 交换指令

② SWPB 字节交换指令

(1) SWP 指令

本指令格式：

```
SWP{<cond>} <Rd>,<Rm>,[<Rn>]
```

本指令的伪代码：

```
if cond passed then
if Rn[1:0] == 0b00 then
temp = Memory[Rn,4]
else if Rn[1:0] == 0b01 then
temp = Memory[Rn,4] rotate_right 8
else if Rn[1:0] == 0b10 then
temp = Memory[Rn,4] rotate_right 16
else
temp = Memory[Rn,4] rotate_right 24
Memory[Rn,4] = Rm
Rd = temp
```

本指令的使用：

- 主要用于实现信号量操作。

本指令示例：

```
SWP     R1,R2,[R3] ; R1 <——  [R3] then [R3] <——  R2
SWP     R1,R1,[R2] ; R1 <——  [R2] then [R2] <——R1
```

(2) SWPB 指令

本指令格式：

```
SWP{<cond>} <Rd>,<Rm>,[<Rn>]
```

本指令的伪代码：

```
if cond passed then
if Rn[1:0] == 0b00 then
temp = Memory[Rn,4]
else if Rn[1:0] == 0b01 then
temp = Memory[Rn,4] rotate_right 8
else if Rn[1:0] == 0b10 then
temp = Memory[Rn,4] rotate_right 16
else
temp = Memory[Rn,4] rotate_right 24
Memory[Rn,4] = Rm
Rd = temp
```

本指令的使用：

- 主要用于实现信号量操作。

本指令示例：

```
SWPB   R1,R2,[R3] ; R1[7:0] <——  [R3],R1[31:8]=0 then [R3] <——  R2[7:0]
SWPB   R1,R1,[R2] ; R1[7:0] <——  [R2],R1[31:8]=0 then [R2] <——  R1[7:0]
```

4.2.10 异常中断产生指令概述

① SWI 软中断指令

② BKPT 断点中断指令

(1) SWI 指令

本指令格式：

```
SWI {<cond>} <Immed_24>
```

本指令伪代码：

```
if cond passed then
R14_svc = address of next instruction after the SWI instruction
SPSR_svc = CPSR
CPSR[4:0] = 0b10011                    /*Enter the Supervisor mode*/
CPSR[5]    = 0                         /*ARM state*/
CPSR[7]    = 1                         /*Disable normal interruputs*/
if high vectors configured then
PC = 0xffff0008
else
PC = 0x00000008
```

本指令的使用：

- 指令中的 24 位立即数指定了用户请求 ide 服务类型，参数通过通用寄存器传递。
- 指令中的 24 位立即数被忽略，用户请求的服务类型有寄存器 R0 的数值决定，参赛通过其他的通用寄存器传递。

(2) BKPT 指令

本指令格式：

```
BKPT <Immed_16>
```

本指令伪代码：

```
if (not overridden by debug hardware) then
R14_abt = address of BKPT instruction + 4
SPSR_svc = CPSR
CPSR[4:0] = 0b10011                    /*Enter the Supervisor mode*/
CPSR[5]    = 0                         /*ARM state*/
CPSR[7]    = 1                         /*Disable normal interruputs*/
if high vectors configured then
PC = 0xffff000c
else
PC = 0x0000000c
```

本指令的使用：

- 本指令主要供软件测试程序使用。

4.2.11 ARM 协处理指令概述

① CDP 协处理器数据操作指令；
② LDC 协处理器数据读取指令；
③ STC 协处理器数据写入指令；
④ MCR ARM 寄存器到协处理器的数据传送指令；
⑤ MRC 协处理器寄存器到 ARM 寄存器的传送指令。

(1) CDP 指令

指令语法格式：

```
CDP{<cond>} <coproc>,<CRd>,<CRn>,<CRm>,<opcode_2>
CDP{<cond>} <coproc>,<opcode_1>,<CRd>,<CRn>,<opcode_2>
```

coproc：协处理器的编号。

opcode_1：协处理器将执行的操作码。

CRd：协处理器的目的寄存器。

CRn：协处理器第一操作数存放寄存器

CRm：协处理器第二操作数存放寄存器。

本指令伪代码：

```
if cond passed then Coprocessor[cp_num] -dependent operation
```

本指令的使用：

- 通知 ARM 协处理器执行特定的操作。该操作不涉及 ARM 寄存器和内存单元。本指令示例：

```
;P5 号协处理器、操作码 1 为 2，操作码 2 为 4、目的协处理器寄存器 C12，源操作数协处理器寄存器为 C10 和 C3
CDP       p5, 2, c12, c10, c3, 4
```

(2) LDC 指令

指令语法格式：

```
LDC{<cond>}{L} <coproc>,<CRd>,<addressing_mode>
LDC2{L} <coproc>,<CRd>,<addressing_mode>     <---无条件执行
```

coproc：协处理器的编号

opcode_1：协处理器将执行的操作码

CRd：协处理器的目的寄存器

本指令伪代码：

```
if cond passed then
address = start_address
load Memory[address,4] for coprocessor[cp_num]
while (NotFinished(Coprocessor[cp_num]))
address = address + 4
load Memory[address,4] for coprocessor[cp_num]
assert address == end_address
```

本指令的使用：

- 从一系列连续的内存单元将数据读取到协处理器的寄存器中。

```
;将 ARM 处理器寄存器[R2+4]指向的内存单元数据传输到 P6 号协处理器寄存器 CR3 中。
CDP p6,CR3,[R2,#4]
```

(3) STC 指令

指令语法格式：

```
STC{<cond>}{L} <coproc>,<CRd>,<addressing_mode>
STC2{L} <coproc>,<CRd>,<addressing_mode>     <——无条件执行
```

coproc：协处理器的编号

opcode_1: 协处理器将执行的操作码

CRd：协处理器的目的寄存器

本指令伪代码：

```
if cond passed then
address = start_address
Memory[address,4] from coprocessor[cp_num]
while (NotFinished(Coprocessor[cp_num]))
address = address + 4
Memory[address,4] = value from coprocessor[cp_num]
assert address == end_address
```

本指令的使用：

- 将协处理器的寄存器中的数据写入到一系列连续的内存单元中。

```
;P8 号协处理器寄存器 CR8 中的数据写入到 ARM 处理器寄存器[R2+4]指向的内存单元。
STC p8,CR8,[R2,#4]
```

MCR 指令

指令语法格式：

MCR{<cond>} <coproc>,<opcode_1>,<Rd>,<CRn>,<Rm>,{<opcode_2>}

MCR<coproc>,<opcode_1>,<Rd>,<CRn>,<CRm>,{<opcode_2>}

指令的伪代码：

```
if cond passed then
send Rd value to coprocessor[cp_num]
```

指令的使用：

- 将 ARM 寄存器的数据传送到协处理器的寄存器中。指令示例：

```
MCR P14,3,R7,C7,C11,6 ;略
```

(4) MRC 指令

指令语法格式：

```
MRC{<cond>} <coproc>,<opcode_1>,<Rd>,<CRn>,<Rm>,{<opcode_2>}
MRC<coproc>,<opcode_1>,<Rd>,<CRn>,<CRm>,{<opcode_2>}
```

指令的伪代码：

```
if cond passed then
data = value from coprocessor[cp_num]
if Rd is R15 then
```

```
N = data[31]
Z = data[30]
C = data[29]
V = data[28]
else
Rd = data
```

指令的使用：

- 将协处理器的寄存器中的数据传送到 ARM 寄存器中。指令示例：

```
MRC P15,2,R5,C0,C2,4 ;略
```

4.3 基本 ARM 指令功能段

4.3.1 逻辑运算指令的应用。

4.3.2 位操作指令应用举例

下面的代码将 R2 中的高 8 位传送到 R3 的低 8 位中。

```
MOV     R0, R2, LSR #24 ;
ORR     R3, R0, R3, LSL#8
```

4.3.3 实现乘法的指令段举例

```
MOV     R0, R0, LSL #n ;R0 = R0*(2^n)
ADD     R0, R0, R0, LSL #n ;R0 = R0+R0*(2^n)
RSB     R0, R0, R0, LSL #n ;R0 = R0*2^n - R0
ADD     R0, R1, R0, LSL #1 ;R0 = R1 + R0*2^1
```

4.3.4 64 位数据运算举例

数据存放格式为： R1R0, R3R2

```
R1R0 = R1R0 + R3R2
ADDS    R0, R0, R2 ;低 32 位加法
ADDC    R1, R1, R3 ;高 32 位带进位加法
R1R0 = R1R0 - R3R2
SUBS    R0, R0, R2 ;
SBC     R1, R1, R3 ;
```

下面的指令实现两个 64 个数据的比较操作，并正确设置 N,Z,C 条件标志，V 标志可能有错。

```
CMP     R1, R3 ;比较高 32 位
CMPEQ   R0, R2 ;如果高 32 位相等，比较低 32 位
```

4.3.5 转换内存数据格式指令段

(1) 以下指令实现小端数据存储格式到大端数据格式的转换。

执行前： R0 = ABCD 执行后： R0 = DCBA

```
EOR     R1, R0, R0, ROR #16 ; R1 = A^C,B^D,C^A,D^B
BIC     R1, R1, #0x00FF0000 ; R1 = A^C,0,C^A,D^B
MOV     R0, R0, ROR #8      ; R0 = D,A,B,C
```

```
EOR      R0, R0, R1, LSL #8   ; R0 = D,C,B,A
```

(2) 下面的指令用于转换大量的字数据的存储方式。

指令执行前需要转换的数据放在 R0 中，其存储方式为： R0 = ABCD 指令执行后 R0 中存储方式为：R0 = DCBA

```
MOV      R2, #0XFF ;
MOV      R2, R2, #0X00FF0000 ;
```

重复下面的指令段，实现数据存放方式的转换

```
AND      R1, R2, R0                ;R1 = 0 B 0 D
AND      R0, R2, R0, ROR #24 ;R0 = 0 C 0 A
ORR      R0, R0, R1, ROR #8   ;R0 = D C B A
```

4.3.6 跳转指令的应用

(1) 子程序调用；

```
...
BL      FUNCTION
...
FUNCTION
...
MOV     PC, LR
```

(2) 条件执行；

求两个数的公约数。

```
GCD      CMP     R0, R1
SUBGT R0, R0, R1
SUBLT R1, R1, R0
BNE     GCD
MOV     PC, LR
```

(3) 条件判断语句；

```
CMP      R0, #0
CMPNE    R1, #1
ADDEQ    R2, R3, R4
```

(4) 循环语句；

```
MOV      R0, #LOOPCOUNT
LOOP     ...
SUBS     R0, R0, #1
BNE      LOOP
...
```

(5) 多路分支程序语句。

```
CMP        R0, #MAXINDEX                    ;判断跳转索引是否在范围内
ADDLO    PC, PC, R0, LSL #ROUTINESIZELOG2

Index0handler
...
Index1handler
```

```
...
Index2handler
...
```

4.3.7 LOAD/STORE 指令的应用。

(1) 链表操作；

下面的代码段在链表中搜索于某以数据相等的元素。链表的每个元素包括两个字，第 1 个值中包含一个字节数据。第 2 个字中包含指向下一个链表元素的指针，当这个指针位 0 表示链表结束。代码指向前 R0 指向链表的头元素，R1 中存放将要搜索的数据。代码执行后 R0 指向第 1 个匹配的元素，当没有匹配元素时，R0 为 0。

```
LLSEARCH    CMP     R0, #0
LDRNEB R2, [R0]
CMPNE   R1, R2
LDRNE   R0, [R0, #4]
BNE     LLSEARCH
MOV     PC, LR
```

(2) 简单的串比较；

下面的代码段实现比较两个串的大小。代码执行前，R0 指向第 1 个串，R1 指向第 2 个产。代码执行后 R0 中保存比较的结果，如果两个串相同，R0 为 0.如果第 1 个串大于第 2 个串，R0>0，如果 1 串小于 2 串，R0<1。

```
STRCMP      LDRB    R2, [R0], #1
LDRB    R3, [R1], #1
CMP     R2, #0
CMPNE   R3, #0
BEQ     RETURN
CMP     R2, R3
BEQ     STRCMP
RETURN      SUB     R0, R2, R3
MOV     PC, LR
```

(3) 长跳转；

通过直接向 PC 寄存器中读取字数据，程序可以实现 4GB 的地址空间的任意跳转，这种跳转叫长跳转。

```
ADD   LR, PC, #4      ;LR 中放 return_here 的地址，以便返回。
LDR   PC, [PC, #-4]   ;读取 FUNCTION 地址。
DCD   FUNCTION        ;分配 FUNCTION 地址
return_here           ;
```

(4) 多路跳转；

下面的代码段通过函数地址表实现多路跳转。其中，MAXINDEX 为跳转的最大索引号，R0 中为跳转的索引号。

```
CMP     R0, #MAXINDEX
LDRLO   PC, [PC, R0, LSL #2]
```

```
B          IndexOutOfRange
DCD        Handler0
DCD        Handler1
DCD        Handler2
DCD        Handler3
...
```

4.3.8 批量 LOAD/STORE 指令的应用。

(1) 简单的块复制；

R12 为被复制块首地址，R13 为目的块首地址。R14 为源数据区末地址。

```
LOOP     LDMIA      R12!, (R0-R11)
STMIA      R13!, (R0-R11)
CMP        R12, R14
BLO        LOOP
```

(2) 子程序进入和退出时数据的保存和恢复；

```
function      STMFD R13!, {R4-R12, R14}
...
Insert the function body here
...
LDMFD R13!, {R4-R12, R14}
```

4.3.9 信号量指令的应用。

信号量用于实现对临界区数据访问的同步。代码中用进程标识符来表示个信号量的所有者，大门执行前进程的标识符保存在 R1 中，信号量的地址保存在 R0 中当信号量值位 0 时，示与该信号量相关的临界区可以用。当信号量值位-1 时，表示当前有进程正在查看该信号量的值。如果当前进程查看的信号量正忙，当前进程将一直等待该信号量。为了避免当前进程的查询操作阻塞操作系统的进程调度，可以在下一次查询之前调用操作系统中系统调用，使当前进程休眠一段时间。

```
MVN     R2, #0          ;R2 = -1
SPININ   SWP     R3, R2, [R0] ;将信号量的值读入 R3,同时其值设置为-1
CMN     R3, #1          ;判断读取到的信号量，判断是否有其他进程访问该信号量
...
;如果有其他进程正在访问该信号量，则使当前进程休眠一段时间，以保证系统能够进行任务调度。
...
BEQ     SPININ   ;如果有其他进程访问该信号量，跳转到 SPININ
CMP     R3, #0     ;判断当前信号量是否可用，即 R0 值是否为 0，当;不为 0，表其他的进程正拥有该信号量。
STRNE R3, [R0]          ;这时恢复该信号量的值，即该信号量拥有者的;进程标识符。
...
;如果该信号量正被别的进程占用，则使当前进程休眠一段时间，以保证操作系统能够进行任务调度。
...
BNE     SPININ          ;重新获取该信号量。
STR     R1, [R0]        ;当进程得到该信号量，将自己的进程标识符写入到内存单元 R0 处。
...                     ;这里时该信号量所保护的临界区数据
...
```

```
SPINOUT
SWP     R3, R2, [R0]
CMN     R3, #1
...
BEQ     SPINOUT
CMP     R3, R1
BNE     CorruptSemaphore
MOV     R2, #0
STR     R2, [R0]
```

4.3.10 与系统相关的一些指令代码段

(1) SWI 中断处理程序示例;

SWI 指令执行时通常完成下面的工作：

```
R14_svc = SWI 下面一条指令的地址
SPSR_svc = CPSR
CPSR[4:0] = 0B10011
CPSR[5] = 0
CPSR[7] = 1
if high vectors configured then
PC = 0XFFFF0008
else
PC = 0X00000008
```

SWI 处理的程序的基本框架：

① 发生 SWI 中断

② 跳转到 0X00000008 处执行跳转

③ 提取 SWI 指令中的立即数确定响应的中断号

基本程序：

```
SWIHandler      STMFD   SP!, {R0-R3, R12, LR}      ;资源保护
MRS     R0, SPSR      ;是否位 ARM 状态
TST     R0, #0X20
LDRNEH R0, [LR, #-2]                ;THUMB 状态
BICNE   R0, R0, #0XFF00
LDREQ   R0, [LR, #-4]                ;ARM 状态
BICEQ   R0, R0, #0XFF000000
CMP     R0, #maxSWI
LDRLS   PC, [PC, R0, LSL #2]        ;取得服务地址
B       SWIOutOfRange

SWITABLE        DCD     DO_SWI_0        ;SWI 0 号服务程序
DCD     Do_SWI_1
...
```

(2) IRQ 中断处理程序示例。

ARM 响应 IRQ 中断请求完成以下工作：

```
R14_irq =  当前指令地址  + 8
SPSR_irq = CPSR
CPSR[4:0] = 0B10010
CPSR[5] = 0 /*ARM 状态*/
CPSR[7] = 1 /*IRQ disabled*/
if high vectors configured then
PC = 0XFFFF0018
else
PC = 0X00000018
```

IRQ 中断处理程序的基本框架：

```
SUB       R14, R14, #4
STMFD    R13!, {R12, R14}
MRS       R12, SPSR
STMFD    R13!, {R12}
MOV       R12, #IntBase
LDR       R12, [R12, #IntLevel]

MRS       R14, CPSR
BIC       R14, R14, #0X80
MSR       CPSR_c, R14

LDR       PC, [PC, R12, LSL #2]

DCD       PRIORITY0HANDLER
DCD       PRIORITY1HANDLER
...

PRIORITY0HANDLER
STMFD    R13!, {R0-R11}
...
...
MRS       R12, CPSR
ORR       R12, R12, #0X80
MSR       CPSR_c, R12

LDMFD    R13!, {R0-R11}
MSR       SPSR_cxsf, R12
LDMFD    R13!, {R12, PC}^

PRIORITY1HANDLER
```

5. ARM 汇编语言程序设计

5.1 伪操作(derective)

5.1.1 符号定义伪操作

① GBLA,GBLL 及 GBLS 声明全局变量
② LCLA,LCLL 及 LCLS 声明局部变量
③ SETA,SETL 及 SETS 给变量赋值
④ RLIST 为通用寄存器列表定义名称
⑤ CN 为协处理器的寄存器定义名称
⑥ CP 为协处理器定义名称
⑦ DN 及 SN 为 VFP 的寄存器定义名称
⑧ FN 为 FPA 的浮点寄存器定义名称

(1) GBLA,GBLL,GBLS。

声明全局变量。

(2) LCLA,LCLL,LCLS。

声明局部变量。

(3) SETA,SETL,SETS。

给 ARM 程序中的变量赋值。

(4) RLIST。

为一个通用寄存器列表定义名称。

格式：name RLIST {List-of-registers}

注：定义的名称可以在 LDM/STM 指令中使用。

(5) CN

为一个协处理器的寄存器定义名称。

格式：name CN expr

注：expr 的值范围为 0~15

(6) CP

为一个协处理器定义名称。

格式：name CP expr

注：expr 的值范围为 0~15

(7) DN/SN

DN 为一个双精度的 VFP 寄存器定义名称。

SN 为一个单精度的 VFP 寄存器定义名称。

格式：name DN expr

注 name SN expr 其中 name 是该 VFP 寄存器的名称。expr 为 VFP 双精度寄存器编号 0~15 或者时单精度寄存器编号 0~31。

(8) FN

为一个 FPA 浮点寄存器定义名称。

格式： name FN expr

注：expr 为浮点寄存器的编号，数值范围为 0~7。

5.1.2 数据定义伪操作

① LTORG 声明一个数据缓冲池(literal pool)的开始。

② MAP 定义一个结构话的内存表(storage map)的首地址。

③ FIELD 第一结构化的内存表中的一个数据域(field)。

④ SPACE 分配一块内存单元，并用 0 初始化。

⑤ DCB 分配一段字节的内存单元，并用指定的数据初始化。

⑥ DCD/DCDU 分配一段字的内存单元，用指定的数据初始化。

⑦ DCDO 分配一段字的内存单元，并将单元的内容初始化成该单元相对于静态基址寄存器的偏移量。

⑧ DCFD/DCFDU 分配一段双字的内存单元，并用双精度的浮点数据初始化。

⑨ DCFS/DCFSU 分配一段字的内存单元，并用单精度的浮点数据初始化。

⑩ DCI 分配一段字节的内存单元，用指定的数据初始化，指定内存单元中存放的时代码，而不时数据。

⑪ DCQ/DCQU 分配一段双字的内存单元，并用 64 位的整数数据初始化。

⑫ DCW/DCWU 分配一段半字的内存单元，并用指定的数据初始化。

⑬ DATA 在代码段中使用数据。现在已不再使用，仅用于向前兼容。

(1) LTORG

- ARM 汇编编译器常把缓冲池放在代码段的最后面，即下一个代码段开始之前，或者 END 伪操作之前。
- 当程序中使用 LDFD 之类的指令时，数据缓冲池的使用可能越界。这是可以使用 LTORG 伪操作定义数据缓冲池，已越界发生。通常大的代码段可以使用多个数据缓冲池。
- LTORG 伪操作通常放在无条件跳转指令之后，或者子程序返回指令之后，这样处理器就不会把数据缓冲池的数据当作时指令来执行。

示例：

```
AREA     Example, CODE, READONLY
start    BL      FUNC1
FUNC1
;CODE
         LDR     R1, =0X55555555
;CODE
         MOV     PC, LR                  <----返回指令之后
         LTORG                           <----文字池
data     SPACE 4200
         END                             <----END 之前
```

(2) MAP

- 定义一个结构化的内存表的首地址。此时内存表的位置计数器{VAR}设置成该地址值。

语法格式：

```
MAP     expr，{，bace-register}
```

示例：

```
MAP     0X80,R9        ;内存表的地址为 R9+0X80
```

(3) FIELD

- 定义一个结构化的内存表中的数据域。#时 FIELD 的同义词。
- MAP 伪操作和 FIELD 伪操作配合使用来定义结构化的内存表结果。MAP 伪操作定义
- 内存表的首地址；FIELD 伪操作定义内存表中个数据域的字节长度，并可以为每
- 一个数据域指定一个标号，其他指令可以引用该标号。
- MAP 伪操作中的 base-register 寄存器的值对于其后所有的 FIELD 伪操作定义的数据域时默认使用的，直至遇到新的包含 base-register 项的伪操作。
- MAP 和 FIELD 伪操作仅仅时定义数据结构，并不真实的分配空间。

语法格式：

```
{lable}   MAP    expr
```

示例：

① 基于绝对地址的内存表；

```
MAP          4096
consta       FIELD   4
constb       FIELD   4
x            FIELD   8
y            FIELD   8
string       FIELD   256

                LDR     R6, consta
```

4096-----> ┌──────┬──────┐
| consta | 4 Bytes |
5000-----> ├──────┼──────┤
| constb | 4 Bytes |
5004-----> ├──────┼──────┤
| x | 8 Bytes |
5012-----> ├──────┼──────┤
| y | 8 Bytes |
5020-----> ├──────┼──────┤
| string | 256 B |
5276-----> └──────┴──────┘

② 相对地址内存表；

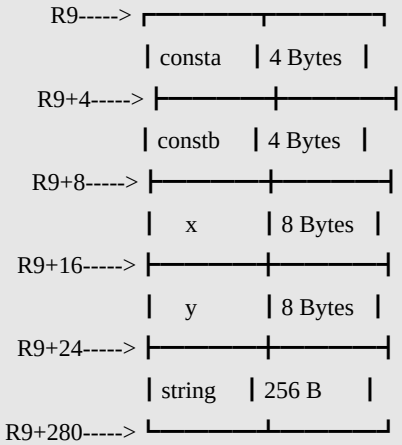
```
MAP 0
consta       FIELD   4
constb       FIELD   4
x            FIELD   8
y            FIELD   8
string       FIELD   256

MOV       R9, #4096
LDR       R5,[R9,consta]
```

0-----> ┌──────┬──────┐
| consta | 4 Bytes |
4-----> ├──────┼──────┤
| constb | 4 Bytes |
8-----> ├──────┼──────┤
| x | 8 Bytes |
16-----> ├──────┼──────┤
| y | 8 Bytes |
24-----> ├──────┼──────┤
| string | 256 B |
280-----> └──────┴──────┘

/***/

③ 相对地址内存表；

```
MAP 0, R9
consta      FIELD   4
constb      FIELD   4
x           FIELD   8
y           FIELD   8
string      FIELD   256
```




```
ADR     R9, DataArea
LDR     R6,consta
```

④ 基于 PC 的内存表；

```
DataStruct    SPACE   280
MAP     DataStruct
consta      FIELD     4
constb      FIELD     4
x           FIELD     8
y           FIELD     8
string      FIELD     256
LDR     R5, constb      ;相当于 LDR     R5, [PC, offset]不超过 4KB 范围。
```

⑤ FIELD 伪操作数为 0。

当 FIELD 伪操作中的操作数为 0 时，其中的标号即为当前内存单元的地址，由于其中操作数为 0，汇编编译器处理该伪操作后，内存表的位置计数器的值并不改变。可以利用这种技术来判定当前内存的使用没有超过程序可分配的可用内存。

下面的伪操作序列定义一个内存表，首地址为 PC 寄存器的值，该内存表中包含 5 个域:consta 4 bytes, constb 4 bytes, x 8 bytes , y 8 bytesstring 长度为 maxlen bytes,为了防止 maxlen 的取值超出内存，越界使用，可以利用 endofstru 监视内存的使用情况，保证其不超过 endofmem。

```
        startofmem      EQU     0x1000  ;内存区域起始地址
        endofmem        EQU     0x2000  ;内存区域结束地址
        MAP     startofmem              ;从起始地址放内存表
        consta  FIELD   4
        constb  FIELD   4
        x       FIELD   8
        x       FIELD   8
        string  FIELD   maxlen          ;最大地址
        endofstru FIELD 0               ;获得最大地址值
```

```
ASSERT    endofstru <= endofmem ;保证其是小于内存区域结束地址
```

(4) SPACE

- 用于分配一块内存单元，用 0 初始化，"%"是 SPACE 的同义词。

语法格式：

```
{lable} SPACE expr
```

(5) DCB

- 用于分配一段字节内存单元，并用指定值初始化。"="是 DCB 的同义词。

语法格式：

```
{lable} DCB expr{,expr}...
```

(6) DCD/DCDU

- 分配一段字内存单元，并用指定值初始化。"&"是 DCD 的同义词；
- DCD 和 DCDU 的主要区别是 DCDU 分配的内存单元并不严格字对齐。DCD 可能在分配的第一个内存单元填补字节以保证分配的内存是字对齐的。

语法格式：

```
{lable} DCD expr{,expr}...
```

(7) DCDO

- 用于分配一段字内存单元(分配的内存都时字对齐的)，并将字单元的内容初始化为 expr 标号基于静态基址寄存器 R9 的偏移量。

语法格式：

```
{lable} DCDO expr{,expr}...
```

示例：

```
IMPORT   externsym
DCDO     externsym     ;字单元，其值为 externsym 基于 R9 的偏移量
```

(8) DCFD/DCFDU

- DCFD 用于为双精度的浮点数分配字对齐的内存单元，并将这个字单元的内容初始化为 fpliteral 表示的双精度浮点数。每个双精度浮点数占据两个字单元。
- DCFD 和 DCFDU 的不同之处在于 DCFDU 分配的内存单元并不严格字对齐。

语法格式：

```
{lable} DCFD(U) fpliteral{,fpliteral}...
```

示例：

```
DCFD     1E308, -4E-100
DCFDU    10000, -.1, 3.1E26
```

(9) DCFS/DCFSU

- 用于单精度的浮点数分配字对齐的内存单元，并将这个字单元的内容初始化为 fpliteral 表示的单精度浮点数。每个单精度的浮点数占据 1 个字节单元。
- DCFS 和 DCFSU 的区别也时在于 DCFSU 不严格字对齐。

语法格式：

```
{lable} DCFS(U) fpliteral{,fpliteral}...
```

(10) DCI

- 用于分配一段字内存单元(内存的分配都是字对齐)，并用指定的值初始化；
- 在 THUMB 代码中 DCI 用于分配一段半字内存单元(半字单元对齐)，并初始化；

- DCI 和 DCD 分配很相识，区别在于 DCI 分配的内存中的数据被标识为指令，可用于通过宏指令来定义处理器指令系统不支持的指令。

语法格式：

```
{lable} DCFS(U) expr{,expr}...
```

示例：

```
MACRO
newinst     $Rd, $Rm
DCI           0xe16f0f10 :OR: ($Rd :SHL: 12) :OR: $Rm ;这里放的是 MEND
```

(11) DCQ/DCQU

- 分配一段以 8 个字节为单位的内存(字对齐)，并用指定值初始化。
- DCQ 和 DCQU 的区别在于 DCQU 不严格的字对齐。DCQ 可能在分配的内存单元前插入多达 3 个填补字节以保证字对齐。

语法格式：

```
{lable} DCQ(U) {-}literal{,{-}literal}...
```

注：literal 为 64 位的数字表达式。其取值范围：0~2^64-1。当在 literal 前面加上"-"时 literal 的取值范围是：-2^64~-1。

(12) DCW/DCWU

- 分配一段半字内存单元(半字对齐)，并用指定值初始化；
- DCW 和 DCWU 的区别在于 DCWU 分配的内存单元不严格半字对齐。

语法格式：

```
{lable} DCW(U) expr{，expr}
```

5.1.3 汇编控制伪操作

① IF,ELSE, ENDIF
② WHILE, WEND
③ MACRO, MEND
④ MEXIT

(1) IF,ELSE,ENDIF

- 根据条件把一段源代码包括到汇编语言程序内或者排除在外。
- "["是 IF 的同义词，"]"是 ENDIF 的同义词，"|"是 ELSE 的同义词。
- IF,ELSE,ENDIF 可以嵌套使用。

语法格式：

```
IF logical expression
instructions or derectives
{ELSE                                    ;可选项
instructions or derectives
}
ENDIF
```

(2) WHILE,WEND

- 能够根据条件汇编一段相同或几乎相同的源代码；
- WHILE,WEND 可以嵌套使用。

语法格式：

WHILE logical expression

instructions or derectives

WEND

示例：

```
count SETA 1
WHILE count <= 4              ;循环控制
count SETA count + 1
;code
WEND
```

(3) MACRO,MEND

- 宏定义，替换展开；
- 用于子程序短，而需要比较多的传递形式参数。提高速度。单代码量增加；
- 如果变量在宏定义中被定义，在其作用范围即对该宏定义体。

语法格式：

```
MACRO
{$lable} macroname {$parameter{,parameter}...}
;code
...
;code
MEND
```

示例：

```
MACRO
$lable     xmac    $p1, $p2
;code...
...
$lable.loop1
;code...
BGE    lable.loop1
;code...
$lable.loop2
BL     $p1
BGT    $label.loop2
;code...
ADR    $p2
;code...
MEND
```

(4) MEXIT

- 从宏中跳出

示例：

```
MACRO
$abc       macroabc    $param1, $param2
```

```
;code
WHILE     condition1
;code
IF condition2
;code
MEXIT                <----直接退出宏
ELSE
;code
ENDIF
WEND
```

5.1.4 栈中数据帧描述伪操作

① ASSERT
② INFO
③ OPT
④ TTL,SUBT

(1) ASSERT

- 汇编器对汇编源程序的第二遍扫描中，如果 ASSERTION 中的条件不成立 ASSERT 伪操作将报告该错误信息。
- 用于保证源程序被汇编时满足相关的条件，如果条件不满足，ASSERT 伪操作报告错误类型，并终止汇编。

语法格式:

```
ASSERT     logical expression
```

示例：

```
ASSERT     a >= 10       ;测试 a >= 10 条件是否满足
```

(2) INFO

- 用于用户自己定义的错误信息。
- 支持在汇编处理过程的第一遍扫描或者第二遍扫描时报告诊断信息。

语法格式：

```
INFO    numeric-expression, string-expression
```

其中：

numeric-expression 为一个数字表达式，如果 numeric-expression 为 0，则在第二遍扫描时，伪操作打印 string-expression;如果 unmeric-ex-pression 的值不为 0，则在汇编处理中，第一遍扫描时，伪操作打印 string-expression,并终止汇编。

示例：

```
INFO       0, "VERSION 1.0"                    ;第二遍扫描时打印"VERSION 1.0"
IF    endofdata <= label1                      ;在第一遍扫描时条件成立
INFO 4, "data overrun at label1"        ;扫描时报告错误信息，终止汇编
ENDIF
```

(3) OPT

- 可以在源程序中设置列表选项。
- OPT 选项伪操作的编码及其意义。

表格 5.1.1 OPT 选项伪操作编码

编码	选项含义
1	设置常规列表选项
2	关闭常规列表选项
4	设置分页符，在新的一页开始显示
8	将行号重新设置为 0
16	设置选项,显示 SET,GBL,LCL 伪操作
32	设置选项，不显示 SET,GBL,LCL 伪操作
64	设置选项，显示宏展开
128	设置选项，不显示宏展开
256	设置选项，显示宏调用
512	设置选项，不显示宏调用
1024	设置选项，显示第一遍扫描列表
2048	设置选项，不显示第一遍扫描列表
4096	设置选项，显示条件汇编伪操作
8912	设置选项，不显示条件汇编伪操作
16384	设置选项，显示 MEND 操作
32768	设置选项，不显示 MEND 操作

- 使用编译选项-list 将使编译器产生列表文件。
- 默认情况下，-list 选项将生成常规的列表文件，包含变量声明、宏展开、条件汇编伪操作已经 MEND 伪操作，而且列表文件只是第二遍扫描时给出。通过 OPT 伪操作，可以在源程序宏改变默认的选项。

示例：

在 FUNC1 前插入 OPT4 伪操作，FUNC1 将在新的一页中显示。

```
AREA EXAMPLE, CODE, READONLY
start  ；code
；code
BL FUNC1
；code
OPT 4    ;Place a page berak before FUNC1
FUNC1   ;code
```

(4) TTL，SUBT

- TTL 伪操作在列表文件的每一页的开头插入一个标题。该 TTL 伪操作将作用在其后的每一页，直到遇到新的 TTL 伪操作。
- TTL 伪操作在列表文件的页顶显示一个标题。如果要在列表文件的第一页显示标题，TTL 伪操作要放在源程序的第一行。
- 当 TTL 伪操作改变页标题时，新的标题在下一页开始起作用。
- SUBT 伪操作在列表文件的每一页的开头插入一个子标题。该 SUBT 伪操作将作用其后的每一页，直到遇到新的 SUBT 伪操作。

- SUBT 伪操作的其他性质和 TTL 类似。

语法格式：

```
TTL title
SUBT subtitle
```

5.1.5 其他的伪操作

① ALIGN
② AREA
③ CODE16,CODE32
④ END
⑤ ENTRY
⑥ EQU
⑦ EXPORT,GLOBAL
⑧ EXTERN
⑨ GET,INCLUDE
⑩ IMPORT
⑪ INCBIN
⑫ KEEP
⑬ NOFP
⑭ REQUIRE
⑮ REQUIRE8,PRESERVE8
⑯ RN
⑰ ROUT

(5) CODE16,CODE32

- 告诉汇编编译器后面的指令序列位 16 位 THUMB 指令和 32 位的 ARM 指令。
- 本身不进行处理器状态的切换。

(6) EQU

- 为数字常量、基于寄存器的值和程序中的标号定义一个字符名称。
- "*"是 EQU 的同义词。

语法格式：

```
name    EQU    expr{,type}
```

其中：

- expr 位基于寄存器的地址值、程序中的标号、32 位的地址常量或者 32 位的常量。
- type 为 expr 为 32 位常量时，可以使用 type 指示 expr 表示的数据的类型。有三种取值：CODE16, CODE32, DATA

(7) AREA

- 定义一个代码段和数据段。

语法格式：

```
AREA    sectionname {,attr}{,attr}...
```

其中：

- sectionname 为所定义的代码段和数据段的名称。如果该名称是以数字开头，则该名称必须用"|"括起来。如：|7wolfs|。还有一些代码段具有约定的名称，如：|.text|表示 C 语言编译器产生的代码段或者是与 C 语言库相关的代码段。
- attr 是该代码段(或者程序段)的属性。在 AREA 操作中，个属性间用逗号隔开。下面列举所有可能的属性：

① ALIGN = expression。默认的情况下，ELF 的代码段和数据段是 4 字节对齐的。expression 可以取 0~31 的数值，相应的对齐方式为(2^expression)字节对齐。如 ALIGN 3 就是以 2^3(即 8)字节对齐。

② ASSOC = section。指定本段相关的 ELF 段。任何时候连接 section 段也必须包含 sectionname 段。

③ CODE 定义代码段。默认的属性为 READONLY。

④ COMDEF 定义一个通用的段。该段可以包含代码或者数据。在整个源文件中，同名的 COMDEF 段必须相同。

⑤ COMMON 定义一个通用的段。该段不包含任何用户代码和数据，连接器将其初始化为 0。各源文件同名的 COMMON 段公用通用的内存单元，连接器为其分配合适的尺寸。

⑥ DATA 定义数据段。默认属性为 READWRITE

⑦ NOINT 指定本数据段仅仅保存了数据单元，而没有将各个初始值写入内存单元。或者将整个内存单元初始化为 0。

⑧ READONLY 指定本段为只读。代码段的默认属性。

⑨ READWRITE 指定本段为可读可写，数据段的默认属性。

注：通常可以使用 AREA 将程序分为多个 ELF 格式的段。段名称可以相同，这时这些同名的段被放在同一个 ELF 段中。一个大的程序可以包含多个代码段和数据段。一个汇编语言程序至少包含一个段。

(8) ENTRY

- 指定程序的入口点。
- 一个程序中至少要有一个(或多个)ENTRY，但时一个源程序中最多只能有一个(或没有)ENTRY。

(9) END

- 告诉编译器已经到达源程序的结尾。
- 每个汇编语言源程序都包含 END 伪操作，以表明源程序的结束。

(10) ALIGN

- 通过添加补丁字节使当前位置满足一定的对齐方式。

语法格式：

ALIGN {expr{，offset}}

其中：

- expr 为数字表达式，用于指定对齐方式。可能的取值为 2 的次幂。如果伪操作没有指定 expr，则当前位置对齐到下一个字边界处。
- offset 为数字表达式。当前位置对齐到：offset + n*expr。

Thumb 的宏指令 ADR 要求地址是字对齐的，而 Thumb 代码中地址标号可能不时字对齐的。这时就要使用伪操作 ALIGN 4 使 Thumb 代码中的地址标号是字对齐的。

- 由于有些 ARM 处理器的 CACHE 采用了其他对齐方式，如 16 字节的对齐方式，这时使用 ALIGN 伪操作指定合适的对齐方式可以充分发挥该 CACHE 的性能优势。

- LDRD 及 STRD 指令要求内存单元时 8 字节对齐的。这样在为 LDRD/STRD 指令分配的内存单元前要使用 ALIGN 8 实现 8 字节对齐方式。
- 地址标号通常自身没有对齐要求。而在 ARM 代码中要求地址标号时字对齐的。在 Thumb 代码中要求字节对齐。这样需要使用合适的 ALIGN 伪操作来调整对齐方式。

示例：

① <1>.在 AREA 伪操作中的 ALIGN 与 ALIGN 伪操作中的 expr 含义是不同的。

```
AREA cacheable， CODE， ALIGN = 3 <-----2^3 = 8
subrout1 ;code
;code
MOV PC, LR
ALIGN 8 <----------指定以下的指令为 8 字节对齐。
subrout2 ;code
```

② 将两个字节数据放在同一个字的第一字节和第四字节中。

```
AREA OffsetExample， CODE
DCB 1
ALIGN 4,3
DCB 1
```

③ 通过 ALIGN 伪操作使程序中地址标号字对齐。

```
AREA Example, CODE, READONLY
start LDR r6, =label1
;code
MOV PC, LR
label1 DCB 1 <----------本伪操作破坏了字对齐
ALIGN <----------重新使数据对齐
subroutine1
MOV R5, #0X05
```

(11) EXPORT,GLOBAL

- 声明一个符号可以被其他文件引用。GLOBAL 使 EXPORT 的同义词。

语法格式：

```
EXPORT symbol{WEAK}
```

其中：

- symbol 为声明的符号的名称。它使区分大小写的。
- [WEAK] 选项声明其他的同名符号优先于本符号被引用。

(12) IMPORT

- 告诉编译器当前的符号不是在本源文件中定义的，而是在其他文件源文件中定义的，在本源文件中可以引用该符号。
- 不论本源文件是否实际引用该符号，该符号都将被加入到本源文件的符号表中。
- 如果 IMPORT 伪操作声明一个符号时在其他源文件中定义的，如果连接处理不能解析该符号，且也没有指定 WEAK，则就报错。如果连接处理时不能解析该符号，但是指定了 WEAK，连接器将不会报告错误，而是进行下面的操作：

① 如果该符号被 B,BL 指令引用，则该符号被设置成下一条指令的地址，该 B，BL 指令相当于一条 NOP 指令。

② 其他情况下该符号被设置为 0。

语法格式：

```
IMPORT      symbol{WEAK}
```

其中：

- symbol 为声明的符号的名称。它使区分大小写的。
- [WEAK] 指定这个选项后，如果 symbol 在所有的源文件都没有被定义，编译器也不会产生任何错误信息，同时编译器也不会到当前没有被 INCLUDE 进来的库中去查找该符号。

(13) EXTERN

- 告诉编译器当前的符号不时在本源文件中定义的，而是在其他源文件定义的，在本源文件中可能引用该符号。如果本源文件没有实际引用该符号，该符号都将不会加入到本源文件的符号表中。

语法格式：

- 如果 IMPORT 伪操作声明一个符号时在其他源文件中定义的，如果连接处理不能解析该符号，且也没有指定 WEAK，则就报错。如果连接处理时不能解析 该符号，但是指定了 WEAK，连接器将不会报告错误，而是进行下面的操作：

① 如果该符号被 B,BL 指令引用，则该符号被设置成下一条指令的地址，该 B，BL 指令相当于一条 NOP 指令。

② 其他情况下该符号被设置为 0。

语法格式：

```
EXTERN      symbol{WEAK}
```

其中：

- symbol 为声明的符号的名称。它使区分大小写的。
- [WEAK] 指定这个选项后，如果 symbol 在所有的源文件都没有被定义，编译器也不会产生任何错误信息，同时编译器也不会到当前没有被 INCLUDE 进来的库中去查找该符号。

(14) GET,INCLUDE

- 将一个源文件包含到当前源文件中，并将被包含的文件在其当前位置进行处理。INCLUDE 时 GET 的同义词。
- 通常可以在一个源文件中定义宏、EQU 定义常量名称、MAP/FIELD 定义结构化的数据类型，这样一的源文件类似于 C 语言中的.H 文件。然后用 GET/INCLUDE 伪操作将其包含到它们的源文件中。
- 编译器通常在当前目录中查找被包含的源文件。可以使用-I 条件其他的查找目录。同时被包含的源文件中也可以使用 GET 伪操作，即 GET 可以嵌套使用。
- GET 伪操作不能用来包含目标文件。包含目标文件需要使用 INCBIN 伪操作。

语法格式：

```
GET     filename      ;filename 可以是路径信息
```

示例：

```
AREA    Example, CODE, READONLY
GET     file1.s
GET     c:\project\file2.s
```

(15) INCBIN

- 将一个文件包含到当前源文件中，被包含的文件不进行汇编处理。
- 通常可以用 INCBIN 将一个执行文件或者任意的数据包含到当前文件中。被包含的执行文件或数据将被原封不动的放到当前文件中。编译器从 INCBIN 伪操作后面开始进行处理。
- 编译器通常在当前目录中查找被包含的源文件。可以使用-I 条件其他的查找目录。同时被包含的源文件中也可以使用 INCBIN 伪操作，即 INCBIN 可以嵌套使用。
- 这里所包含的文件名称及其路径信息中不能有空格。

语法格式：

```
INCBIN filename     ;filename 可以是路径信息
```

(16) KEEP

- 告诉编译器将局部符号包含在目标文件的符号列表中。
- 默认的情况下编译器仅仅将以下的符号包含到目标文件的符号表中：

① 被输出的符号。

② 将会被重定位的符号。

- KEEP 伪操作可以将局部符号也包含到目标的符号表中，从而使得调试工作更加方便。

语法格式：

```
KEEP  {symbol}
```

其中：

- symbol 为包含在目标文件的符号列表中的符号。如果没有指定 symbol
- 则除了基于寄存器的所有符号将被包含在目标文件的符号中。

示例：

```
label    ADC    R2, R3, R4
KEEP label       ;将标号 label 包含到目标文件的符号列表中。
```

(17) NOFP

- 当系统中没有硬件或软件仿真代码直至浮点运算指令时，使用 NOFP 伪操作禁止源程序中包含浮点运算指令。这时如果源程序中包含浮点运算指令，编译器同样将会报告错误。

(18) REQUIRE

- 指定段之间的相互依赖关系。
- 当进行连接处理包含了有 REQUIRE label 伪操作的源文件，则定义 label 的源文件也将被包含。

语法格式：

```
REQUIRE   label
```

(19) REQUIRE8/PRESERVE8

- REQUIRE8 伪操作指示当前代码中要求数据栈 8 字节对齐。
- PRESERVE8 伪操作指示当前代码中数据栈 8 字节对齐。
- LDRD/STRD 指令要求内存单元地址是 8 字节对齐的。当在程序中使用这些指令在数据栈中传送数据时，要求数据栈是 8 字节对齐的。
- 连接器要保证要求 8 字节对齐的数据栈代码只能被数据栈 8 字节的代码调用。

(20) RN

- 为一个特定的寄存器定义名称。

语法格式：

```
name    RN   expr
```

(21) ROUT

- 定义局部变量的作用范围。
- 当没有 ROUT 伪操作指令定义局部变量的范围时，局部变量的作用范围为其所在的段(AREA)。ROUT 伪操作作用范围为本 ROUT 伪操作到下一个 ROUT(同一个段中)伪操作之间。

5.2 ARM 汇编语言伪指令

① ADR 小范围的地址读取伪指令

② ADRL 中等范围的地址读取伪指令

③ LDR 大范围的地址读取伪指令

④ NOP 空操作伪指令

(1) ADR

- 将基于 PC 的地址值或基于寄存器的地址值读取到寄存器中。
- 在汇编编译器处理源程序时,ADR 将被编译器替换成一条合适的指令。通常，编译器用一条 ADD 指令和 SUB 指令来实现该 ADR 伪指令的功能。如果不能用一条指令实现 ADR 伪指令的功能，编译器将报告错误。
- 因为 ADR 伪指令中的地址是基于 PC 或者基于寄存器的，所以 ADR 读取到的地址为位置无关的地址。当 ADR 伪指令中的地址时基于 PC 时，该地址与 ADR 伪指令必须在同一个代码段中。

语法格式：

```
ADR{cond} register, expr
```

其中：

expr 为基于 PC 或者基于寄存器的地址表达式，取值范围如下：

- 当地址值不式字对齐，其范围为：-255 - +255；
- 当地址值是字对齐时，其范围为：-1020 - +1020；
- 当地址值是 16 字对齐时，其范围将更大。

(2) ADRL

- 该指令将基于 PC 或基于寄存器的地址读取到寄存器中。ADRL 比 ADR 指令可以读取到更大范围的地址。ADRL 伪指令在汇编时被编译器替换成两条指令。
- 编译器将 ADRL 指令替换成两条合适的指令，即使一条指令可以完成，也将要两条指令替换。如果不能用两天指令来实现 ADRL 伪指令的功能，编译器报错。

语法格式：

```
ADRL{cond} register, expr
```

其中：

expr 为基于 PC 或者基于寄存器的地址表达式，取值范围如下：

- 当地址值不式字对齐，其范围为：-64KB - +64KB；
- 当地址值是字对齐时，其范围为：-256KB - +256KB；
- 当地址值是 16 字对齐时，其范围将更大。

示例：

```
start    MOV    R0, #10
ADRL     R4, start+60000                    ;编译器替换成下面两条指令。
ADD      R4, PC, #0XE800
ADD      R4, R4, #0X254
```

(3) LDR

- 将一个 32 位的常数或者一个地址值读取到寄存器中。

语法格式：

```
LDR{cond} register, =[expr | label-expr]
```

其中：

expr 为 32 位的常量。编译器将根据 expr 的取值情况，如下处理 LDR 伪指令。

- 如果 expr 表示的值没有超过 MOV,MVN 指令中的地址范围，编译器就用合适的 MOV 和 MVN 指令代替该 LDR 伪指令。
- 当 expr 表示的地址值超过了 MOV 或 MVN 指令中地址的范围，编译器将该常数放在数据缓冲区中，同时用一条基于 PC 的 LDR 指令读取该常数。
- label-expr 为基于 PC 的地址表达式或者时外部表达式。当 label-expr 为基于 PC 的地址表达式时，编译器将 label-expr 表示的数字放入数据缓冲池中，同时用一条基于 PC 的 LDR 指令读取该数值。当 label-expr 为外部表达式，或者非当前段的表达式时，汇编编译器将在目标文件中插入连接重定位伪操作，这样连接器将在连接时生成该地址。

示例：

```
LDR   R1, =0XFF0                    ;汇编后得到   MOV    R1,0XFF0
LDR   R1, =0XFFF                    ;汇编后得到   LDR    R1, [PC, OFFSET_TO_LPOOL]
LPOOL DCD 0XFFF
LDR   R1, =ADDR1                    ;汇编后得到   LDR    R1, [PC, OFFSET_TO_LPOOL]
LPOOL DCD ADDR1
```

(4) NOP

- 在汇编时被替换成 ARM 的空操作，如 MOV R0, R0 等。
- 不影响 CPSR 的条件标志。

5.3 ARM 汇编语言语句格式

格式如下：

```
{symbol} {instruction | directive | pseudo-instruction} {;comment}
```

- symbol 顶格写不能包含空格；
- instruction 不能顶格写；
- 分号作为注释符一直到该行结束；
- 太长的语句可以用"\"分成几行来写。

5.3.1 ARM 汇编语言中的符号

1. 符号的命名规则：

① 符号由大小写字母、数字、下划线组成。

② 局部标号可以用数字开头，其他的标号不能。

③ 符号区分大小写。

④ 符号中的所有字符都时由意义的。

⑤ 符号在其作用范围内必须时惟一的。

⑥ 程序中的符号不能于系统内部变量或者系统预定义的符号同名。

⑦ 程序中的符号不要与指令助记符或者伪操作同名。当程序中的符号和指令助记符或者伪操作同名时，用双竖线将符号括起来，如：||require||，这时双竖线并不是符号的组成部分。

(1) 变量

- 数字变量、逻辑变量、串变量。

(2) 汇编的变量替换

- $字符后面的字符将被原样替换。包括其本身$$。
- 对于数字变量来说，如果该变量前面有$字符，在汇编时编译将该数字变量的数字转换成十六进制的串，然后用该十六进制串取代$字符后的变量。
- 对于逻辑变量来说，如果该逻辑变量前面有一个$字符，在汇编时编译器将该逻辑变量替换成它的取值(T 或者 F)。

示例：

```
GBLS    STR1
GBLS    B
GBLA    NUM1
NUM1 SETA    14
B       SETB    "CHANGED"
STR1   SETS    "ABC$$B$NUM1"      ;汇编后得到： ABCB
```

- 通常情况下，包含在两"|"之间的"$"并不表示变量替换。如果双竖线在双引号内，则将进行变量替换。
- 使用"."来表示变量名称的结束。

示例：

```
GBLS    STR1
GBLS    STR2
STR1     SETS    "AAA"
STR2    SETS    "bbb$STR1.CCC"      ;汇编后得到： bbbAAACCC
```

(3) 标号

- 基于 PC 的标号；
- 基于寄存器的标号；
- 绝对地址标号。

(4) 局部标号

- 主要有两部分组成：开头时一个 0~99 之间的数字。后面紧跟一个通常表示局部变量作用范围的符号。
- 局部标号的作用范围通常为当前段，也可用伪操作 ROUT 来定义局部变量的作用范围。

语法格式：

```
N{routname}
```

- N 为 0~99 之间的数字。
- routname 为符号，通常为该变量作用范围的名称(用 ROUT 伪操作定义的)。

局部变量引用的语法格式如下：

```
%{F|B}{A|T} N{routname}
```

其中：

routname 为当前作用范围的名称(用 ROUT 伪操作定义的)。

- %表示引用操作；
- F 指示编译器只向前搜索；
- B 指示编译器只向后搜索；
- A 指示编译器搜索宏的所有嵌套层次；
- T 指示编译器搜索宏的当前层次。

① 如果 F 和 B 都没有指定，编译器先向前搜索，再向后搜索。

② 如果 F 和 T 都没有指定，编译器搜索所有从当前层次到宏的最高层次，比当前层次低的层次不再搜索。

③ 如果指定了 routname，编译器向前搜索最近的 ROUT 伪操作，若 routname 与该 ROUT 伪操作定义的名称不匹配，编译器报错，汇编失败。

2. ARM 汇编语言中的表达式

④ 括号内的表达式优先级最高；

⑤ 各种操作符有一定的优先级；

⑥ 相邻的单目操作符的执行顺序为由右到左，单目运算符优先级高于其他操作；

⑦ 优先级相同的双目运算符执行顺序为由左到右。

(1) 字符串表达式

- 字符串：由包含在双引号内的一系列字符组成。字符串的长度受到 ARM 汇编语言语句长度的限制。"$$"表示"$",""表示"。

示例：

```
STR1 SETS "this string contains only one"" double quote"
STR2 SETS "this string contains only one $$ dollar symbol"
```

- 字符串变量：字符串变量由 GBLS 和 LCLS 声明，用 SETS 赋值。取值范围与字符表达式相同。
- 操作符：

① LEN（返回字符串的长度）

语法格式：

```
:LEN: A
```

② CHR

可以将 0~255 之间的整数作为含一个 ASCII 字符的字符串。当有些 ASCII 字符不方便放在字符串中时，可以使用 CHR 将其放在字符串表达式中。

语法格式：

```
:CHR: A      <----A 为某一字符的 ASCII 值
```

③ STR

将一个数字量或逻辑表达式转换成串。对于 32 位的数字量而言，STR 将其转换成 8 个十六进制数组成的串。对于逻辑表达式而言，STR 将其转换成字符串 T 或者 F。

语法格式：

```
:STR: A    <----A 为数字量或逻辑表达式
```

④ LEFT

返回一个字符串最左端一定长度的字符串。

语法格式：

```
A :LEFT: B    <----A 为字符串，B 为返回长度
```

⑤ RIGHT

返回一个字符串最右端一定长度的字符串。

```
A :RIGHT: B   <----A 为字符串，B 为返回长度
```

⑥ CC

用于连接两个字符串，B 串接到 A 串后面。

```
A :CC: B   <----A 为第一源字符串，B 为第二源字符串
```

⑦ 字符变量的声明和赋值

- 声明用 GBLS 或者 LCLS。
- 赋值用 SETS。

示例：

```
GBLS   string1
GBLS   string2
string1   SETS "aaaccc"
string2   SETS "BB" :CC: (string2 :LEFT: 3); string2 为：BBAAA
```

(2) 数字表达式

- 数字表达式通常由数字常量、数字变量、操作符和括号组成。
- 数字表达式表示一个 32 位的整数。当作为无符号数时，其取值范围为：0~2^32-1, 当作为有符号整数时，其取值范围为：-2^31~2^31-1
- 汇编编译器并不区分有符号还时无符号数，事实上-n 与 2^32-n 其实在内存中都时一样的。
- 进行大小比较时，数字表达式表示的都是无符号数。按照这种规则:0<-1。

① 整数数字量

- decimal-digits
- Ox hexadecimal-digits
- hexadecimal-digits
- n_basse_n-digits

示例：

```
a          SETA 34906
addr       DCD   0xA10E
LDR        R4, &1000000F
DCD        2_11001010          ;0xCA
C3         DCD   8_74007
DCQ        0x0123456789abcdef
```

② 浮点数字量

- {-}digits E{-}digits
- {-}digits.digits E{-}digits
- 0x hexdigits
- hexdigits

单精度浮点数表示的范围：3.40282347e+38 - 1.17549435e-38

双精度浮点数表示的范围：1.79769313486231571e+308 --- 2.22507385850720138e-308

示例：

```
DCFD          1E308,-4E-100
DCFS          1.0
DCFD          3.725E15
LDFS          0x7FC00000
LDFD          &FFF0 0000 0000 0000
```

③ 数字变量

用 GBLA/LCLA 声明，用 SETA 赋值。

④ 操作符

- NOT 按位取反

语法格式：

```
：NOT: A      <----A 为 32 位数字量
```

- *+、-、*、/、MOD 加减乘除取余

语法格式：

```
A+B、A-B、A*B、A/B、A :MOD: B
```

- ROL/ROR/SHL/SHR 位移

语法格式：

```
A :ROL: B        将整数 A 循环左移 B 位
A :ROR: B        将整数 A 循环右移 B 位
A :SHL: B        将整数 A 左移 B 位
```

```
A :SHR: B        将整数 A 右移 B 位
```

- AND/OR/EOR　按位逻辑操作

```
A :AND: B       A 与 B 按位逻辑与操作
A :OR: B        A 与 B 按位逻辑或操作
A :EOR: B       A 与 B 按位逻辑异或操作
```

(3) 基于寄存器和 PC 的表达式

① BASE

返回基于寄存器的表达式中的寄存器编号。

语法格式：

```
:BASE: A
```

② INDEX

返回基于寄存器的表达式相对于其基址寄存器的偏移量。

语法格式：

```
:INDEX: A
```

③ +/-

正负号。它们可以放在数字表达式和基于 PC 的表达式前面。

(4) 逻辑表达式

① 关系操作符

- 数字表达式
- 字符串表达式
- 基于寄存器表达式
- 基于 PC 的表达式

```
A = B     等于
A > B     大于
A < B     小于
A <= B    小于等于
A >= B    大于等于
A /= B    不等于
A <> B    不等于
```

② 逻辑操作符

```
:LNOT: A       逻辑表达式 A 的值取反
A :LAND: B     逻辑表达式 A 和 B 的逻辑与
A :LOR: B      逻辑表达式 A 和 B 的逻辑或
A :EOR: B      逻辑表达式 A 和 B 的逻辑异或
```

(5) 其他的一些操作符

① ?

返回定义符号 A 的代码行所生成的可执行代码的字节数。

语法格式：

```
? A
```

② DEF

判断某个符号是否已经定义。

语法格式：

```
:DEF: A
```

③ SB_OFFSET_19_12

返回(label-SB)的 bit[19:12]

语法格式：

```
:SB_OFFSET_19_12:  label
```

④ SB_OFFSET_11_0

返回(label-SB)的 bit[11:0]

语法格式：

```
:SB_OFFSET_11_0:  label
```

5.4 ARM 汇编语言程序格式

ARM 汇编语言以段(section)为单位组织源程序。段时相对独立、具有特定名称的、不可分割的指令或者数据序列。段有可以分为代码段和数据段，代码段存放执行代码，数据段存放代码运行时需要用到的数据。一个 ARM 源程序至少需要一个代码段，大的程序可以包含多个代码段和数据段。

ARM 汇编语言源程序经过汇编处理后生成一个可执行的映像文件(类似 WINDOWS 系统下的 EXE 文件)。该可执行文件的映像文件通常包括下面 3 部分：

- 一个或多个代码段。代码段通常是只读的。
- 零个或多个包含初始值的数据段。这些数据段通常时可读写的。
- 零个或多个不包含初始值的数据段。这些数据段被初始化为 0，通常是可读写的。

连接器根据一定的规则将各个段安排到内存中的相应位置。源程序中段之间的相邻关系与执行映像文件中段之间的相邻关系并不一定相同。

示例：

```
AREA EXAMPLE, CODE, READONLY
ENTRY
START    MOV R0, #10
         MOV R1, #3
         ADD R0, R0, R1

END
```

5.5 ARM 汇编编译器的使用

ARMASM 的各参数。

(1) -16 告诉编译器所处理的源程序时 THUMB 指令的程序。其功能和在源程序使用 CODE16 伪操作相同；

(2) -32 告诉编译器所处理的源程序时 ARM 指令的程序。其功能和在源程序中使用 CODE32 伪操作相同；

(3) -apcs [none | [qualifer[/qualifer[...]]]] 用于指定源程序所使用的 ATPCS。使用 ATPCS 并不影响 ARMASM 所产生的目标文件。ARMASM 只是根据 ATPCS 选项在其产生的目标文件中设置相应的属性，连接器将会根据这些属性检查程序中调用关系等是否合适，并连接到合适类型的库文件。ATPCS 选项可能的取值有：

- /none 不使用任何 ATPCS
- /interwork ARM 和 THUMB 指令混合使用
- /nointerwork 指定源程序中没有 ARM、THUMB 指令混合。默认。
- /ropi 指定源程序是 ROPI(只读位置无关)。默认。
- /rwpi 指定源程序时读写位置无关。/norwpi
- /pic /ropi 同义词
- /pid 同/rwpi
- /swstackcheck 指定源程序进行软件数据栈检查。/noswstackcheck
- /swstna 指定源程序既与进行软件数据栈检查的源程序 兼容也与不进行软件数据栈检查的源程序兼容。

(4) -bigend 告诉源程序汇编成适合于 BIG ENDIAN 的模式；

(5) -littleend 告诉源程序汇编成合适于 LITTLE ENDIAN 的模式。 默认；

(6) -checkreglist 告诉 ARMASM 检查 RLIST,LDM,STM 中的寄存器列表。保证寄存器列表中的寄存器时按照寄存器编号由小到大的顺序排列的，否则将产生警告信息。

(7) -cpu 告诉 ARMASM 目标 CPU 的类型。

(8) -depend 告诉 ARMASM 将源程序的依赖列表保存到文件 dependfile。

(9) -m 告诉 ARMASM 将源程序的依赖列表输出到标准输出。

(10) -md 告诉 ARMASM 将源程序的依赖列表输出到文件 inputfile.d。

(11) -errors errorfile 告诉 ARMASM 将错误信息输出到文件 errorfile 中。

(12) -fpu name 指定目标系统中的浮点运算单元的体系。可能的取值：

- none 指定没有浮点选项
- vfpv1 符合 VFPV1 的硬件向量浮点运算单元
- vfpv2 符合 VFPV2 的硬件向量浮点运算单元
- fpa 指定系统使用硬件的浮点加速器(float point accelerator)
- softvfp+vfp 指定系统中使用硬件向量浮点运算单元
- softvfp 指定系统使用软件的浮点运算库，这时使用单元的内存模式
- softfpa 指定系统使用软件的浮点运算库，这时使用混合的内存模式

(13) -g 指示 ARMASM 产生 DRAWF2 格式的调试信息表。

(14) -help 显示本汇编编译器的选项。

(15) -i dir[,dir]...添加搜索路径。指定 GET/INCLUDE 中变量的搜索路径。

(16) -keep 将局部符号保留在目标文件的符号表中，共调试器进行调试时使用。

(17) -list [listingfile][option] 指示 ARMASM 将产生的汇编程序列表保存到列表定义列表 listingfile 中。如果没有指定 listingfile，则保存到 inputfile.lst 文件中。下面是 option 控制选项：

- -noterse 源程序中由于条件汇编被排除的代码也将包含在列表文件中；
- -width 指定列表文件中每行的宽度；
- -length 指定列表文件中每页的行数；
- .-xref 指示 ARMASM 列出各符号的定义和引用情况。

(18) -maxcache n 指定最大的源程序 cache(源程序的 cache 时指 ARMASM 在第一遍扫描时将源程序缓存到内存中，在第二遍扫描时，从内存中读取该源文件)大小。默

认为 8 MB。

(19) -memaccess attributes 指定目标系统的存储访问模式。默认的情况下时允许字节对齐、半字对齐、字对齐的读写访问。可以指定下面的访问属性。

- +L41 允许非对齐的 LDR 访问；
- -L22 禁止半字的 LOAD 访问；
- -S22 禁止半字的 STORE 访问；
- -L22-S22 禁止半字的 LOAD 访问和 STORE 访问。

(20) -nocache 禁止源程序 cache。通常情况下 ARMASM 在第一遍扫描时把源程序保存到内存中，第二遍直接从内存中读取。

(21) -noesc 指示 ARMASM 忽略 C 语言风格的退出类的特殊字符。

(22) -noregs 指示 ARMASM 不要预定义寄存器名称。

(23) -nowarn 不产生警告信息。

(24) -o filename 指定输出的目标文件名称。

(25) -predefine “directive” 指示 ARMASM 预先执行某个 SET 伪操作，可能的 SET 伪操作包括 SETA,SETL,SETS。

(26) -split_ldm 使用该选项时，如果指令 LDM/STM 中的寄存器个数超标，ARMASM 则认为该指令错误。

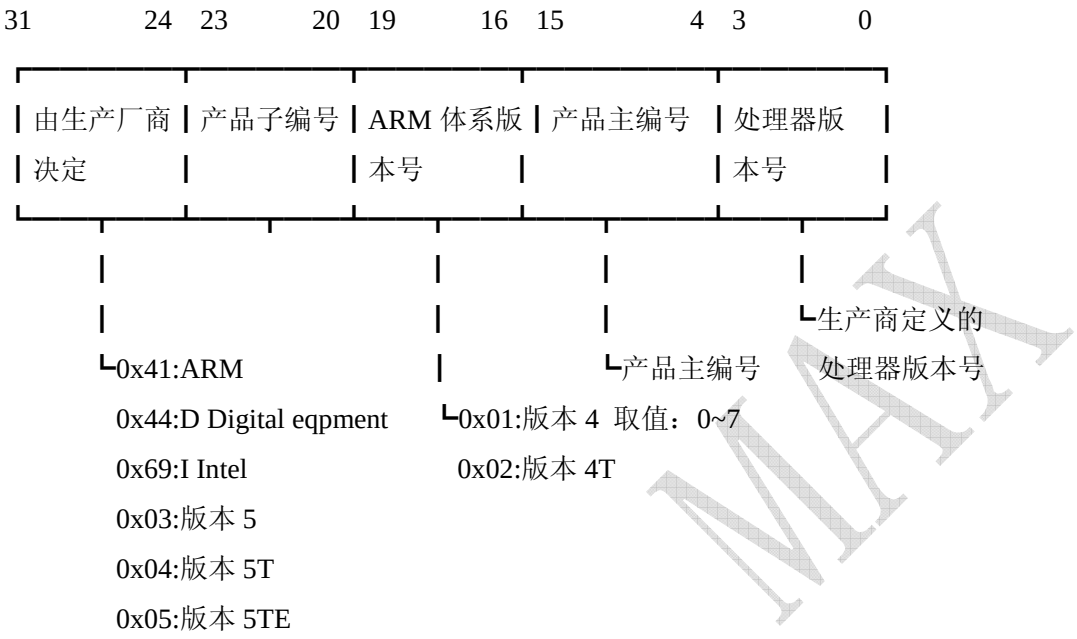
(27) -unsafe 允许源程序中包含目标 ARM 体系或者处理器不支持的指令，这时 ARMASM 对于该类错误报告警告信息。

(28) -via file 指示 ARMASM 从文件 file 中读取各选项信息。

(29) inputfile 为输入的源程序，必须为 ARM 汇编程序或者 THUMB 汇编程序。

6. ARM 存储系统概述

6.1 ARM7 之后的处理器主标识符编码格式



6.2 ARM7 之前处理器主标识符

6.3 MMU 存储器管理单元

在 ARM 系统中,MMU 主要完成以下工作：

- 虚拟存储空间到物理存储空间的映射。在 ARM 中采用了页式虚拟存储管理。它把虚拟地址空间分成一个个固定大小的块，每一块称为一页，把物理存储的内存地址空间也分成同样大小的页。页的大小可以分为粗粒度和细粒度两种。MMU 就要实现虚拟地址到物理地址的转换；
- 存储器访问权限的控制；
- 设置虚拟存储空间的缓冲的特性。

页表:它是一个位于内存中的表。表的每一行对于于虚拟存储空间的一个页，该行包含了该虚拟内存页(虚页)对应的物理内存页(实页)的地址、该页的访问权限和该页的缓冲特性等。将页表里这样一行称为一个地址变换条目。页表存放在内存中，通常有一个寄存器保存页表的基地址。

快表:一段时间内访问的页表有一定的局限性，再而从内存中频繁读取这个表是 很慢的，所以建立一个存储一定时间段内频繁访问的小容量的页表，这个表叫快表。TLB(translation lookaside buffer)。

==========

由于 MMU 关系到具体的处理器型号，稍后补齐！

==========

7. ATPCS 介绍

7.1 ATPCS 概述：

ATPCS 规定了一些子程序之间调用的基本规则。包含：子程序调用过程中寄存器的使用规则，数据栈的使用规则，参数的传递规则。为适应一些基本的调用规则进行一些修改得到不同的子程序调用规则。如下：

- 支持数据栈限制检查的 ATPCS
- 支持只读段位置无关的 ATPCS
- 支持可读写段位置无关的 ATPCS
- 支持 ARM 程序和 THUMB 程序混合使用的 ATPCS
- 处理浮点运算的 ATPCS

7.2 基本 ATPCS

基本 ATPCS 包含以下 3 方面内容：

- 各寄存器的使用规则及其相应的名称；
- 数据栈的使用规则；
- 参数传递的规则。

相对于其他类型的 ATPCS，满足基本 ATPCS 的程序的执行速度更快，所占用的内存更少，但是它不能提供以下的支持：

- ARM 程序和 THUMB 程序相互调用；
- 数据以及代码的位置无关的支持；
- 子程序的可重入性；
- 数据栈检查的支持。

7.2.1 寄存器使用规则

(1) 子程序通过 R0 - R3 来传递参数。此时 R0 - R3 可以记作 A0 - A3。被调用的子程序在返回前无需恢复寄存器 R0 - R3 的内存。

(2) 在子程序中使用 R4 - R11 来保存局部变量。这时 R4 - R11 可以记作 V1 - V8。当然对这些寄存器可能要进行保护。在 THUMB 程序中，通常只能使用 R4 - R7 来保存局部变量。

(3) 寄存器 R12 用作子程序 scratch 寄存器，记作 IP。在子程序间的连接代码段中常有这种使用规则。

(4) 寄存器 R13 用作数据栈指针，记作 SP。在子程序中寄存器 R13 不能用作其他用途。寄存器 R13 在进入子程序时的值和退出子程序时的值必须要相等。

(5) 寄存器 R14 称为连接寄存器，记作 LR。它用于保存子程序的返回地址。如果在子程序中保存了返回地址，寄存器 R14 则可用于其他用途。

(6) 寄存器 R15 时程序计数器，记作 PC。不能用作其他用途。

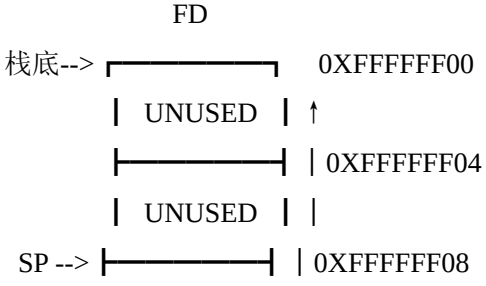
表格 7.2.1 ARM 部分寄存器用途

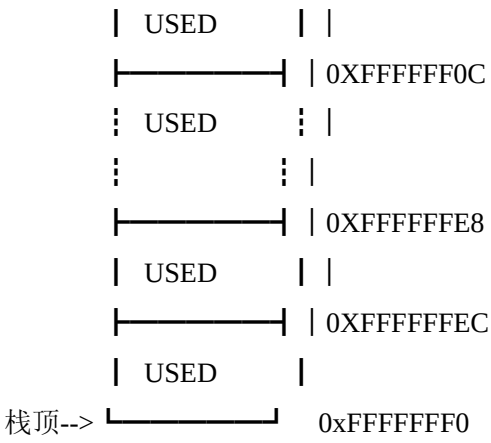
寄存器	别名	特殊名称	使用规则
R15		PC	程序计数器
R14		LR	连接寄存器
R13		SP	数据栈指针
R12		IP	子程序内部调用 scratch 寄存器
R11	V8		ARM 状态局部变量寄存器 8
R10	V7	SL	ARM 状态局部变量寄存器 7 支持数据栈检查的 ATPCS 中为数据栈限制指针
R9	V6	SB	ARM 状态局部变量寄存器 6 在支持数据栈检查的 ATPCS 中为静态基址寄存器
R8	V5		ARM 状态局部变量寄存器 5
R7	V4	WR	ARM 状态局部变量寄存器 4 THUMB 状态工作寄存器
R6	V3		ARM 状态局部变量寄存器 3
R5	V2		ARM 状态局部变量寄存器 2
R4	V1		ARM 状态局部变量寄存器 1
R3	A3		参数、结果、scratch 寄存器 4
R2	A2		参数、结果、scratch 寄存器 3
R1	A1		参数、结果、scratch 寄存器 2
R0	A0		参数、结果、scratch 寄存器 1

7.2.2 数据栈使用规则

(1) 栈顶指针可以指向不同的位置。

- 指向的栈顶不为空，称为 FULL 栈；
- 指向的为空数据单元，称为 EMPTY 栈；
- 向内存地址减小的方向增长时，称为 DESCENDING；
- 向内存地址增加的方向增长时，称为 ASCENDING。
- ATPCS 规定数据栈为 FD 模式。并且对数据栈的操作是 8 B 对齐的。下面是一个数据栈的示例及其相关名词。
- SP(stack pointer)最后一个写入栈的数据内存地址；
- SB(stack base)数据栈的基地址，FD 模式下指的时数据栈的最高地址；
- SL(stack limit)数据栈的界限，数据栈可以使用的最低内存单元地址；
- US(used stack)数据栈基址和栈指针之间的区域；
- UUS(unused stack)数据栈界限到栈指针之间的区域。暂未使用部分；
- SF(stack frames)指在数据栈中，为子程序分配的用来保存寄存器和局部变量的区域。





(2) 在 ARM V5TE 版本中,批量传送指令 LDRD/STRD 要求数据栈是 8 B 对齐的,以提高数据传送速度.用 ADS 编译器产生的目标文件中,外部的数据栈都时 8 B 对齐的,并且编译器将告诉连接器: 本目标文件中的数据时 8 B 字节对齐的.而对于汇编程序来说,如果目标文件包含了外部调用,则必须满足下列条件:

- 外部接口的数据栈必须是 8 B 对齐的.也就是要保证进行该汇编代码后,直到该汇编代码调用外部程序之间,数据栈的栈指针变化偶数个字(如栈指针加 2 个字,而不能加 3 个字)。
- 在汇编程序中使用 PRESERVE8 告诉连接器，本汇编程序时 8 字节对齐的。

7.2.3 参数传递规则

根据参数个数是否固定可以将子程序分为：

- 参数个数固定子程序；
- 参数个数可变子程序。

(1) 参数个数可变的子程序参数传递规则。

- 当参数不超过 4 个时，可以使用寄存器 R0 - R3 来传递参数，当参数超过 4 个时，还可以使用数据栈来传递参数；
- 在传递参数时，将所有参数看作时存放在连续的内存单元中的字数据。然后，依次将各字数据传送到寄存器 R0,R1,R2,R3 中，如果参数多于 4 个，将剩余的字数据传送到数据栈中，入栈的顺序与参数顺序相反，即最后一个字数据先入栈；
- 按照上面的规则，一个浮点数参数可以通过寄存器传递，也可以通过数据栈传递，也可能一半通过寄存器传递，另一半通过数据栈传递。

(2) 参数个数固定的子程序参数传递规则。

- 第一个整参数通过 R0 - R3 来传递。其他通过数据栈传递。
- 如果系统包含浮点运算的硬件部分，浮点参数将按照下面的规则传递：

① 各个浮点数按顺序处理；

② 为每个浮点数分配 FP 寄存器；

③ 分配的方法是:满足该浮点参数需要的且编号最小的一组连续的 FP 寄存器。

(3) 子程序结果返回规则。

- 结果为 32 位的整数可以通过 R0 返回。
- 结果为 64 位整数时，通过 R1R0 返回，依次类推。
- 结果为一个浮点数时，可以通过浮点运算部件的寄存器 f0，d0 或者 s0 来返回。
- 结果为复合型的浮点数(如复数)时，可以通过寄存器 f0 - fn 或者 d0 - dn 返回。
- 对于位数更多的结果需要通过内存来传递。

7.3 几种特定的 ATPCS

几种特定的 ATPCS 是在遵守基本的 ATPCS 同时，增加一些规则以支持一些特定的功能:

- 支持数据栈检查的 ATPCS；
- 支持只读段位置无关(ROPI)的 ATPCS；
- 支持可读写段位置无关(RWPI)的 ATPCS；
- 支持 ARM,THUMB 程序混合使用的 ATPCS；
- 处理浮点运算的 ATPCS。

7.3.1 支持数据栈检查的 ATPCS

(1) 支持数据栈限制检查的 ATPCS 的基本原理。

- 如果在程序设计期间能够准确的计算程序需要的内在总量，就不需要进行数据栈的检查。但是，通常情况下这是很难做到的，这时需要进行数据栈的检查。
- 在进行数据栈检查时，使用 R10 作为数据栈限制指针，这时 R10 又记作 S1。用户在程序中不能控制该寄存器。具体来说，支持数据栈限制检查的 ATPCS 要满足下面的规则：

① 在已经占用的栈的最低地址必须和 S1 之间必须要有 256 B 的空间。当中断处理程序可以使用用户的数据栈时，除了保留以上要保留的 256 B 空间，还必须为中断处理预留足够的内存空间。

② 用户在程序中不能修改 S1 的值。

③ 数据栈指针 SP 的值不能小于 S1 的值。

- 支持数据栈限制检查的 ATPCS 相关的编译/汇编选项有下面几种：

④ 选项/swst(software stack limit checking) 指示编译器产生的代码遵守支持数据栈限制检查的 ATPCS。用户在程序涉及期间不能准确计算出程序所需要的所有数据栈大小时，需要指定该选项。

⑤ 选项/noswst 指示编译器生成的代码不支持数据栈限制检查功能。用户在程序设计期间能预测数据栈大小时，指定该选项。这是编译器默认选项。

⑥ 选项/swstna(software stack limit checking not applicable) 如果汇编程序对于是否进行数据栈检查无所谓，而与该汇编程序连接的其他程序指定了选项/swst 或者选项/noswst，这时使用选项/swstna。

(2) 编写遵守支持数据栈限制检查的 ATPCS 的汇编语言程序。

- 对于 C/C++语言编写的源程序在编译时指定选项/swst，生成的目标代码将遵守支持

数据栈限制检查的 ATPCS。

- 对于汇编语言程序来说，如果要遵守支持数据栈限制检查的 ATPCS，用于在编写程序时必须满足支持数据栈限制检查的 ATPCS 所要求的规则，然后在汇编时指定/swst 选项。

编写汇编语言程序的一些要求：

① 数据栈小于 256 B 的叶子子程序。数据栈小与 256 B 的叶子子程序不需要进行数据栈限制检查。

② 数据栈小于 256 B 的非叶子子程序。对于该类子程序可以用下面的代码段来进行数据栈检查。

ARM 程序下的代码段：

```
SUB     SP, #-size          ;size 为 SP 和 S1 必须预留的空间大小。
CMP     SP, S1
BLLO    __ARM_stack_overflow
THUMB 程序下的代码：
ADD     SP, #-size
CMP     SP, S1
BLO     __Thumb_stack_overflow
```

③ 数据栈大于 256 字节的子程序。对于该类子程序，为了保证 SP 的值不小于数据栈可用的内存单元最小地址值，需要引入相应的寄存器。

ARM 程序下的代码段：

```
SUB     IP, SP, #size       ;size 为 SP 和 S1 必须预留的空间大小。
CMP     IP, S1
BLLO    __ARM_stack_overflow
THUMB 程序下的代码：
LDR     WR, #-size
ADD     WR, SP
CMP     WR, S1
BLO     __Thumb_stack_overflow
```

7.3.2 支持只读段位置无关(ROPI)的 ATPCS

(1) 支持只读位置无关的 ATPCS 的应用场合。

- 位置无关的只读段可能为位置无关的代码段，也可能为位置无关的数据段。使用支持只读段位置无关的(ROPI)的 ATPCS 可以避免必须将程序放在特定的位置。它经常应用在以下一些场合：

① 程序在运行期间动态加载到内存中；

② 程序在不同的场合与不同的程序组合后加载到内存中；

③ 在运行期间映射到不同的地址。在一些嵌入式系统中，将程序发在 ROM 中，运行时在建在到 RAM 中不同的地址。

(2) 遵守支持只读段位置无关的 ATPCS 的程序设计。这类程序设计规则:

- 当 ROPI 段中的代码引用同一个 ROPI 段中的符号，必须是基于 PC 的；
- 当 ROPI 段中的代码引用另一个 ROPI 段中的符号时，必须是基于 PC 的，并且两个

ROPI 段的位置关系必须固定；

- 其他被 ROPI 段中的代码引用的必须是绝对地址，或是基于 SB 的可写数据；
- ROPI 段移动后，对 ROPI 中的符号的引用要作相应的调整。

7.3.3 支持可读写段位置无关(RWPI)的 ATPCS

如果一个程序中所有的可读写段都时位置无关，则称该程序遵守支持可读写段位置无关(RWPI)的 ATPCS。

- 使用 RWPI 的 ATPCS 可以避免必须将程序存放到特定的位置。这时 R9 通常用作静态基址寄存器，记作 SB。可重入的子程序可以在内存中同是有多个实例，各个实例拥有独立的可读写段。在生成一个新的实例时，SB 指向该实例的可读写段。RWPI 段中的符号的计算方法为：连接器首先计算出该符号相对于 RWPI 段中某一位置的偏移量，通常该特定位置选为 RWPI 段的第一字节出。在程序运行时，将该偏移量加上 SB 即可生成该符号的地址。

7.3.4 支持 ARM 程序和 THUMB 程序混合使用的 ATPCS

在编译和汇编时，使用/interwork 告诉编译器(或汇编器)生成的目标代码遵守支持 ARM,THUMB 程序混合使用的 ATPCS。它用在以下场合：

- 程序中存在 ARM 程序调用 THUMB 程序的情况；
- 程序中存在 THUMB 程序调用 ARM 程序的情况；
- 需要连接器进行 ARM 和 THUMB 状态切换的情况；
- 在下述情况，使用选项/nointerwork。本选项为默认选项。

① 程序不包含 THUMB 程序；

② 用户自己进行 ARM 状态和 THUMB 状态的切换。

注意：在同一个 C/C++源程序中不能同是出现 ARM 指令和 THUMB 指令。

7.3.5 处理浮点运算的 ATPCS

- ATPCS 支持 VFP 体系和 FPA 体系两种不同的浮点硬件体系和指令集。两种体系对应的代码不兼容。
- ADS 的编译器和汇编器有下面 6 种与浮点数有关的选项：

③ -fpu VFP；

④ -fpu FPA；

⑤ -fpu softVFP；

⑥ -fpu softVFP+VFP；

⑦ -fpu softFPA；

⑧ -fpu none。

- 当系统中包含浮点运算部件时，可以选择选项-fpu VFP, -fpu softVFP+VFP 或者-fpu FPA
- 当系统中包含有浮点运算部件，并且想在 THUBM 程序中使用浮点数子程序，可以选择选项-fpu softVFP+VFP。
- 当系统中没有浮点运算部件时，分 3 种情况考虑：

⑨ 如果要与 FPA 体系兼容，应选择-fpu softFPA；

⑩ 如果程序中没有浮点算术运算，并且程序要和 FPA 体系和 VFP 体系都兼容，应 选

择选项-fpu none；

⑪ 其他情况下选择选项-fpu softVFP。

8. ARM 程序和 THUMB 程序混合使用

8.1 概述

(1) ARM 程序和 THUMB 程序混合使用的场合。

- 强调速度的场合；
- 有一些共只有 ARM 能够完成；
- 当处理器进入到异常中断处理程序时，程序自动切换到 ARM 状态；
- ARM 处理器总是从 ARM 状态开始运行。

(2) 在编译和汇编时使用选项-apcs/interwork

- 如果目标代码包含以下内容，则需要使用选项-apcs/interwork

① 需要返回到 ARM 状态的 THUMB 子程序；

② 需要返回到 THUMB 状态的 ARM 子程序；

③ 间接地调用 ARM 子程序的 THUMB 子程序；

④ 间接地调用 THUMB 子程序的 ARM 子程序。

- 当在编译或者汇编时使用了-apcs/interwork 时：

⑤ 编译器和汇编器将 interwork 的属性写入到目标文件中。

⑥ 连接器在子程序入口处提供用于状态切换的小程序(称为 VENEERS)。

⑦ 在汇编语言子程序中，用户必须编写相应的返回代码，使得程序返回到和调用者相同的状态。

⑧ C/C++子程序中，编译器生成合适的返回代码，使得程序返回到和调用者相同的状态。

- 在下列情况下，不必指定/interwork 选项：

⑨ 在 THUMB 状态下发生异常中断时，处理器自动切换到 ARM 状态，这时不需要添加状态切换代码；

⑩ 在 THUMB 状态下发生异常中断时，异常中断处理程序返回不需要添加状态切换。被调用的 ARM 子程序返回时需要相应的切换代码，调用者的 THUMB 程序则不需要；

⑪ THUMB 程序调用其他文件中的 ARM 子程序时，在该 THUMB 程序中不需要状态切换代码；

⑫ ARM 程序调用其他文件中的 THUMB 子程序时，在该 ARM 程序中不需要状态切换代码。

8.2 汇编程序中通过用户代码支持 interwork

对于 C/C++语言来说，编译时指定-apcs/interwork 选项，连接器生成的代码就遵守支持 ARM,THUMB 程序混合的使用 ATPCS。

对于汇编语言来说，可以有两种方法来实现程序状态的切换：

- 利用连接器提供的小程序(veneers)来实现程序状态的切换；
- 用户自己编写状态切换程序。

8.2.1 可以实现程序状态切换的指令

在 ARM4 中可以实现程序状态的切换的指令是 BX。在 ARM5 版本开始，下面的指令也可以实现程序状态的切换：

- BLX；
- LDR、LDM、POP。

(1) BX 指令。

BX 指令跳转到指令指定的目标地址。目标地址处的指令可以是 ARM 指令，也可以是 THUMB 指令，如果目标地址程序状态和 BX 指令处程序状态不同。指令将进行程序状态切换。目标地址值为指令的值和 0xFFFFFFFE 做与操作的结果，目标地址处的指令类型由寄存器<Rm>的 bit[0]决定。

语法格式：

```
BX{<cond>} <Rm>
```

其中：

<Rm>寄存器的 bit[0]为 0 时，目标地址处的指令时 ARM 指令；当<Rm>寄存器的;bit[0]为 1 时，目标地址处的指令为 THUMB 指令。

注意：当 Rm[1:0]=0b10 时，由于 ARM 指令是字对齐的，会产生不可预料的结果。

(2) BLX(1)指令。

这种格式的 BLX 指令从 ARM 指令跳转到指令中指定的目标地址，并将程序状态切换到 THUMB 状态，该指令同时将 PC 寄存器的内容复制到 LR 寄存器中。

语法格式：

```
BLX<target_addr>
```

其中：

target_addr 为指令跳转的目标地址。目标地址的计算方法为:将指令中 24 位的带符号补码立即数扩展为 32 位；将此 32 位数左移两位；将得到的地址值的 bit[1]设置成 H 位；将得到的值加到 PC 寄存器中。由计算方法可知跳转的范围大致为-32 MB ~ +32 MB。

指令的伪代码：

```
LR = address of the instruction after the BLX instruction
T = 1
PC = PC + (SignExted(signed_immed_24)<<2)+(H << 1)
```

(3) BLX(2)指令。

这种格式的 BLX 指令从 ARM 指令跳转到指令中指定的目标地址，并将程序状态切换到 THUMB 状态，该指令同时将 PC 寄存器的内容复制到 LR 寄存器中。

语法格式：

```
BLX{<cond>} <Rm>
```

其中：

<Rm>寄存器的 bit[0]为 0 时，目标地址处的指令时 ARM 指令；当<Rm>寄存器的;bit[0]为 1 时，目标地址处的指令为 THUMB 指令。当<Rm>为 R15 时，会产生不可预知的结果。

指令的伪代码：

```
if cond passed then
LR = address of the instruction after the BLX instruction
```

```
T = Rm[0]
PC = Rm AND 0xFFFFFFFE
```

8.2.2 进行状态切换的汇编程序实例

```
        AREA AddReg, CODE, READONLY
            ENTRY
        main ADR R0, ThumbProg + 1      ;保证最低位为 1，以便切换到 THUMB 状态。
            BX     r0

            CODE16
        Thumbprog
            MOV R2, #2
            MOV R3, #3
            ADD R2, R2, R3
            ADR R0, ARMprog             ;最低位为 0，以便切换到 ARM 状态。
            BX    R0

            CODE32
        ARMprog
            MOV R4, #4
            MOV R5, #5
            ADD R4, R4, R5

        stop MOV R0, #0X18
            LDR R1, =0X20026
            SWI 0X123456

            END
```

8.3 在 C/C++程序中实现 interwork

(1) 需要考虑 interwork 的场合。

- 如果 C/C++程序中包含需要返回到另一种程序状态的子程序时，需要在编译该 C/C++程序时指定选项-apcs/interwork。
- 如果 C/C++程序间接地调用另一种指令系统的子程序，或者 C/C++程序中地虚函数用另一种指令系统的子程序时，需要在编译该 C/C++程序时指定选项-apcs/interwork。
- 如果调用程序和被调用程序时不同指令集的，而被调用者是 non-interwork 代码，这时不要使用函数指针来调用该被调用程序。
- 如果在连接时目标文件包含了 THUMB 程序，这时连接器会选择 THUMB C/C++库进行连接。
- 通常情况下，如果不能肯定程序中不进行程序状态切换，使用该编译选项。

(2) 编译选项-apcs/interwork 的作用。

- tcc -apcs/interwork；

- armcc -apcs/interwork；
- tcpp -apcs/interwork；
- armcpp -apcs/interwork。

指定编译选项-apcs/interwork 后，编译器会进行以下的处理：

- 编译生成的目标程序可能会稍微大一些。THUMB 程序可能大 2%。ARM 程序可能大 1%。
- 对于子叶程序(指不调用其他子程序的子程序)来说，编译器会将程序中的 MOV PC,LR 指令替换为 BX LR。这时因为 MOV PC,LR 指令不进行程序的状态切换。
- 对于非子叶程序，Thumb 编译器将进行一些指令替换。如：

```
POP{R4,R5,PC}
替换为:
POP{R4,R5}
POP{R3}
BX R3
```

- 编译器在目标程序的代码段中写入 interwork 属性，连接器根据该属性插入相应的用于程序状态切换的代码段。

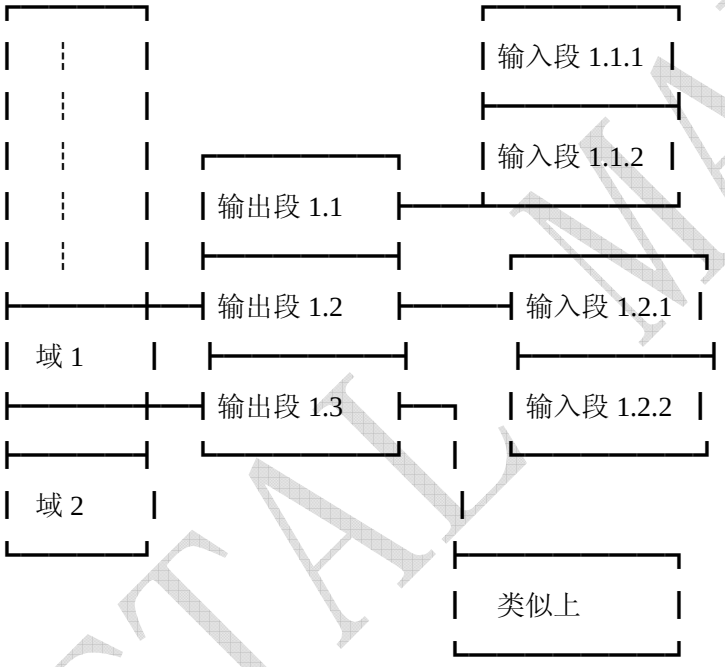
9. ARM 连接器

9.1 ARM 映像文件的组成

(1) ARM 映像文件的组成部分。

ARM 映像文件是一个层次性结构的文件，其中包括了域(region)，输出段(outputsection)和输入段(input section)。各部分的关系如下:

- 一个映像文件由一个或多个域组成;
- 每个域包含一个或多个输出段;
- 每个输入段包含一个或多个输入段;
- 各输入段包含了目标文件中的代码和数据。



ARM 映像文件的组成

下面具体介绍各部分的具体组成。

输入段包含了 4 个内容:

- 代码 ----------CODE
- 已经初始化的数据 ----------INIT DATA
- 未经过初始化的数据存储区域 ----UNINT DATA
- 内容初始化为 0 的存储区域 -------ZI DATA

每个输入段都可以有自己的属性可以为:

- 只读 ----------RO
- 读写 ----------RW
- 零初始化 ------ZI

ARM 连接器根据这些输入段的属性，再把这些输入段分组，再组成不同的输出段以及域。这样我们可以理解:一个输出段就时一些同属性的输入段组成的。这个输出段的属性就时组成它的同属性输入段的属性。一个域通常包含 1-3 个输出段。这些输出段在这个域中的排列顺序由其本身的属性决定的。通常 RO 段排在最前面，其次时 RW 段，最后时 ZI 段。一个域通常映射到物理存储器上。

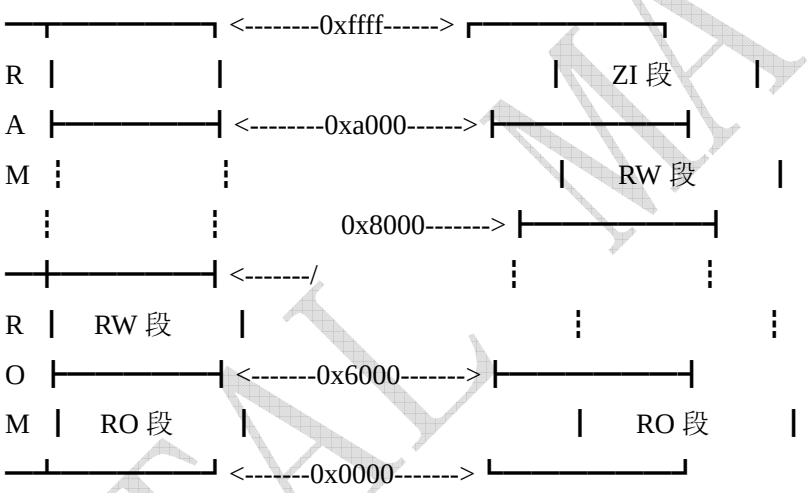
(2) ARM 映像文件个组成部分的地址映射。

ARM 映像文件个组成部分在存储系统中的地址映射有两种:

- 当映像文件位于存储器中时(即映像文件还没有执行之前)，称为加载时地址；
- 当映像文件运行时的地址称为运行地址。

之所以同一个映像文件有两种地址，就是在运行时映像文件的有些域时可以移动到新的存储区域。比如:已经初始化的 RW 属性段的数据所在的段在运行前保存在系统的 ROM 中，当运行时它有可能被移动到 RAM 里。这样可以提高程序的运行速度。

示例：



加载前地址映射关系　　运行时地址映射关系

通常一个映像文件包含若干个域，各域又包含若干个输出段。ARM 连接器就需要知道以下的信息，以决定如何输出相应的映像文件。

- 分组信息 决定如何将各个输入段组织成相应的输出段和域；
- 定位信息 决定各个域在存储空间中的起始地址。

根据映像文件中地址映射的复杂程度，有两种方法告诉 ARM 连接器这些相关的信息。

- 对于映像文件关系比较简单的情况，可以使用命令行选项。
- 对于映像文件关系比较复杂的情况，可以使用一个配置文件。
- 当映像文件包含最多两个域，每个域中可以最多有 3 个输出段时。可以使用如下的连接器选项告诉连接器相关的地址映射关系。

① -ropi

② -rwpi

③ -ro_base

④ -rw_base

⑤ -split

当映像文件地址映射关系比较复杂时，可以使用分散加载配置文件告诉连接器相关的地址映射关系。可以通过以下连接选项:

-scatter filename

9.2 ARM 映像文件的入口点

(1) ARM 映像文件的两类入口点。

- 映像文件运行时的入口点，称为初始入口点(initial entry point)。初始入口点时映像文件运行时的入口点，每个映像文件只有惟一的初始入口点。它保存在 ELF 头文件中。如果该映像文件是被操作系统加载的，操作系统正是通过跳转到该入口点的地址来执行加载该映像文件。
- 普通的入口点(entry point)。它时在汇编程序中用 ENTRY 伪操作定义的。它通常用于标识该代码段时通过异常中断程序进入的。这样连接器删除无用的段时不会将该代码段删除。一个映像文件可以有多个普通入口点。

注意：初始入口点可以是普通入口点，但也可以不是入口点。

(2) 定义初始入口点。

初始入口点必须满足下面两个条件:

- 初始入口点必须位于映像文件的运行时域内。
- 包含初始入口点的运行时域不能被覆盖，它的加载时地址和运行时地址必须是相同的(这种域称为 root region)。

这时可以用 -entry address 来指定映像文件的初始入口点。对于地址 0x0 处为 ROM 的嵌入式应用系统，可以使用-entry 0x0 来指定映像文件的初始入口点。当系统复位后，自动跳转到该入口点开始执行。如果映像文件是被一个加载器加载的，如:被一个引导程序或者操作系统加载，该映像文件必须包含一个初始入口点。

(3) 当用户没有指定连接选项-entry address 时，连接器根据下面的规则来决定映像文件的初始入口点。

- 如果输入的目标文件中只有以个普通入口点，该普通入口点被连接器当作是映像文件的初始入口点。
- 如果输入的目标文件中没有一个普通入口点，或者其中的普通入口点数目多于一个，则连接器生成的映像文件中不包含初始入口点，并且产生如下的警告信息:

```
L6035W : Image does not have an entry point .(not specified of not set due to multiple choices)
```

(4) 普通入口点的使用方法。

- 普通入口点在汇编源程序中用 ENTRY 伪操作定义。在嵌入式应用系统中，各种异常中断的处理入口使用普通入口点标识。这样连接器在删除无用段时不会将该代码段删除；
- 一个映像文件可以有多个普通入口点；
- 当没有指定-entry address 选项时，如果输入的目标文件中只有一个普通入口点，该普通入口点被连接器当成映像文件的初始入口点。

9.3 输入段的排序规则

连接器根据个输入段的属性来组织这些输入段。具有相同属性的输入段被放到域中一个段连续的空间中，组成一个输出段。在一个输出段中，个输入段的起始地址与输入段的起始地址和该输出段中个输入段的排列顺序有关。

(1) 通常情况下，一个输出段中的输入段的排列顺序是由下面几个因素决定的。用户可以使用-fist 和-last 来改变这些规则。

- 输入段的属性按照输入段的属性其排列顺序如下:

① 只读代码段 ---------- RO-CODE；

② 只读数据段 ---------- RO-DATA；

③ 可读写代码段 ---------- RW-CODE；

④ 其他已经初始化的数据段 ----- I-DATA；

⑤ 未初始化的数据 ---------- UI-DATA。

- 输入段的名称

具有相同属性的输入段，按照其名称来排序。这时，输入段的名称时区分大小写的，按照ASCII 码顺序进行排序。个输入段在连接命令行的输入段列表中的排列顺序。

⑥ 具有相同属性和名称的输入段，按照其在输入段列表中的顺序进行运行排序。

⑦ 可以使用连接选项-first, -last 来改变上述的输入段排序规则。如果在连接时使用了配置文件，可以在配置文件中通过伪属性 FIRST,LAST 达到相同的效果。

⑧ 连接选项-first，-last 选项不能改变根据段属性进行的排序。它只能改变根据输入段名称和其在输入段列表中的顺序的排序规则。也就是说，如果使用连接选项-first 指定的一个输入段，只有该输入段所在的输出段位于运行时域的开始位置时，该输入段才能位于整个运行时域的开始位置。

⑨ 在整个输入段排好序后，在确定这个段的起始地址之前，可以通过填充“补丁”使输入段满足地址对齐要求。

9.4 ARM 连接器介绍

(1) ARM 开发包中的连接器 ARMLINK 可以完成以下操作：

- 连接编译后得到的目标文件和相应的 C/C++库，生成可执行的映像文件。
- 将一些目标文件进行连接，生成一个新的目标文件，供将来进一步连接时使用，这称为部分连接。
- 指定代码和数据在内存中的位置。
- 生成被连接文件的调试信息和相互间调用的信息。

(2) ARMLINK 在进行部分连接和全部连接生成可执行的映像文件所进行的操作时不同的。

- 完全连接:

① 解析输入的目标文件之间的符号引用关系。

② 根据输入目标文件对 C/C++函数的调用关系，从 C/C++运行时库中提取相应模块。
③ 将整个输入段排序，组成相应的输出段。
④ 删除重复的调试信息段。
⑤ 根据用户指定的分组和定位信息，建立映像文件的地址映射关系。
⑥ 重定位需要重定位的值。
⑦ 生成可执行的映像文件。

- 部分连接:

⑧ 删除重复的调试信息段。
⑨ 最小话符号表的大小。
⑩ 保留那些未被解析的符号。
⑪ 生成新的目标文件。

(3) ARMlINK 命令行选项:

- 提供关于 ARMLINK 帮助信息的选项。

① -help
② -vsn

- 指定输出文件名称和类型。

③ -output
④ -partial
⑤ -elf

- 指定选项文件，其中可以包含一些连接选项。

⑥ -via

- 指定可执行映像文件的内存映射关系。

① -rwpi
② -ropi
③ -rw_base
④ -ro_base
⑤ -spit
⑥ -scatter

- 控制可执行映像文件的内容。

⑦ -first
⑧ -last
⑨ -debug/-nodebug
⑩ -entry
⑪ -keep
⑫ -libpath
⑬ -edit

⑭ -locals/-nolocals

⑮ -remove/-noremove

⑯ -scanlib/-noscanlib

- 生成与映像文件相关的信息。

⑰ -callgrah

⑱ -infor

⑲ -map

⑳ -symbols

21 -symdefs

22 -xref

23 -xreffrom

24 -xrefto

- 控制 ARMLINK 生成相关的诊断信息。

25 -errors

26 -list

27 -verbose

28 -strict

29 -unsolved

30 -mangled

31 -unmangled

9.5 ARM 连接器生成的符号

ARM 连接器定义了一些符号，这些符号都包含了字符$$。ARM 连接器在生成映像文件时，用它们来代表映像文件中各域的起始地址及存储区域界限，各输出段的起始地址和区域界限，各输入段的起始地址和区域的界限。

如:

- load$$region_name$$base 代表域 region_name 加载时的起始地址。
- image$$region_name$$base 代表域 region_name 运行时的起始地址。

32 这些符号可以被汇编程序引用，用于地址重定位。这些符号可以被 C 程序作为外部符号引用；

33 所有这些符号，只有在其被引用时，ARM 连接器才会生成该符号；

34 推荐使用映像文件中域相关的符号，而不是使用与段相关的符号。

9.5.1 连接器生成的与域相关的符号

各符号的命名规则是:

- 如果使用了地址映射配置文件(scatter 文件)，该文件内规定了映像文件中各域的名称。
- 如果未使用地址映射配置文件(scatter 文件),连接器按照下面的规定确定各符号中的 region_name :

35 对于只读的域，使用名称 ER_RO；

36 对于可读写的域，使用名称 ER_RW；

37 对于使用 0 初始化的域，使用名称 ER_ZI。

连接器生成的与域相关的符号:

表格 9.5.1 连接器生成于域有关的符号

符号名称	含义
Load$$region_name$$Base	域 region_name 的加载时起始地址
Image$$region_name$$Base	域 region_name 的运行时起始地址
Image$$region_name$$Length	域 region_name 的运行时的长度(为 4 字节的倍数)
Image$$region_name$$Limit	域 region_name 运行时存储区域末尾的下一个字节地址(通常是另一个域的开始地址)

对于映像文件的每个域，如果其中包含了 ZI 属性的输出段，该连接器将会为该 ZI 输出段生成另外的符号。这些符号为:

表格 9.5.2 包括 ZI 段后的符号

符号名称	含义
Image$$r_n$$ZI$$Base	域 region_name 中 ZI 输出段的运行时起始地址
Image$$r_n$$ZI$$Length	域 r_n 中 ZI 输出段运行时的长度(为 4 字节的倍数)
Image$$r_n$$ZI$$Limit	域 region_name 运行时存储区域末尾的下一个字节地址(通常是另一个域的开始地址)

9.5.2 连接器生成的与输出段相关的符号

如果未使用地址映射配置文件(scatter)，连接器生成的与输出段相关的符号如下表所示:

表格 9.5.3 连接器生成的与输出段相关的符号

符号名称	含义
Image$$RO$$Base	RO 输出段运行时起始地址
Image$$RO$$Limit	RO 输出段内存区域的末尾的下一字节地址
Image$$RW$$Base	RW 输出段运行时起始地址
Image$$RW$$Limit	RW 输出段内存区域的末尾的下一字节地址
Image$$ZI$$Base	ZI 输出段运行时起始地址
Image$$ZI$$Limit	ZI 输出段内存区域的末尾的下一字节地址

如果使用了地址映射配置文件，上表所列的符号没有意义，如果应用程序使用了这些符号将可能得到错误的结果，这时应该使用上一节中介绍的与域相关的符号。

9.5.3 连接器生成的与输入段相关的符号

ARM 连接器为映像文件中的每一个输入段生成两个符号，如表:

符号名称	含义
SectionName$$Base	SectionName 输入段的运行时起始地址
SectionName$$Limit	SectionName 输入段运行时存储区域界限

9.6 连接器的优化功能

ARM 连接器的主要功能主要包括删除映像文件中重复的部分一节插入小代码段实现 ARM 状态到 THUMB 状态的转换以及长距离跳转。

(1) 删除重复的调试信息段。在 ARM 中，编译器和汇编器为每个源文件生成一个调试信息段。ARM 连接器可以删除重复的调试信息段，仅保留一个版本，从而很大地减小生成的目标文件的大小；

(2) 删除重复的代码段。ARM 连接器可以删除重复的代码段。有时这些代码段可能来自统一运行时库的不同类型的文件，ARM 连接器尽力选择最合适的一个版本；

(3) 删除未使用的段。

- * ARM 连接器默认情况下会删除映像文件中未被使用的代码和数据。有些连接选择可以控制这个操作。
- 连接选项 -info unused 可以列出被删除的未使用代码段。
- 如果一个段要最终的保留在映像文件中，它必须满足下列条件之一。

① 其中包含了普通入口点或者初始入口点；

② 被包含了普通入口点或者初始入口点的输入段 noweak 方式引用的段；

③ 使用连接选项 -first 或者 -last 指定的段；

④ 使用连接选项 -keep 指定的段。

(4) 生成小代码段。

- ARM 连接器可以根据需要生成一些小代码段，称为 veneer。这些小代码段用于实现 ARM 状 态到 THUMB 状态的转换以及长距离跳转。当跳转指令涉及到处理器在 ARM 状态和 THUMB 状态之间进行转换时，或者时跳转指令的目标地址超出了该跳转指令所能到达的最大地址时，ARM 连接器根据需要将生成一些小的代码段以实现这些功能。
- ARM 连接器为每个 veneer 生成一个代码段，称为 Veneer$$Code。如果两个输入段长距离跳转到同一个目标段，生成一个 veneer，使两个输入段都可以到底该 veneer。

ARM 连接器产生的 veneer 按照其功能进行分类包括:

① ARM 状态到 ARM 状态的长跳转；

② ARM 状态到 THUMB 状态的长跳转；

③ THUMB 状态到 ARM 状态的长跳转；

④ THUMB 状态到 THUMB 状态的长跳转。

9.7 运行时库的使用

/***/

待

/***/

9.8 从一个映像文件中使用另一个映像文件中的符号

/***/

待

/***/

9.9 隐藏或者重命名全局符号

/***/

待

/***/

9.10 ARM 连接器命令行选项

(1) -help 帮助

(2) -vsn 显示 ARM 连接器版本信息

(3) -partial 部分连接操作

(4) -output file 指示输出文件。映像文件为：.axf,目标文件为：.o

(5) -elf 输出 ELF 格式的文件。

(6) -ro-base address 设置映像文件 RO 属性的输出段的加载、运行时的地址。

(7) -ropi 指定映像文件加载时域和运行时域是位置无关的(PositionIndependent)。连接器保证以下操作:

- 检查各段之间的重定位关系，保证其是合法的；
- 保证 ARM 连接器自身输出的代码是只读位置无关的。通常情况下只读属性的输入段应该时 PI 的。

(8) -rw-base address 设置映像文件 RW 属性的输出段的运行时的地址；

(9) -rwpi 指定映像文件包含 RW 属性和 ZI 属性的输出段的加载时域和运行时域是位置无关的。指定本操作，ARM 连接器保证以下操作：

- 检查并确保各 RW 属性的运行时域包含的各输入段 PI。
- 检查各段之间的重定位关系，保证其时合法的。
- 在 Region$$Table 和 ZIsection$$Table 中添加基于静态寄存器 SB 的条目。通常情况下可读写属性的输入段应该时 PI 的。

(10) -split 将包含 RW 属性和 RO 属性的输出段的加载时域分割成两个加载时域其中:

- 一个加载时域包含所有的 RO 属性的输出段。可以使用-ro-base 来修改其加载时的地址。
- 一个加载时域包含所有的 RW 属性的输出段。可以用-rw-base 来修改其加载时地址。

(11) -scatter file 指定分散加载文件；

(12) -debug 指定在输出文件中包含调试信息。这些调试信息包含调试信息输入、符号表以及字符串表；

(13) -nodebug 指定在输出文件中不包含调试信息。这时调试器不能提供源码级别的调试功能。这时连接器会处理加载到调试器中的映像文件(.axf 文件)，但是

不处理加载到存储器中的映像文件(.bin 文件)；

- ARM 连接器进行部分连接，则生成的目标文件不包含调试信息，单依然包含了符号表以及字符串表。
- 如果将来要使用 fromELF 来转换映像文件的格式，则在生成该映 像文件时不要使用-nodebug 选项。

(14) -remove 删除映像文件中没有使用的段。其中可以指定只要删除的段。如:-remove RO / -remove RW / -remov ZI；

(15) -unremove 不删除映像文件中没有使用的段；

(16) -entry location 指定映像文件中的初始入口点的地址。初始入口点必须满足下面的条件：

- 必须位于映像文件的运行时域内；
- 包含初始入口点的运行使用不能被覆盖，它的加载时地址和运行时地址必须相同。

选项中的 location 可能的取值有以下几条:

- 入口的点的地址。如：-entry address
- 地址符号：symbol。如：-entry int-handler。如果被指定的符号在映像文件中多个定义，连接器报错。
- 相对于某个目标中特定的一定的偏移量。如：-entry 8+startup

(17) -keep section-id 指定目标 section-id 不能被删除。section-id 可能的取值有：

- symbol。 包含 symbol 符号的输入段都将被保留。
- object 目标文件中的段将被保留。如：-keep XXX.o。可以在 ARM 连接器命令行包含多个这样的-keep。
- object。当目标文件中只有一个输入段时，该段被保留。

(18) -first section-id 将输入段 section-id 放置在运行时域的开始位置。

(19) -last section-id 将输入段 section-id 放置在运行时域的最后位置。

(20) -libpath pathlist 指定 ARM 标准 C/C++运行时库的路径。

(21) -scanlib 指示 ARMLINK 扫描默认的 C/C++运行时库。

(22) -noscanlib 指示 ARMLINK 进行连接操作时，不扫描默认的 C/C++库。

(23) -locals 指示 ARM 连接器生成映像文件时，将局部符号也保存到输出符号列表中。

(24) -nolocals 指示 ARM 连接器生成映像文件时，不要将局部符号也保存到输出符号列表中。当用户需要减小映像文件大小的时候可以使用该选项。

(25) -callgrah 指示连接器生成以个基于 HTLM 格式的静态的函数库调用图。该图包含了映像文件中所有函数的定义和引用情况。对于函数 func()来说，函数调用图包含了以下信息：

- 函数 func()编译时的处理器状态。
- 调用本函数的函数集合。
- 被本函数调用的函数集合。
- 函数 func()在映像文件中被寻址的次数。此外，函数调用图还包含了函数的以下特征：

① 被 interworking 的小代码段调用的函数。

② *在本映像文件之外调用的函数。

③ *允许未被定义的函数，如被以 weak 方式引用的函数。函数调用图还表示了数据栈使用的相关信息：

④ *每次调用使用数据栈的大小。

⑤ *函数调用使用数据栈最大栈的大小。

(26) -info topics 显示特定种类的信息：

- size：显示映像文件个输入段或者 C/C++运行时库成员 的代码和数据大小。包含 RW 数据段,RO 数据段,ZI 数据段和调试数据。
- totals：显示映像文件中所有输入段或者 C/C++运行时库成员的代码和数据大小的总和。
- veneers：显示 ARM 连接器产生的 veneers 的详细信息。
- unused：显示用-remove 删除未被使用的段的信息。

(27) -map 产生一个关于映像文件的信息图。

(28) -symbols 列出连接过程中的局部和全局符号及其数值，包括连接器产生的符号。

(29) -symdefs file 生成 sysdefs 文件。有两种情况：

- 如果连接选项中指定的文件 file name 不存在，在 ARM 连接器生成的包括所有全局符号的 symdefs 文件。
- 如果连接选项中指定的文件 file name 存在，则该文件的内容将限制 ARM 连接器生成的 symdefs 文件中包括那些符号。

(30) -edit file 指定一个 steering 类型的文件，用于修改输出文件中的输出符号表的内容。steering 类型的文件可以完成以下操作：

- 隐藏全局符号。
- 重命名全局符号。

(31) -xref 列出所有输入段之间的交叉引用。

(32) -xreffrom object(section) 列出所有从目标文件 object 中的 section 段到其他输入段的引用。

(33) -xrefto object(section) 列出所有从其他输入段到目标文件 object 中的 section 段的引用。

(34) -errors file 将诊断信息从标准输出流重定向到文件 file 中。

(35) -list file 将连接选项-info,-map,-symbol,-xref,-xreffrom,-xrefto 的输出重定向到文件 file 中。

(36) -verbose 显示本次连接操作的详细信息。

(37) -unmangled 指示连接器在诊断信息和连接选项-ref，-xrefrom,-xrefto,-symbol 产生的列表中显示 unmangled 的 C++符号名称。

(38) -mangled 指示连接器在诊断信息和连接选项-ref，-xrefrom,-xrefto,-symbol 产生的列表中显示 mangled 的 C++符号名称。

(39) -via file 指定 via 格式的文件。via 格式的文件中包含了 ARM 连接器各命令行的选项，ARM 连接器可以从该文件中读取相应的连接器命令行选项。

(40) -strict 指示连接器将可能早餐失效的条件作为错误信息来报告，而不时作为警告信息来报告。

(41) -unresolved symbol 其中 symbol 为一个已经定义的全局符号。ARM 连接器在进行连接操作时，将所有未被解析的符号引用指向符号 symbol。

(42) -input-file-list 选项-input-file-list 是一个用空格分割的目标文件和库文件的列表。

9.11 使用 scatter 文件定义映像文件的地址映射

9.11.1 scatter 文件概述

scatter 文件是一个文本文件，它可以用来描述连接器生成映像文件时需要的信息。具体来说，在 scatter 文件中可以指定下列信息。

- 各个加载时域的加载时起始地址和最大尺寸；
- 各个加载时域的属性；
- 从每个加载时域分割出的运行时域；
- 各个运行时域的运行时起始地址和最大尺寸；
- 各个运行时域存储访问特性；
- 各个运行时域的属性；
- 各个运行时域包含的输入段。

这里用 BNF 语法来描述 scatter 文件的格式。BNF 语法的基本元素如表：

表格 9.11.1 BNF 语法的基本元素

符号	含义
A::=B	将 A 定义成 B
[A]	A 为可选项
A+	A 重复 1 多任意多次
A*	A 重复 0 多任意多次
A | B	或者 A 或者 B
(AB)	A 与 B 时一起出现的

9.11.2 scatter 文件中各部分介绍

scatter 文件各组成部分的语法格式：

(1) 加载时域的描述。

加载时域包括名称、起始地址、属性、最大尺寸和以各运行时域的列表。

运用 BNF 语法(不完全)描述，加载时域的格式如下所示：

```
load_region_description ::= load_region_name
base_designator [attribute_list][max_size]
{
    execution_region_description+
}
base_designator ::= base_address | (PLUS offset)
```

详解：

- load_region_name 为本加载时域的名称。该名称中只有前 31 个字符有意义。
- base_designator 用来表示本加载时域的起始地址，可以有下面两种格式：

① base_address 表示本加载时域中的对象在连接时的起始地址。地址必须是字对齐的。

② +offset 表示本加载时域中的对象在连接时的起始地址是在前一个加载时域的结束地址后偏移量 offset 字节处。本加载时域是第一个加载时域，则它的起始地址即 offset。

③ attribute_list 表示本加载时域中的属性，其可能的取值为下面之一。默认的取值为 ABSOLUTE。

- PI 位置无关属性。

- RELOC 重定位。
- OVERLAY ADS 目前没有提供地址空间重叠的管理机制。如果有加载时域地址空间重叠，需要用户自己提供地址空间重叠的管理机制。
- ABSOLUTE

④ max_size 指定本加载时域的最大尺寸。如果本加载时域超出了该值，连接器将报告错误。默认的取值为 0XFFFFFFFF。

⑤ execution_region_description 含义后面有介绍。

(2) 运行时域的描述。

运行时域包括名称、起始地址属性、最大尺寸和一个输入段的集合。

```
execution_region_description ::= exec_region_name
base_descriptor[attribute_list][max_size]
{
    input_section_description*
}
base_designator ::= base_address | (PLUS offset)
```

详解：

- exec_region_name 为本运行时域的名称。该名称中只有前 31 个字符有意义。
- base_designator 用来表示本加载时域的起始地址，可以有下面两种格式：

① base_address 表示本加载时域中的对象在连接时的起始地址。地址必须是字对齐的。

② +offset 表示本运行时域中的对象在连接时的起始地址是在前一个运行时域的结束地址后偏移量 offset 字节处。本运行时域是第一个运行时域，则它的起始地址即 offset。

③ attribute_list 表示本运行时域中的属性，其可能的取值为下面之一。默认的取值为 ABSOLUTE。

* PI 位置无关属性。

* RELOC 重定位。

* OVERLAY ADS 目前没有提供地址空间重叠的管理机制。如果有运行时域地址空间重叠，需要用户自己提供地址空间重叠的管理机制。

* ABSOLUTE

④ max_size 指定本运行时域的最大尺寸。如果本运行时域超出了该值，连接器将报告错误。

⑤ input_region_description 含义后面有介绍。

(3) 输入段描述。

这里描述了一个模式，符合该模式的输入段都将包含在当前域中。其格式使用 BNF 语法描述，如下：

```
input_section_description ::= module_selector_pattern [{input_selectors}]
```

详解：

- module_selector_pattern 定义了一个文本字符串的模式。其中可以使用匹配符，符号*代表零个或者多个字符，符号?代表单个字符。进行匹配时，所有字符是大小写

无关的。满足下面条件之一，认为该输入段是与 module_selector_pattern 匹配的。

① 包含输入段的目标文件的名称与 module_selector_pattern 匹配。

② 包含输入段的库成员名(不带前导路径)与 module_selector_pattern 匹配。

③ 包含输入段的库的代路径名称与 module_selector_pattern 匹配。

- input_selectiors 含义后面介绍。

(4) 输入段选择符。

输入段选择符定义了一个用逗号分割的模式列表。该列表中的每个模式定义了输入段名称或者输入段属性的匹配方式。当匹配模式使用输入段名称时，它前面必须使用符号+，而符号+前面紧跟的逗号可以省略。描述如下：

```
input_selectiors ::= (PLUS input_selection_attrs | input_section_pat)
([COMMA] PLUS input_section_attrs | COMMA input_section_pat)*
```

详解：

- input_selectiors 定义了输入段的属性匹配模式，这些属性匹配模式是大小写无关的，包括：

① RO-CODE

② RO-DATA

③ RO 包括了 RO-CODE,RO-DATA

④ RW-DATA

⑤ RW 包括了 RW-CODE,RW-DATA

⑥ ZI

⑦ CODE 同 RO-CODE

⑧ CONST 同 RO-DATA

⑨ TEXT 是 RO 的同义词

⑩ DATA 是 RW 的同义词

⑪ BSS 是 ZI 的同义词

可以使用属性 FIRST，LAST 来指定某输入段处于本运行时域的开头或者结尾。使用.ANY 标识以个输入段后，连接器可以根据情况将该输入段安排到任何一个它认为合适的运行时域。

- * 定义输入段名称的匹配模式。其中可以使用匹配符，符号*代表零个或者多个字符，符号？代表单个字符。进行匹配时，所有字符是大小写无关的。

9.11.3 scatter 文件使用举例

(1) 1 个加载时域和 3 个连续运行时域。

在本例中，映像文件包括一个加载时域和 3 个连续运行时域。这种模式，适合那些将其他程序加载到 ARM 中的程序，如系统的引导程序和 ANGEL 等。

scatter 文件内容：

命令：-ro-base 0x80000000

```
LR_1 0X80000000        ;定义加载时域的名称为 LR_1，起始地址为 0x80000000
{                      ;开始定义运行时域。
    ER_RO +0           ;第一个运行时域的名称为：ER_RO 。
    {                  ;其起始地址为 0X80000000 。
        *(+RO)         ;本域包含了所有的 RO 属性的输入段，它们被连续放置。
    }

    ER_RW +0           ;第二个运行时域的名称为：ER_RW 。
    {                  ;它的起始地址为上个域结束位置的下一个地址。
        *(+RW)         ;本域包含了所有 RW 属性的输入段，它们被连续放置。
    }

    ER_ZI +0           ;第三个运行时域的名称为：ER_ZI 。
    {                  ;它的起始地址为上个域结束位置的下一个地址。
        *(+ZI)         ;本域包含了所有 ZI 属性的输入段，它们被连续放置。
    }
}
```

解释：在映像文件运行之前，该映像文件包括一个单一的加载时域，该加载时域中包含所有的 RO 属性的输出段和 RW 属性的输出段，ZI 属性的输出段此时还不存在。在映像文件运行时，生成 3 个运行时域，属性为 RO,RW 和 ZI。其中分别包含了 RO 输出段，RW 输出段和 ZI 输出段。RO 属性的运行时域和 RW 属性的运行时域的起始地址与其加载地址相同，这就不需要进行数据移动，ZI 属性的运行时域在映像文件开始执行之前建立。

(2) 1 个加载时域和 3 个不连续运行时域。

在本例中，映像文件包括一个加载时域和 3 个不连续运行时域。这种模式，适合域嵌入式应用场合。

scatter 文件内容：

命令： -ro-base 0x80000000

-rw-base 0x80004000

```
LR_1 0X80000000        ;定义加载时域的名称为 LR_1，起始地址为 0x80000000
{                      ;开始定义运行时域。
    ER_RO +0           ;第一个运行时域的名称为：ER_RO 。
    {                  ;其起始地址为 0X80000000 。
        *(+RO)         ;本域包含了所有的 RO 属性的输入段，它们被连续放置。
    }

    ER_RW 0x80004000 ;第二个运行时域的名称为：ER_RW 。
    {                  ;它的起始地址为 0x80004000
        *(+RW)         ;本域包含了所有 RW 属性的输入段，它们被连续放置。
    }

    ER_ZI +0           ;第三个运行时域的名称为：ER_ZI 。
    {                  ;它的起始地址为上个域结束位置的下一个地址。
        *(+ZI)         ;本域包含了所有 ZI 属性的输入段，它们被连续放置。
```

```
        }
    }
```

(3) 2 个加载时域和 3 个不连续的运行时域

scatter 文件内容：

命令： -split

-ro-base 0x80000000

-rw-base 0x80004000

```
LR_1 0X80000000        ;定义第一加载时域的名称为 LR_1，起始地址 0x80000000
{                      ;开始定义运行时域。
    ER_RO +0           ;第一个运行时域的名称为：ER_RO 。
    {                  ;其起始地址为 0X80000000 。
        *(+RO)         ;本域包含了所有的 RO 属性的输入段，它们被连续放置。
    }
}

LR_2 0X80004000        ;定义第二加载时域的名称为 LR_2，起始地址为 0X80004000
{
    ER_RW +0           ;第二个运行时域的名称为：ER_RW 。
    {                  ;它的起始地址为 0x80004000
        *(+RW)         ;本域包含了所有 RW 属性的输入段，它们被连续放置。
    }

    ER_ZI +0           ;第三个运行时域的名称为：ER_ZI 。
    {                  ;它的起始地址为上个域结束位置的下一个地址。
        *(+ZI)         ;本域包含了所有 ZI 属性的输入段，它们被连续放置。
    }
}
```

(4) 固定的运行时域。

在一个映像文件中需要指定一个初始入口点(initial entry point)，它是影响文件运行时的入口点。初始入口点必须位余一个固定域中。所谓固定域是指该域的加载时地址和运行地址时相同的。如果初始入口点不时位于一个固定域中。ARM 连接器会产生下面的错误信息：

```
L6203E: Entry point(0x00000000)lies within non-root region 32bitARM
```

使用 scatter 文件时，可以有下面两种方法来设置固定域：

加载地址相同。这样该运行时域就时一个固定域。具体操作步骤：

- 将运行时域的地址指定为其所加载时域地址相同的值，这里的运行时域时该加载时域包含的第一个域；
- 在指定运行时域的地址时，设定起始地址值或者设定偏移量 offset 为 0；
- 设置该运行时域属性为 ABSOLUTE，这也是默认的值。

(5) 使用 FIXED 属性将某个域放置在 ROM 中固定位置。

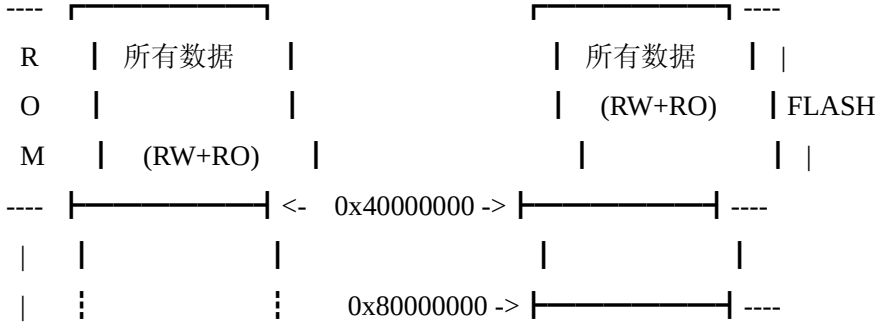
使用 FIXED 属性还可以将映像文件的特定内容放置在 ROM 中的特定位置。本例中将数据块 datablock.o 放置在 0x70000000 处，这样便于使用指针来访问该数据块。同时本例说明了属性.ANY 的用法。

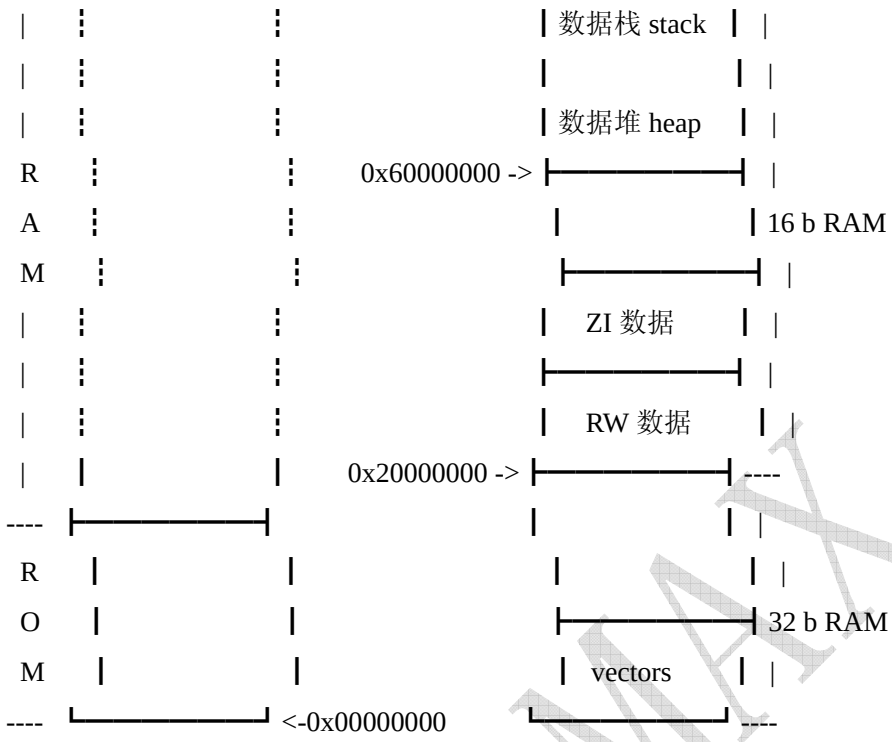
scatter 文件内容：

```
LOAD_ROM 0X00000000        ;第一加载时域，起始地址为 0x00000000
{
    ER_INIT 0X00000000     ;第一运行时域 ER_INIT，起始地址为 0x00000000
    {
        init.o(+RO)            ;本运行时域包含了 init.o 代码
    }

    ER_ROM +0                  ;第二个运行时域，紧跟 ER_INIT 之后。
    {                          ;使用.ANY 属性，表示可以用那些没有被指定特别的
        .ANY(+RO)              ;定位信息的输入段来填充本域。
    }

    DATABLOCK 0X70000000 FIXED   ;第三个运行域，起始地址规定 0X70000000
    {
        data.o(+RO)            ;本域包含了 data.o
    }

    ER_RAM 0X80000000          ;第四个域从 0X80000000 处开始放置
    {
        *(+RW,+ZI)             ;本域中包含了 RW,ZI 数据。
    }
}
```

(6) 一个接近实际系统的例子。

- 在一个嵌入式系统中，为了保持好的性能价格比，通常存在多种存储器。
- 在本系统中，包含了 FLASH 存储器，16 位的 RAM 和 32 位的 RAM。在系统运行之前所有的程序和数据都保存在 FLASH 存储器中。
- 系统启动后，包含异常中断处理和数据栈的 vectors.o 模块被移动到 32 位的片内 RAM 中，在 32 位 RAM 里运行速度可以加快。RW 数据和 ZI 数据被移动到外部 16 位 RAM 中。其它多数的 RO 代码在 FLASH 存储器中运行，它们所在的域为固定域。
- 作为嵌入式系统，在系统复位时，RAM 中不包含任何程序和数据，这时所有的程序和数据都存放在 FLASH 中。通常在系统复位时把 FLASH 存储器映射到地址 0x00000000 处，从而使系统可以开始运行。在 FLASH 存储器中的前几条指令实现重新 RAM 映射到地址 0x00000000 处。
- 图例：





系统复位时地址映射关系　　存储映射建立后地址映射关系

详解：

- 本例中在映像文件运行之前，即加载时，该映像文件包括单一的加载时域。该加载时域包括了所有的 RO 属性的输出段和 RW 属性的输出段，ZI 属性的输出段此时还不存在。从图中可以看出此时的所有数据被保存在以 0X40000000 开始的 FLASH 存储器中。
- 系统复位后，FLASH 存储器被系统中的存储器管理部件映射到地址 0X0000000 处。绝大多数的 RO 代码直接在 FLASH 存储器中运行，它们加载时的地址和运行时的地址相同，该域称为固定域。包含异常中断处理和数据栈的 vectors.o 模块被移动到 32 位的片内 RAM 中其起始地址为 0X00000000(这时 ARM 存储器管理系统已经重新进行了地址映射)。在这里可以得到较快的运行速度。RW 数据和 ZI 数据被移动到 16 位片外 RAM 中。其起始地址为：0X20000000

能实现上述地址映射的 scatter 文件：

```
FLASH 0X4000000 0X00080000              ;定义一个加载时域，名称为 FLASH。
{
    FLASH 0X40000000 0X00080000     ;第一个运行时域地址为 0X40000000，位于 FLASH
    {
        init.o(Init, +First)          ;init.o 被放在了本域的开头处。
        *(+RO)                              ;本域包含绝大多数的 RO 代码。
    }

    32bitRAM 0x00000000 0x20000000;第二个运行时域，在 32b RAM 中。
    {
        vectors.o(Vect, +First)     ;这里包含了 vector.o 模块，且其在头部。
```

```
        }

        16bitRAM 0x20000000 0x60000000;第三个运行时域，在 16bRAM 中
        {
            *(+RW,+ZI)                    ;包含了所有的 RW,ZI 属性的输入段。
        }
    }
```

10. 结束语

通过几个月“游击式”的排版，终于有了能够稍微拿得出手的版本了。其他的也不多说，花了不少精力在这上面，也算是给我寒假的努力有个交代。当中的缺陷肯定不少，毕业临近确实没有多少时间来弄这个东西了。

还想重申一下，选择适合自己的书，适合自己的才是最好的，可以让学习少走弯路。我还是希望这个文档中的东西能够有您用得上的地方，足矣！

期待下一个更有品质的笔记出现吧！

METAL MAX

2008 年 5 月 1 日

11. 致谢

感谢所有值得感谢的人 ：)!