```
/*****************************************************************************
**********************************文件头*************************************
**                          《ARM体系结构与编程》读书笔记
**
**                                        成功的步伐在您不懈的努力中加快!
**
**源自:      METAL MAX, CUIT
**起始日期：01/25/08
**当前版本：Version0.08.0125
**
**备注：  1.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,
           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,
           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!

         2.由于刚接触ARM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
           西理解不是很透彻，甚至会又错误。有些地方加入了自己的一些东西(主要是一些
           的理解和自己做的图表)。

         3.由于本人水平太菜的缘故，排版不工整，存在错别字等问题，见谅！更希望有心人
           忙修改和完善，众志成城!

         4.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
           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
           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。

         5.编辑时使用了三号新宋体字体以及演示版的Editplus2，TABLE缩进为4，制表符代
           空格。

         6.非常喜欢我的事业，也很希望能和大家一起学习很进步，当然交流是前提!
           Email : liyangbbs@126.com
           QQ: 249456711
日志:    01/25/08 - 02/10/08     Version0.08.0125    METL MAX

*****************************************************************************

2008-1-25

一、前言:
    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。


第一章、ARM体系的结构和特征。

CHAP1.1
AMR芯片具有RISC体系的一般特点，如:
        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
      器之间传递数据。
    3.寻找方式简单。
```

4.采用固定长度的指令格式。

AMR芯片具有的一些特别的技术：
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。

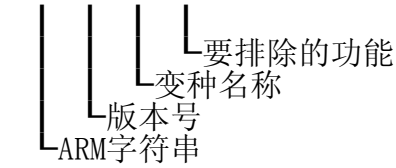
CHAP1.2
AMR体系结构的版本和命名方法。
迄今为止，ARM体系结构共定义了6个版本，版本号分别为1-6。

CHAP1.2.1 ARM体系的结构版本：
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。
*共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，目前已经不在使用。（2^26 = 64MB）
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。
*程序状态寄存器从原来的R15移动到新的专用寄存器：
Current Program Status Register
*增加了SPSR用于异常中断程序时，保存被中断程序的状态。
Saved Program Status Register
*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,MRS。用于访问CPSR,SPSR寄存器。
*修改了原来版本中的从异常返回的指令。
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。它适合永电池供电的高性能便携式设备。

CHAP1.2.2 ARM体系的变种：
1.T(THUMB)。
2.M(长乘法指令)。
3.E(增强型DSP指令)。

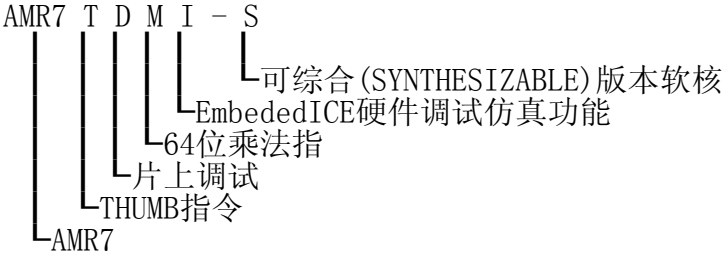
4.J(JAVA加速器JAZELLE)。
5.SIMD(ARM多媒体扩展)。

CHAP1.2.3 ARM/THUMB体系版本的命名格式
表示AMR/THUMB体系版本的字符串是由下面几个部分组成的：



1.ARM字符串。
2.ARM版本号。
3.X表示排除某种功能。

2008-1-26
CHAP1.3 ARM处理器系列
1.ARM7:
ARM7TDMI,ARM7TDMI-S,ARM7EJ-S,ARM720T-4



2.ARM9:
ARM9系列处理器使用了ARM9TDMI内核，其中包含了16位的THUMB指令集。

3.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。

4.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

5.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。

CHAP1.4 ARM处理器模式
7种运行模式：
1.USR(USER).
2.FIQ(FAST INTERRUPT RQUEST)
3.IRQ(INTERRUPUT RQUEST)
4.SVE(SUPERVISOR)
5.ABT(ABORT)
6.UND(UNDEFINED)
7.SYS(SYSTEM)

CHAP1.5 ARM寄存器介绍

37个寄存器：

第一类：31个通用寄存器，包括PC在内。都是32位的。

第二类：6个状态寄存器，都是32位的目前只使用了一部分的位。

user mode	Privileged Mode					
	system	exception				
usr	sys	svc	abt	und	irq	fiq
R0 <----> R7						
R8 <----> R12						R8_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

寄存器的讲解：

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。

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。

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THUMB)

R15(PC) : 1.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。

2.注意：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。
测试的程序：

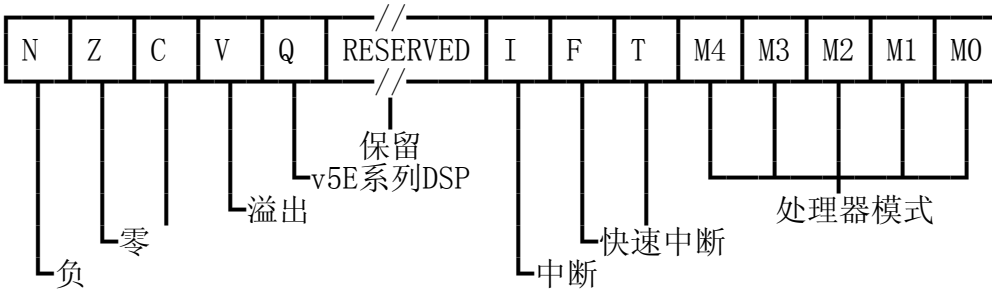
```
        SUB     R1, PC, #4;获取STR指令的地址。
PC-4-->STR      PC, [R0]  ;保存PC + X的值，也就是把偏移值读出来。
          PC---> LDR      R0, [R0]  ;保存偏移值到R0中
PC+4-->SUB      R0, R0, R1;求出偏移值。
          PC+8-->XXX      X, X, X
PC+12->XXX      X, X, X
```

3.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AR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.

4.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T状态。

5.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
的影响。

CPSR :



负零进溢，中快状模。

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，可以吧协处理器产生的条件标志位的
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
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
CPSR中，这种操作主要用于从异常中断程序中返回。

2. CPSR中的【控制位】：

I,F,T,M

M的组合：

M[4:0]	MODE
10000	USR
10001	FIQ
10010	IRQ
10011	SVE
10111	ABT
11011	UND
11111	SYS

CHAP1.6 ARM体系的异常中断

1.ARM中异常中断种类：

优先级	异常	地址	
1	RESET	0x00000000	
2	D_ABT	0x00000010	
3	FIQ	0x0000001C	<----可以直接把FIQ的服务程序放在后面。
4	IRQ	0x00000018	
5	I_ABT	0x0000000C	
6	UND	0x00000004	
6	SWI	0x00000008	

2.ARM处理器对异常中断的响应过程。

伪指令表示：

响应：

R14_<exception_mode> = return link ;保存返回地址

SPSR_<exception_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

返回：

CPSR = SPSR_<exception_mode>

PC = Return link

CHAP1.7 ARM体系中的存储系统

1.ARM体系中的存储空间。

ARM体系使用FLAT模式的地址空间。

可以为2^32(即4GB)个字节。

2.ARM体系中的存储器格式。

*big-endian

30 24	23 16	15 8	7 0	
1	2	3	4	<----数据为:1234

*little-endian

30 24	23 16	15 8	7 0	
4	3	2	1	<----数据为:1234

3.非对齐的存储器访问操作。

*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。

*非对齐的数据预取操作：

*执行的结果不可预知。

*忽略字单元的最低两位值。即Address and 0xfffffffc。
忽略半字单元的最低位值。即Address and 0xfffffffe。
*忽略字、半字单元的低位无效值。由存储器系统完成。
4.指令预取和自修改代码。

/***
/***
第二章 ARM指令分类及其寻址方式

CHAP2.1 ARM指令集概要介绍
1.ARM指令的分类: 6种
*跳转指令
*数据处理指令
*程序状态寄存器传输指令
*Load/Store指令
*协处理器指令
*异常中断产生指令

2.ARM指令的一般编码格式

31-28	27-25	24-21	20	19-16	15-12	11-0
cond	001	opcode	s	Rn	Rd	Shifter_operand

cond 条件
001 固定编码
opcode 操作符
s 影响标志
Rn 第一操作数编码
Rd 目的寄存器编码
Shifter_operand 第二操作数

3.ARM指令的条件码域

条件	助记符	含义	CPSR条件标志位
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	无符号数大于	C=1且Z=0
1001	LS	无符号数小于等于	C=0或Z=1
1010	GE	有符号大于等于	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	有符号数小于等于	Z=1或N !=V
1110	AL	无条件执行	
1111	NV	该指令从不执行	

CHAP2.2 ARM指令寻址方式

1.数据处理指令的操作数的寻址方式。

*立即数方式

*寄存器方式

*寄存器移位方式

2.字及无符号字节的LOAD/STORE指令的寻址方式。

各种类型的LOAD/STORE指令寻址方式由两部分组成。

A部分:基址寄存器。

B部分:地址偏移量。

地址偏移量可以由3种格式:

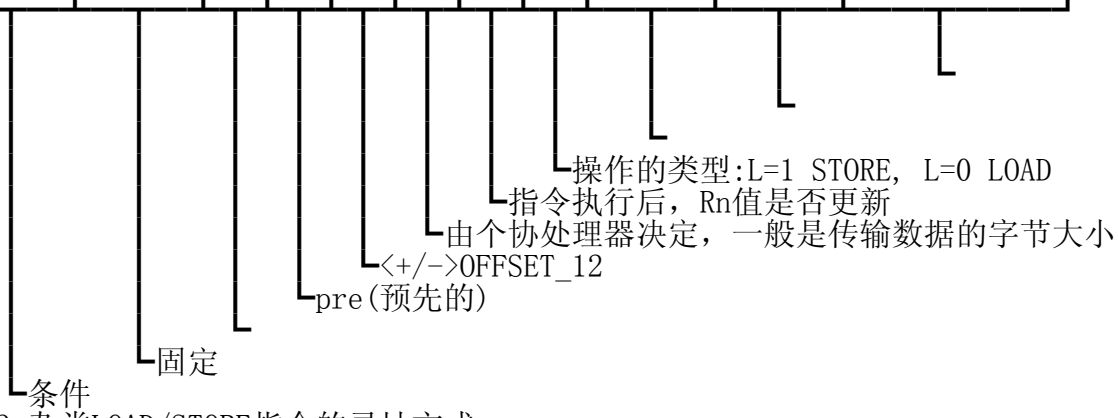
(1).立即数

(2).寄存器

(3).寄存器移位

以下各个位只是一些指令的，可能其他的不一样，不是标准。

31-28	27-26	25	24	23	22	21	20	19-16	15-12	11-0
cond	01	I	P	U	N	W	L	Rn	Rd	Address_mode



3.杂类LOAD/STORE指令的寻址方式。

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

/***/

/***/

第三章 ARM指令集介绍

CAHP3.1 ARM指令集

1.跳转指令。

B及BL指令:

31-28	27-25	24	23-0
cond	101	L	signed_immed24

有L表示保存当前PC寄存器的值于LR中。

跳转的范围大致为：-32MB ~ +32MB

返回：
*BX R14
*MOV PC,R14
*STMFD R13!,{<registers>,R14}
LDMFD R13!,{<registers>,PC}
Signed_immed24的来历:
(1).将PC寄存器的值作为本跳转指令的基地址值。
(2).从目的地址减去基地址形成偏移地址。
(3).当上面的偏移地址超过33554432~33554430时，程序需要做相应的处理。
(4).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]。
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

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。

31-28	27-25	24	23-0
1111	101	H	signed_immed24

返回：
*BX R14
*PUSH{<registers>,R14}
POP{<registers>,PC}
Signed_immed24的来历:
(1).将PC寄存器的值作为本跳转指令的基地址值。
(2).从目的地址减去基地址形成偏移地址。
(3).当上面的偏移地址超过33554432~33554430时，程序需要做相应的处理。
(4).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]。

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

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

2.数据处理指令。
*数据传送指令
MOV,MVN
*算术逻辑运算指令

```
541          AND,EOR,ORR,ADD,SKUB,RSB,ADC,SBC,RSC,BIC
542         *比较指令
543              CMP,TST,TEQ
544
545      ---->MOV指令:
546          指令操作的伪代码:
547          if cond passed then
548             Rd = shifter_operand
549         if S==1 and Rd==R15 then
550            CPSR = SPSR
551                 else if S==1 then
552            N = Rd[31]
553            Z = if Rd==0 then 1
554                else 0
555                    C = shifter_carry_out
556            V = unaffected
557
558          MOV指令完成的功能:
559          *将数据从一个寄存器传送到另外一个寄存器
560          *将一个常数传送到另外一个寄存器中
561          *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。
562          *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。
563          *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。
564
565         ---->MVN指令:
566          本指令的伪代码:
567          if cond passed then
568             Rd = NOT shifter_oprand   <-----取反传送
569         if S==1 and Rd==R15 then
570            CPSR = SPSR
571                 else if S==1 then
572            N = Rd[31]
573            Z = if Rd==0 then 1
574                else 0
575                    C = shifter_carry_out
576            V = unaffected
577              MVN指令完成的功能:
578          *向寄存器中传送一个负数。
579          *生成位掩码。
580          *求一个数的反码。
581
582         ---->ADD指令:
583          本指令的伪代码:
584          if cond passed then
585             Rd = Rn + shifter_oprand
586         if S==1 and Rd==R15 then
587            CPSR = SPSR
588                 else if S==1 then
589            N = Rd[31]
590            Z = if Rd==0 then 1
591                else 0
592                    C = carryfrom(Rn + shifter_oprand)
593            V = overflowfrom(Rn + shifter_oprand)
594              ADD指令完成的功能:
595          *完成两个操作数相加。
596
597         ---->ADC指令:
598          本指令的伪代码:
599          if cond passed then
600             Rd = Rn + shifter_oprand + C
```

```
601          if S==1 and Rd==R15 then
602             CPSR = SPSR
603                  else if S==1 then
604             N = Rd[31]
605             Z = if Rd==0 then 1
606                 else 0
607                     C = carryfrom(Rn + shifter_oprand + C)
608             V = overflowfrom(Rn + shifter_oprand + C)
609               ADC指令完成的功能：
610           *ADC和ADD指令联合使用可以实现两个64位的操作数相加。
611            如：  ADD  R0,R2 ; 具体的放置位置R1R0,R3R2
612                  ADC  R1,R3
613
614          ---->SUB指令：
615           本指令的伪代码：
616           if cond passed then
617              Rd = Rn - shifter_oprand
618          if S==1 and Rd==R15 then
619             CPSR = SPSR
620                  else if S==1 then
621             N = Rd[31]
622             Z = if Rd==0 then 1
623                 else 0
624                     C = NOT borrowfrom(Rn - shifter_oprand)<-----进位标志取反
625             V = overflowfrom(Rn - shifter_oprand)
626               SUB指令完成的功能：
627               *实现两个操作数相减。
628           *SUBS指令和条件跳转指令实现循环控制。
629           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，
630                 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
631             是为了适应SBC指令的操作需要。
632
633  2008-1-27
634          ---->SBC指令：
635           本指令的伪代码：
636           if cond passed then
637              Rd = Rn - shifter_oprand - NOT C<-----进位标志取反
638          if S==1 and Rd==R15 then
639             CPSR = SPSR
640                  else if S==1 then
641             N = Rd[31]
642             Z = if Rd==0 then 1
643                 else 0
644                     C = NOT borrowfrom(Rn - shifter_oprand - NOT C)
645             V = overflowfrom(Rn - shifter_oprand - NOT C)
646               SBC指令完成的功能：
647           *SUB和SBC指令联合使用可以实现两个64位的操作数相减。
648            如：  SUBS  R4,R0,R2 ; 具体源操作数的放置位置 R1R0,R3R2
649                  SBC   R5,R1,R3 ; 结果操作数放置位置 R5R4
650               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。
651                 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
652             ADDS指令中的进位指令正好相反。
653
654          ---->RSB逆向减法指令：
655           指令操作的伪代码：
656           if cond passed then
657              Rd = shifter_oprand - Rn
658          if S==1 and Rd==R15 then
659             CPSR = SPSR
660                  else if S==1 then
```

```
661             N = Rd[31]
662             Z = if Rd==0 then 1
663                 else 0
664                     C = NOT borrowfrom(shifter_oprand - Rn)
665             V = overflowfrom(shifter_oprand - Rn)
666               RSB指令完成的功能：
667              *RSB Rd,Rx,#0 ; Rd = 0 - Rx  <---------逆向相减
668               RSB Rd,Rx,Ry,LSL#n ; Rd = RY*2^n - Rx
669                 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
670                   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。
671
672            ---->RSC带借位逆向减法指令：
673             指令操作的伪代码：
674             if cond passed then
675                Rd = shifter_oprand - Rn - NOT C
676            if S==1 and Rd==R15 then
677               CPSR = SPSR
678                    else if S==1 then
679               N = Rd[31]
680               Z = if Rd==0 then 1
681                   else 0
682                       C = NOT borrowfrom(shifter_oprand - Rn - NOT C)
683               V = overflowfrom(shifter_oprand - Rn - NOT C)
684                 RSB指令完成的功能：
685             下面的指令序列可以求一个64位数值的负数。
686              *RSBS  R4,R2,R0 ; 减数：R3R2,被减数：R1R0
687               RSC   R5,R3,R1 ; 结果：R5R4
688               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
689                   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。
690
691            ---->AND逻辑与操作指令：
692             指令操作的伪代码：
693             if cond passed then
694                Rd = Rn AND shifter_oprand
695            if S==1 and Rd==R15 then
696               CPSR=SPSR
697                    else if S==1 then
698                 N = Rd[31]
699                 Z = if Rd==0 then 1
700                     else 0
701                         C = shifter_carry_out
702                 V = unaffected
703                 AND指令完成的功能：
704              *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。
705
706
707            ---->ORR逻辑或指令：
708             指令的伪代码：
709             if cond passed then
710                Rd = Rn OR shifter_oprand
711            if S==1 and Rd==R15 then
712               CPSR = SPSR
713                    else if S==1 then
714                 N = Rd[31]
715                 Z = if Rd == then 1
716                     else 0
717                         C = shifter_carry_out
718                 V = unaffected
719                 ORR指令完成的g功能：
720              *将某些位置1。要置1的位用1或，保留不变的用0或。
```

```
721
722          ---->EOR逻辑或指令：
723           指令的伪代码：
724           if cond passed then
725              Rd = Rn EOR shifter_oprand
726          if S==1 and Rd==R15 then
727             CPSR = SPSR
728                  else if S==1 then
729               N = Rd[31]
730               Z = if Rd == then 1
731                   else 0
732                       C = shifter_carry_out
733               V = unaffected
734               EOR指令完成的功能：
735           *把某些位取反，要取反的就和1异或，保留的用0异或。
736
737          ---->BIC位清除指令：
738           指令的伪代码：
739           if cond passed then
740              Rd = Rn AND NOT shifter_oprand
741          if S==1 and Rd==R15 then
742             CPSR = SPSR
743                  else if S==1 then
744               N = Rd[31]
745               Z = if Rd == then 1
746                   else 0
747                       C = shifter_carry_out
748               V = unaffected
749               ORR指令完成的功能：
750           *将某些位置0。要置0的位用1，保留不变的用0。
751
752          ---->CMP比较指令：
753           指令的伪代码：
754           if cond passed then
755              alu_out = Rn - shifter_oprand     <----不保存结果
756          if S==1 and Rd==R15 then
757             CPSR = SPSR
758                  else if S==1 then
759               N = alu_out[31]
760               Z = if alu_out == then 1
761                   else 0
762                       C = NOT borrowfrom(Rn - shifter_oprand)
763               V = overflowfrom(Rn - shifter_oprand)
764               ORR指令完成的g功能：
765           *只影响CPSR中的标志位，结果不保存。和SUBS类似。
766
767          ---->CMN比较指令：
768           指令的伪代码：
769           if cond passed then
770              alu_out = Rn + shifter_oprand     <----不保存结果
771          N = alu_out[31]
772          Z = if alu_out == then 1
773              else 0
774                  C = NOT carryfrom(Rn + shifter_oprand)
775          V = overflowfrom(Rn + shifter_oprand)
776               ORR指令完成的功能：
777           *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
778            的结果设置CPSR中相应的条件标志位。
779           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
780                 的影响，与Rn中的值加上shifter_oprand表示的数值对CPSR中条件标
```

```
781             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
782             者结果不同，如：
783                 CMP  Rn, #0 ; C=1
784                 CMN  Rn, #0 ; C=0
785
786           ---->TST位测试指令：
787            指令操作的伪代码：
788            if cond passed then
789               alu_out = Rn AND shifter_oprand
790           N flag = alu_out[31]
791           Z flag = if alu_out==0 then 1
792                    else 0
793                  C = shifter_carry_out
794           V = unaffected
795                TST指令完成的功能：
796                *测试某些位是1还时0。
797
798           ---->TEQ相等测试指令：
799            if cond passed then
800               alu_out = Rn EOR shifter_oprand
801           N = alu_out[31]
802           Z = if alu_out==0 then 1
803               else 0
804                   C = shifter_carry_out
805           V = unaffected
806                TEQ指令完成的功能：
807            *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。
808            *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
809            寄存器中N位为两个操作数符号位异或操作的结果。
810
811
812
813              3.乘法指令
814            *MUL       32位乘法指令
815            *MLA       32位带加数的乘法指令
816            *SMULL     64位有符号数乘法指令
817            *SMLAL     64位带加数的有符号数乘法指令
818            *UMULL     64位无符号数乘法指令
819            *UMLAL     64位带加数的无符号数乘法指令
820
821           ---->MUL指令：
822            if cond passed then
823               Rd = (Rm * Rs)[31:0]
824           if S==1 then
825              N = Rd[31]
826              Z = if Rd==0 then 1
827                  else 0
828                      C = unaffected
829              V = unaffected
830                MUL指令的使用：
831            *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，
832             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。
833                *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。
834             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
835             确定的。
836                *寄存器Rm，Rn，Rd为R15时，指令执行的结果不可预测。
837             MUL    R0,R1,R2 ; R0=R1*R2
838             MULS   R0,R1,R2 ; R0=R1*R2,调试设置CPSR中的N、Z位
839
840           ---->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

---->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。

---->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。

---->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。

       ---->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.杂类的算术指令
    ---->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：
        A.计算操作数规范化(使其最高位为1)时所需要左移的位数。
        B.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

          5.状态寄存器访问指令
        *MRS    Register <--- CPSR
        *MSR    CPSR     <--- Register

       ---->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。

       ---->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。

              6.LOAD/STORE内存访问指令
            *LDR        字加载
            *LDRB       字节加载
            *LDRBT      用户模式的字节数据加载
            *LDRH       半字加载
            *LDRSB      有符号字节数据加载
            *LDRSH      有符号半字数据加载
            *LDRT       用户模式的字数据加载
            *STR        字写入
            *STRB       字节写入
            *STRBT      用户模式的字节写入
            *STRH       半字写入
              *STRT       用户模式的字写入

           ---->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
```

```
1021            PC = value AND 0xfffffffe
1022            T  = value[0]
1023                 else
1024            PC = value AND 0xfffffffc
1025                 else
1026            Rd = value
1027              本指令的使用：
1028          *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
1029          对该数据进行一定的操作。
1030          *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。
1031          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
1032          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
1033          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
1034          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
1035          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
1036          执行在ARM状态下。
1037          指令示例：
1038          *LDR    R0,[R1,#4]          ;R0 = [R1+4]
1039              *LDR    R0,[R1,-#4]         ;R0 = [R1-4]
1040          *LDR    R0,[R1,R2]          ;R0 = [R1+R2]
1041          *LDR    R0,[R1,R2,LSL #2] ;R0 = [R1+R2*4]
1042
1043          *LDR    R0,[R1,#4]!         ;R0 = [R1+4],R1 = R1+4
1044          *LDR    R0,[R1,-#4]         ;R0 = [R1-4],R1 = R1-4
1045
1046          *LDR    R0,[R1],#4          ;R0 = [R1],R1 = R1+4
1047          *LDR    R0,[R1],R2          ;R0 = [R1],R1 = R1+R2
1048          *LDR    R0,[R1],R2,LSL #2 ;R0 = [R1],R1 = R1+R2*4
1049
1050
1051         ---->LDRB指令：
1052          本指令伪代码：
1053          if cond passed then
1054             Rd = Memory[address,1]
1055              本指令的使用：
1056          *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
1057          对该数据进行一定的操作。
1058          *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。
1059          指令示例：
1060          *LDRB    R0,[R2,#3]          ;R0[7:0] = [R2+3],R0[31:8] = 0
1061
1062         ---->LDRBT指令：
1063          本指令伪代码：
1064          if cond passed then
1065             Rd = Memory[address,1]
1066              本指令的使用：
1067          *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
1068          对该数据进行一定的操作。
1069          *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
1070          用户模式下的内存访问操作。
1071          指令示例：（注意模式）
1072          *LDRBT    R0,[R2,#3]          ;R0[7:0] = [R2+3],R0[31:8] = 0
1073
1074         ---->LDRH指令：
1075          本指令伪代码：
1076          if cond passed then
1077             if address[0] = 0
1078            data = Memory[address,2]
1079                 else
1080            data = UNPREDICTABLE
```

```
1081                Rd = data
1082              本指令的使用：
1083            *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
1084            对该数据进行一定的操作。
1085            *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。
1086            指令示例：
1087            *LDRH    R0,[R2,#3]          ;R0[15:0] = [R2+3],R0[31:16] = 0
1088
1089          ---->LDRSB指令：
1090            本指令伪代码：
1091            if cond passed then
1092               data = Memory[address,1]
1093          Rd = SignExtend(data)
1094               本指令的使用：
1095               *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
1096            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。
1097            *PC作为目标寄存器时，指令可以实现程序跳转的功能。
1098            指令示例：
1099            *LDRSB    R0,[R2,#3]          ;R0[7:0] = [R2+3],R0[31:8] = 字节符号
1100
1101          ---->LDRSH指令：
1102            本指令伪代码：
1103            if cond passed then
1104               if address[0] = 0
1105                  data = Memory[address,2]
1106                   else
1107              data = UNPREDICTABLE
1108              Rd = SignExtend(data)
1109                本指令的使用：
1110                *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
1111            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。
1112            *PC作为目标寄存器时，指令可以实现程序跳转的功能。
1113            指令示例：
1114            *LDRSH    R0,[R2,#3]          ;R0[15:0] = [R2+3],R0[31:16] = 字节符号
1115
1116          ---->LDRT指令：
1117            本指令伪代码：
1118            if cond passed then
1119               if address[1:0] == 0b00
1120              Rd = Memory[address,4]
1121               else if address[1:0] == 0b01
1122              Rd = Memory[address,4] rotate_right 8
1123               else if address[1:0] == 0b10
1124              Rd = Memory[address,4] rotate_right 16
1125               else
1126              Rd = Memory[address,4] rotate_right 24
1127                本指令的使用：
1128                *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
1129            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
1130            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
1131            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
1132            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
1133            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)
1134            *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
1135            模式下的内存操作。
1136            指令示例：
1137                ?
1138
1139          ---->STR指令
1140            本指令的伪代码：
```

```
1141            if cond passed then
1142               Memory[address,4] = Rd
1143                本指令的使用：
1144            *用于将一个32位的字数据写入到指令指定的内存单元。
1145            指令示例：
1146            STR    R0,[R1,#0X100] ; R0 ---> [R1+0X100]
1147                STR    R0,[R1],#8     ; R0 ---> [R1], R1 = R1+8
1148
1149           ---->STRB指令
1150            本指令的伪代码：
1151            if cond passed then
1152               Memory[address,1] = Rd[7:0]
1153                本指令的使用：
1154            *用于将一个32位的字数据写入到指令指定的内存单元。
1155            指令示例：
1156            STR    R0,[R1,#0X100] ; R0 ---> [R1+0X100]单元的低8位
1157                STR    R0,[R1],#8     ; R0 ---> [R1]单元低8位, R1 = R1+8
1158
1159           ---->STRH指令
1160            本指令的伪代码：
1161            if cond passed then
1162               if address[0] = 0
1163              data = Rd[15:0]
1164                   else
1165              data = UNPREDICTABLE
1166               Memory[address,2] = data
1167                本指令的使用：
1168            *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。
1169            指令示例：
1170            STR    R0,[R1,#0X100] ; R0 ---> [R1+0X100]
1171  @              STR    R0,[R1],#8     ; R0 ---> [R1], R1 = R1+8
1172
1173           ---->STRT指令
1174            本指令的伪代码：
1175            if cond passed then
1176               Memory[address,4] = Rd
1177                本指令的使用：
1178            *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
1179            的权限内存访问操作。
1180            指令示例：
1181            STR    R0,[R1,#0X100] ; R0 ---> [R1+0X100]
1182                STR    R0,[R1],#8     ; R0 ---> [R1], R1 = R1+8
1183
1184           7.批量LOAD/STORE内存访问指令
1185          *LDM(1)    批量内存字数据读取指令
1186          *LDM(2)    用户模式的批量内存字数据
1187          *LDM(3)    带状态寄存器的批量内存字数据读取指令
1188          *STM(1)    批量内存字数据写入指令
1189          *STM(2)    用户模式的批量内存字数据写入指令
1190
1191           ---->LDM(1)指令
1192            本指令的格式：
1193            LDM{<cond>}<addressing_mode><Rn>{!},<registers>
1194            本指令的伪代码：
1195            if cond passed then
1196               address = start_address
1197           for i = 0 to 14
1198               if register_list[i] == 1 then
1199                  Ri = Memory[address,4]
1200                  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

----->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。

----->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。

2008-1-28
```

```
1261          ---->STM(1)指令
1262               本指令格式：
1263               STM{<cond>}<address_mode><Rn>{!},<registers>
1264               本指令的伪代码：
1265               if cond passed then
1266                  address = start_address
1267                  for i = 0 to 15
1268                      if register_list[i] == 1
1269                         Memory[address,4] = Ri
1270                         address = address + 4
1271                  assert end_address == address - 4
1272               本指令的使用：
1273               *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
1274               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。
1275               *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
1276               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
1277               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
1278               可预知的后果。
1279
1280          ---->STM(2)指令
1281               本指令格式：
1282               STM{<cond>}<address_mode><Rn>,<registers>^
1283               本指令的伪代码：
1284               if cond passed then
1285                  address = start_address
1286                  for i = 0 to 15
1287                      if register_list[i] == 1
1288                         Memory[address,4] = Ri_usr
1289                         address = address + 4
1290                  assert end_address == address - 4
1291               本指令的使用：
1292               *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
1293               用该指令，会产生不可预知的结果。
1294               *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。
1295               *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，
1296               而不是用户模式对应的寄存器。
1297          8.信号量操作指令
1298            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，
1299            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
1300            信号量的操作。
1301            *SWP      交换指令
1302            *SWPB     字节交换指令
1303
1304          ---->SWP指令
1305               本指令格式：
1306               SWP{<cond>} <Rd>,<Rm>,[<Rn>]
1307               本指令的伪代码：
1308               if cond passed then
1309                  if Rn[1:0] == 0b00 then
1310                     temp = Memory[Rn,4]
1311                  else if Rn[1:0] == 0b01 then
1312                     temp = Memory[Rn,4] rotate_right 8
1313                  else if Rn[1:0] == 0b10 then
1314                     temp = Memory[Rn,4] rotate_right 16
1315                  else
1316                     temp = Memory[Rn,4] rotate_right 24
1317                  Memory[Rn,4] = Rm
1318                  Rd = temp
1319              本指令的使用：
1320              *主要用于实现信号量操作。
```

```
1321              本指令示例：
1322              *SWP    R1,R2,[R3] ; R1 <-- [R3] then [R3]<--R2
1323              *SWP    R1,R1,[R2] ; R1 <-- [R2] then [R2]<--R1
1324
1325          ---->SWPB指令
1326               本指令格式：
1327               SWP{<cond>} <Rd>,<Rm>,[<Rn>]
1328               本指令的伪代码：
1329               if cond passed then
1330                  if Rn[1:0] == 0b00 then
1331                     temp = Memory[Rn,4]
1332                  else if Rn[1:0] == 0b01 then
1333                     temp = Memory[Rn,4] rotate_right 8
1334                  else if Rn[1:0] == 0b10 then
1335                     temp = Memory[Rn,4] rotate_right 16
1336                  else
1337                     temp = Memory[Rn,4] rotate_right 24
1338                  Memory[Rn,4] = Rm
1339                  Rd = temp
1340              本指令的使用：
1341              *主要用于实现信号量操作。
1342              本指令示例：
1343              *SWPB  R1,R2,[R3] ; R1[7:0] <-- [R3],R1[31:8]=0 then [R3]<--R2[7:0]
1344              *SWPB  R1,R1,[R2] ; R1[7:0] <-- [R2],R1[31:8]=0 then [R2]<--R1[7:0]
1345
1346          9.异常中断产生指令
1347            *SWI    软中断指令
1348            *BKPT   断点中断指令
1349
1350          ---->SWI指令
1351               本指令格式：
1352               SWI {<cond>} <Immed_24>
1353               本指令伪代码：
1354               if cond passed then
1355                  R14_svc = address of next instruction after the SWI instruction
1356                  SPSR_svc = CPSR
1357                  CPSR[4:0] = 0b10011 /*Enter the Supervisor mode*/
1358                  CPSR[5]   = 0 /*ARM state*/
1359                  CPSR[7]   = 1/*Disable normal interruputs*/
1360               if high vectors configured then
1361                  PC = 0xffff0008
1362               else
1363                  PC = 0x00000008
1364               本指令的使用：
1365               *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。
1366               *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，
1367               参赛通过其他的通用寄存器传递。
1368
1369          ---->BKPT指令
1370               本指令格式：
1371               BKPT <Immed_16>
1372               本指令伪代码：
1373               if (not overridden by debug hardware) then
1374                  R14_abt = address of BKPT instruction + 4
1375                  SPSR_svc = CPSR
1376                  CPSR[4:0] = 0b10011 /*Enter the Supervisor mode*/
1377                  CPSR[5]   = 0 /*ARM state*/
1378                  CPSR[7]   = 1/*Disable normal interruputs*/
1379               if high vectors configured then
1380                  PC = 0xffff000c
```

```
1381              else
1382                 PC = 0x0000000c
1383              本指令的使用：
1384              *本指令主要供软件测试程序使用。
1385
1386           10.ARM协处理指令
1387              *CDP    协处理器数据操作指令
1388              *LDC    协处理器数据读取指令
1389              *STC    协处理器数据写入指令
1390              *MCR    ARM寄存器到协处理器的数据传送指令
1391              *MRC    协处理器寄存器到ARM寄存器的传送指令
1392
1393           ---->CDP指令
1394              指令语法格式：
1395              CDP{<cond>} <coproc>,<CRd>,<CRn>,<CRm>,<opcode_2>
1396              CDP{<cond>} <coproc>,<opcode_1>,<CRd>,<CRn>,<opcode_2>
1397              coproc : 协处理器的编号
1398              opcode_1: 协处理器将执行的操作码
1399              CRd : 协处理器的目的寄存器
1400              CRn : 协处理器第一操作数存放寄存器
1401              CRm : 协处理器第二操作数存放寄存器
1402              本指令伪代码：
1403              if cond passed then
1404                 Coprocessor[cp_num] -dependent operation
1405              本指令的使用：
1406              *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。
1407              本指令示例：
1408              CDP p5,2,c12,c10,c3,4 ;P5号协处理器、操作码1为2，操作码2为4、目的
1409                                    ;协处理器寄存器C12，源操作数协处理器寄存器
1410                                    ;为C10和C3
1411
1412           ---->LDC指令
1413              指令语法格式：
1414              LDC{<cond>}{L} <coproc>,<CRd>,<addressing_mode>
1415              LDC2{L} <coproc>,<CRd>,<addressing_mode>   <---无条件执行
1416              coproc : 协处理器的编号
1417              opcode_1: 协处理器将执行的操作码
1418              CRd : 协处理器的目的寄存器
1419              本指令伪代码：
1420              if cond passed then
1421                 address = start_address
1422                 load Memory[address,4] for coprocessor[cp_num]
1423                 while (NotFinished(Coprocessor[cp_num]))
1424                       address = address + 4
1425                       load Memory[address,4] for coprocessor[cp_num]
1426                       assert address == end_address
1427              本指令的使用：
1428              *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。
1429              CDP p6,CR3,[R2,#4] ;将ARM处理器寄存器[R2+4]指向的内存单元数据传输到
1430                                 ;P6号协处理器寄存器CR3中。
1431
1432           ---->STC指令
1433              指令语法格式：
1434              STC{<cond>}{L} <coproc>,<CRd>,<addressing_mode>
1435              STC2{L} <coproc>,<CRd>,<addressing_mode>   <---无条件执行
1436              coproc : 协处理器的编号
1437              opcode_1: 协处理器将执行的操作码
1438              CRd : 协处理器的目的寄存器
1439              本指令伪代码：
1440              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。
            STC p8,CR8,[R2,#4] ;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。

        ---->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

        ---->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


CHAP3.2    一些基本ARM指令功能段
        1.逻辑运算指令的应用。
          1.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
          2.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
          3.64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
```

```
1501            R1R0 = R1R0 - R3R2
1502            SUBS    R0, R0, R2 ;
1503            SBC     R1, R1, R3 ;
1504            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，
1505            V标志可能有错。
1506            CMP     R1, R3 ;比较高32位
1507            CMPEQ   R0, R2 ;如果高32位相等，比较低32位
1508
1509          4.转换内存中数据存储方式的指令段。
1510            以下指令实现小端数据存储格式到大端数据格式的转换。
1511            执行前： R0 = ABCD          执行后： R0 = DCBA
1512            EOR     R1, R0, R0, ROR #16 ; R1 = A^C,B^D,C^A,D^B
1513            BIC     R1, R1, #0x00FF0000 ; R1 = A^C,0,C^A,D^B
1514            MOV     R0, R0, ROR #8      ; R0 = D,A,B,C
1515            EOR     R0, R0, R1, LSL #8  ; R0 = D,C,B,A
1516            下面的指令用于转换大量的字数据的存储方式。
1517            指令执行前需要转换的数据放在R0中，其存储方式为： R0 = ABCD
1518            指令执行后R0中存储方式为：R0 = DCBA
1519            MOV     R2, #0XFF ;
1520            MOV     R2, R2, #0X00FF0000 ;
1521            重复下面的指令段，实现数据存放方式的转换
1522            AND     R1, R2, R0          ;R1 = 0 B 0 D
1523            AND     R0, R2, R0, ROR #24 ;R0 = 0 C 0 A
1524            ORR     R0, R0, R1, ROR #8  ;R0 = D C B A
1525
1526        2.跳转指令的应用。
1527          1.子程序调用。
1528            ...
1529            BL      FUNCTION
1530            ...
1531            FUNCTION
1532            ...
1533            ...
1534            MOV     PC, LR
1535
1536          2.条件执行。
1537            求两个数的公约数。
1538            GCD     CMP     R0, R1
1539                    SUBGT   R0, R0, R1
1540                    SUBLT   R1, R1, R0
1541                    BNE     GCD
1542                    MOV     PC, LR
1543
1544          3.条件判断语句。
1545            CMP     R0, #0
1546            CMPNE   R1, #1
1547            ADDEQ   R2, R3, R4
1548
1549          4.循环语句。
1550                    MOV     R0, #LOOPCOUNT
1551            LOOP    ...
1552                    SUBS    R0, R0, #1
1553                    BNE     LOOP
1554                    ...
1555          5.多路分支程序语句。
1556                    CMP     R0, #MAXINDEX   ;判断跳转索引是否在范围内
1557                    ADDLO   PC, PC, R0, LSL #ROUTINESIZELOG2
1558
1559                    Index0handler
1560                    ...
```

```
                ...
                Index1handler
                ...
                ...
                Index2handler
                ...
```

2.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.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}
```

5.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，表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]
```

6.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
```

```
1681            if high vectors configured then
1682               PC = 0XFFFF0008
1683            else
1684               PC = 0X00000008
1685
1686            SWI处理的程序的基本框架：
1687               发生SWI中断
1688               跳转到0X00000008处执行跳转
1689               提取SWI指令中的立即数确定响应的中断号
1690            基本程序：
1691            SWIHandler    STMFD  SP!, {R0-R3, R12, LR}     ;资源保护
1692                          MRS    R0, SPSR                  ;是否位ARM状态
1693                          TST    R0, #0X20
1694                          LDRNEH R0, [LR, #-2]             ;THUMB状态
1695                          BICNE  R0, R0, #0XFF00
1696                          LDREQ  R0, [LR, #-4]             ;ARM状态
1697                          BICEQ  R0, R0, #0XFF000000
1698                          CMP    R0, #maxSWI
1699                          LDRLS  PC, [PC, R0, LSL #2]      ;取得服务地址
1700                          B      SWIOutOfRange
1701
1702            SWITABLE      DCD    DO_SWI_0       ;SWI 0号服务程序
1703                          DCD    Do_SWI_1
1704                          ...
1705
1706
1707
1708
1709         2.IRQ中断处理程序示例。
1710            ARM响应IRQ中断请求完成以下工作：
1711            R14_irq = 当前指令地址 + 8
1712            SPSR_irq = CPSR
1713            CPSR[4:0] = 0B10010
1714            CPSR[5] = 0 /*ARM状态*/
1715            CPSR[7] = 1 /*IRQ disabled*/
1716            if high vectors configured then
1717               PC = 0XFFFF0018
1718            else
1719               PC = 0X00000018
1720
1721            IRQ中断处理程序的基本框架：
1722                          SUB    R14, R14, #4
1723                          STMFD  R13!, {R12, R14}
1724                          MRS    R12, SPSR
1725                          STMFD  R13!, {R12}
1726                          MOV    R12, #IntBase
1727                          LDR    R12, [R12, #IntLevel]
1728
1729                          MRS    R14, CPSR
1730                          BIC    R14, R14, #0X80
1731                          MSR    CPSR_c, R14
1732
1733                          LDR    PC, [PC, R12, LSL #2]
1734
1735                          DCD    PRIORITY0HANDLER
1736                          DCD    PRIORITY1HANDLER
1737                          ...
1738
1739            PRIORITY0HANDLER
1740                          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


/*****************************************************************************/
/*****************************************************************************/
第四章    ARM汇编语言程序设计

CHAP4.1   伪操作(derective)
        1.符号定义伪操作。
          *GBLA,GBLL及GBLS     声明全局变量
          *LCLA,LCLL及LCLS     声明局部变量
          *SETA,SETL及SETS     给变量赋值
          *RLIST    为通用寄存器列表定义名称
          *CN       为协处理器的寄存器定义名称
          *CP       为协处理器定义名称
          *DN及SN   为VFP的寄存器定义名称
          *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}
            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
            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
            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
                  name SN expr
            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
            expr为浮点寄存器的编号，数值范围为0~7。
```

2.数据定义伪操作。

*LTORG	声明一个数据缓冲池(literal pool)的开始。
*MAP	定义一个结构话的内存表(storage map)的首地址。
*FIELD	第一结构化的内存表中的一个数据域(field)。
*SPACE	分配一块内存单元，并用0初始化。
*DCB	分配一段字节的内存单元，并用指定的数据初始化。
*DCD/DCDU	分配一段字的内存单元，用指定的数据初始化。
*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。
*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。
*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。
*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。
*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。
*DCW/DCWU	分配一段半字的内存单元，并用指定的数据初始化。
*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。^时MAP的同义词。

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              4096----->
                constb    FIELD  4
                x         FIELD  8              5000----->
                y         FIELD  8
                string    FIELD  256            5004----->
                                                5012----->
                                                5020----->
                                                5276----->

                LDR   R6, consta
               /*************************************************************************/
                <2>. 相对地址内存表
                MAP 0
                consta    FIELD  4                 0----->
                constb    FIELD  4
                x         FIELD  8                 4----->
                y         FIELD  8
                string    FIELD  256               8----->
                                                  16----->
                                                  24----->
                MOV    R9, #4096                280----->
                LDR    R5,[R9,consta]
               /*************************************************************************/
                <3>. 相对地址内存表
                MAP 0, R9
                consta    FIELD  4                 R9----->
                constb    FIELD  4
                x         FIELD  8              R9+4----->
                y         FIELD  8
                string    FIELD  256            R9+8----->
                                               R9+16----->
                                               R9+24----->
                                              R9+280----->
                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。


2008-1-29
               <5>.FIELD伪操作数为0。
```

consta	4 Bytes
constb	4 Bytes
x	8 Bytes
y	8 Bytes
string	256 B

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 bytes string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。

```
1981            语法格式：
1982            {lable} DCFS(U) fpliteral{,fpliteral}...
1983
1984          (10).DCI
1985            *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。
1986            *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。
1987            *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
1988            于通过宏指令来定义处理器指令系统不支持的指令。
1989            语法格式：
1990            {lable} DCFS(U) expr{,expr}...
1991            示例：
1992            MACRO
1993            newinst    $Rd, $Rm
1994            DCI        0xe16f0f10 :OR: ($Rd :SHL: 12) :OR: $Rm ;这里放的是MEND
1995
1996          (11).DCQ/DCQU
1997            *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。
1998            *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
1999            入多达3个填补字节以保证字对齐。
2000            语法格式：
2001            {lable} DCQ(U) {-}literal{,{-}literal}...
2002            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
2003            上"-"时literal的取值范围是：-2^64~-1。
2004
2005          (12).DCW/DCWU
2006             *分配一段半字内存单元(半字对齐)，并用指定值初始化。
2007             *DCW和DCWU的区别在于DCWU分配的内存单元不严格半字对齐。
2008             语法格式：
2009             {lable} DCW(U) expr{, expr}
2010
2011          3.汇编控制伪操作
2012            *IF,ELSE, ENDIF
2013            *WHILE, WEND
2014            *MACRO, MEND
2015            *MEXIT
2016
2017            (1).IF,ELSE,ENDIF
2018               *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。
2019               *"["是IF的同义词，"]"是ENDIF的同义词，"|"是ELSE的同义词。
2020               *IF,ELSE,ENDIF可以嵌套使用。
2021               语法格式：
2022                IF logical expression
2023                   instructions or derectives
2024                {ELSE                              ;可选项
2025                   instructions or derectives
2026                }
2027                ENDIF
2028
2029             (2).WHILE,WEND
2030               *能够根据条件汇编一段相同或几乎相同的源代码。
2031               *WHILE,WEND可以嵌套使用。
2032               语法格式：
2033               WHILE logical expression
2034                     instructions or derectives
2035               WEND
2036               示例：
2037               count SETA 1
2038               WHILE count <= 4          ;循环控制
2039               count SETA count + 1
2040               ;code
```

```
2041            WEND
2042
2043        (3).MACRO,MEND
2044           *宏定义，替换展开。
2045           *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。
2046           *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。
2047           语法格式：
2048           MACRO
2049           {$lable} macroname {$parameter{,parameter}...}
2050                    ;code
2051                    ...
2052                    ;code
2053           MEND
2054           示例：
2055           MACRO
2056           $lable   xmac  $p1, $p2
2057                    ;code...
2058                    ...
2059           $lable.loop1
2060                    ;code...
2061                    BGE  lable.loop1
2062                    ;code...
2063           $lable.loop2
2064                    BL   $p1
2065                    BGT  $label.loop2
2066                    ;code...
2067                    ADR  $p2
2068                    ;code...
2069           MEND
2070
2071        (4).MEXIT
2072           *从宏中跳出
2073           示例：
2074           MACRO
2075           $abc    macroabc  $param1, $param2
2076           ;code
2077           WHILE   condition1
2078           ;code
2079               IF condition2
2080               ;code
2081               MEXIT          <----直接退出宏
2082               ELSE
2083               ;code
2084               ENDIF
2085           WEND
2086
2087       4.栈中数据帧描述伪操作
2088        *ASSERT
2089        *INFO
2090        *OPT
2091        *TTL,SUBT
2092
2093        (1).ASSERT
2094           *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ASSERTf
2095           伪操作将报告该错误信息。
2096           *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
2097           报告错误类型，并终止汇编。
2098           语法格式:
2099           ASSERT   logical expression
2100           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-expression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。

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
```

6.其他的伪操作
*ALIGN
*AREA
*CODE16,CODE32
*END
*ENTRY
*EQU
*EXPORT,GLOBAL
*EXTERN
*GET,INCLUDE
*IMPORT
*INCBIN
*KEEP
*NOFP
*REQUIRE
*REQUIRE8,PRESERVE8
*RN
*ROUT

(1).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。
(2).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
(3).AREA
*定义一个代码段和数据段。
语法格式：

```
2221            AREA   sectionname {,attr}{,attr}...
2222            其中：
2223                 *sectionname为所定义的代码段和数据段的名称。如果该名称是
2224                 以数字开头，则该名称必须用"|"括起来。如：|7wolfs|。还有一些
2225                 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
2226                 或者是与C语言库相关的代码段。
2227                 *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
2228                 号隔开。下面列举所有可能的属性：
2229                   ->ALIGN = expression。默认的情况下，ELF的代码段和数据段是4字节
2230                   对齐的。expression可以取0~31的数值，相应的对齐方式
2231                   为(2^expression)字节对齐。如 ALIGN 3 就是以2^3(即8)字节对齐。
2232
2233                   ->ASSOC = section。指定本段相关的ELF段。任何时候连接section段
2234                   也必须包含sectionname段。
2235                   ->CODE定义代码段。默认的属性为READONLY。
2236                   ->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
2237                   件中，同名的COMDEF段必须相同。
2238                   ->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
2239                   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
2240                   接器为其分配合适的尺寸。
2241                   ->DATA定义数据段。默认属性为READWRITE
2242                   ->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
2243                   写入内存单元。或者将整个内存单元初始化为0。
2244                   ->READONLY指定本段为只读。代码段的默认属性。
2245                   ->READWRITE指定本段为可读可写，数据段的默认属性。
2246                 *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，
2247                 这时这些同名的段被放在同一个ELF段中。
2248                 *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
2249                 包含一个段。
2250
2251          (4).ENTRY
2252             *指定程序的入口点。
2253             *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
2254             有一个(或没有)ENTRY。
2255
2256          (5).END
2257             *告诉编译器已经到达源程序的结尾。
2258             *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。
2259
2260          (6).ALIGN
2261             *通过添加补丁字节使当前位置满足一定的对齐方式。
2262             语法格式：
2263             ALIGN {expr{, offset}}
2264             其中：
2265                  ->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
2266                    果伪操作没有指定expr，则当前位置对齐到下一个字边界处。
2267                  ->offset为数字表达式。当前位置对齐到：offset + n*expr。
2268              *   ->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
2269                    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
2270                    标号是字对齐的。
2271              *   ->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
2272                    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
2273                    的性能优势。
2274              *   ->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
2275                    令分配的内存单元前要使用ALIGN 8实现8字节对齐方式。
2276              *   ->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
2277                    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
2278                    操作来调整对齐方式。
2279             示例：
2280             <1>.在AREA伪操作中的ALIGN与ALIGN伪操作中的expr含义是不同的。
```

```
2281                AREA cacheable,  CODE,  ALIGN = 3   <-----2^3 = 8
2282                subrout1  ;code
2283                ;code
2284                MOV PC, LR
2285                ALIGN 8         <----------------------指定以下的指令为8字节对齐。
2286                subrout2  ;code
2287
2288             <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。
2289                AREA OffsetExample,  CODE
2290                DCB  1
2291                ALIGN  4,3
2292                DCB  1
2293
2294             <3>.通过ALIGN伪操作使程序中地址标号字对齐。
2295                AREA Example, CODE, READONLY
2296                start LDR  r6, =label1
2297                ;code
2298                      MOV PC, LR
2299                label1  DCB   1    <----------本伪操作破坏了字对齐
2300                ALIGN              <----------重新使数据对齐
2301                subroutine1
2302                       MOV  R5, #0X05
2303
2304          (7).EXPORT,GLOBAL
2305             *声明一个符号可以被其他文件引用。GLOBAL使EXPORT的同义词。
2306             语法格式:
2307             EXPORT    symbol{WEAK}
2308             其中:
2309                  symbol 为声明的符号的名称。它使区分大小写的。
2310                  [WEAK] 选项声明其他的同名符号优先于本符号被引用。
2311
2312          (8).IMPORT
2313             *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
2314             中定义的，在本源文件中可以引用该符号。
2315             *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
2316             表中。
2317             *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
2318             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
2319             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:
2320             -->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
2321                B，BL指令相当于一条NOP指令。
2322             -->其他情况下该符号被设置为0。
2323             语法格式:
2324             IMPORT    symbol{WEAK}
2325             其中:
2326                  symbol 为声明的符号的名称。它使区分大小写的。
2327                  [WEAK] 指定这个选项后，如果symbol在所有的源文件都没有被定义，
2328                         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
2329                         没有被INCLUDE进来的库中去查找该符号。
2330
2331          (9).EXTERN
2332             *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
2333             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，
2334             该符号都将不会加入到本源文件的符号表中。
2335             语法格式:
2336             *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
2337             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
2338             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:
2339             -->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
2340                B，BL指令相当于一条NOP指令。
```

-->其他情况下该符号被设置为0。
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。

(10).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汇编处理。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
```

(11).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
*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
```

(12).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：
-->被输出的符号。
-->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。
```

(13).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。
(14).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：

```
2401                REQUIRE  label
2402
2403            (15).REQUIRE8/PRESERVE8
2404                *REQUIRE8伪操作指示当前代码中要求数据栈8字节对齐。
2405                *PRESERVE8伪操作指示当前代码中数据栈8字节对齐。
2406                *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
2407                在数据栈中传送数据时，要求数据栈是8字节对齐的。
2408                *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。
2409
2410            (16).RN
2411                *为一个特定的寄存器定义名称。
2412                语法格式：
2413                name    RN  expr
2414
2415            (17).ROUT
2416                *定义局部变量的作用范围。
2417                *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
2418                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
2419                个段中)伪操作之间。
2420
2421
2422
2423  CHAP4.2     ARM汇编语言伪指令
2424                *ADR          小范围的地址读取伪指令
2425                *ADRL         中等范围的地址读取伪指令
2426                *LDR          大范围的地址读取伪指令
2427                *NOP          空操作伪指令
2428
2429            1.ADR
2430             *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。
2431             *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，
2432             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
2433             令实现ADR伪指令的功能，编译器将报告错误。
2434             *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
2435             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
2436             在同一个代码段中。
2437             语法格式：
2438             ADR{cond} register, expr
2439             其中：
2440                  expr为基于PC或者基于寄存器的地址表达式，取值范围如下：
2441                  ->当地址值不式字对齐，其范围为：-255 - +255。
2442                  ->当地址值是字对齐时，其范围为：-1020 - +1020。
2443                  ->当地址值是16字对齐时，其范围将更大。
2444
2445            2.ADRL
2446             *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
2447             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。
2448             *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
2449             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。
2450             语法格式：
2451             ADRL{cond} register, expr
2452             其中：
2453                  expr为基于PC或者基于寄存器的地址表达式，取值范围如下：
2454                  ->当地址值不式字对齐，其范围为：-64KB - +64KB。
2455                  ->当地址值是字对齐时，其范围为：-256KB - +256KB。
2456                  ->当地址值是16字对齐时，其范围将更大。
2457             示例：
2458             start    MOV   R0, #10
2459                      ADRL  R4, start+60000     ;编译器替换成下面两条指令。
2460
```

```
                    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 ;汇报后得到  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。

CHAP4.3 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。

1.ARM汇编语言中的符号
*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>程序中的符号不要于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||，这时双竖线并不时符号的组成部分。

(1).变量
*数字变量、逻辑变量、串变量。
(2).数字常量
*32为整数。
(3).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$$。

```
2521                *对于数字变量来说，如果该变量前面有$字符，在汇编时编译将该数字
2522                变量的数字转换成十六进制的串，然后用该十六进制串取代$字符后的变量。
2523                *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
2524                将该逻辑变量替换成它的取值(T或者F)。
2525                示例：
2526                      GBLS    STR1
2527                      GBLS    B
2528                      GBLA    NUM1
2529                NUM1  SETA    14
2530                B     SETB    "CHANGED"
2531  ? ? ? ?       STR1  SETS    "ABC$$B$NUM1"     ;汇编后得到：  ABCB
2532                *通常情况下，包含在两"|"之间的"$"并不表示变量替换。如果双竖线在
2533                双引号内，则将进行变量替换。
2534                *使用"."来表示变量名称的结束。
2535                示例：
2536                      GBLS    STR1
2537                      GBLS    STR2
2538                STR1  SETS    "AAA"
2539                STR2  SETS    "bbb$STR1.CCC"     ;汇编后得到：  bbbAAACCC
2540
2541             (4).标号
2542                *基于PC的标号
2543                *基于寄存器的标号
2544                *绝对地址标号
2545
2546             (5).局部标号
2547                *主要有两部分组成：开头时一个0~99之间的数字。后面紧跟一个通常表示
2548                该局部变量作用范围的符号。
2549                *局部标号的作用范围通常为当前段，也可用伪操作ROUT来定义局部变量的
2550                作用范围。
2551                语法格式：
2552                N{routname}
2553                *N为0~99之间的数字。
2554                *routname为符号，通常为该变量作用范围的名称(用ROUT伪操作定义的)。
2555                局部变量引用的语法格式如下：
2556                %{F|B}{A|T} N{routname}
2557                其中：
2558                      routname为当前作用范围的名称(用ROUT伪操作定义的)。
2559                      %表示引用操作。
2560                      F指示编译器只向前搜索。
2561                      B指示编译器只向后搜索。
2562                      A指示编译器搜索宏的所有嵌套层次。
2563                      T指示编译器搜索宏的当前层次。
2564                      *如果F和B都没有指定，编译器先向前搜索，再向后搜索。
2565                      *如果F和T都没有指定，编译器搜索所有从当前层次到宏的最高层次，
2566                      比当前层次低的层次不再搜索。
2567                      *如果指定了routname，编译器向前搜索最近的ROUT伪操作，若routname
2568                      与该ROUT伪操作定义的名称不匹配，编译器报错，汇编失败。
2569
2570          2.ARM汇编语言中的表达式
2571           *括号内的表达式优先级最高。
2572           *各种操作符有一定的优先级。
2573           *相邻的单目操作符的执行顺序为由右到左，单目运算符优先级高于其他操作。
2574           *优先级相同的双目运算符执行顺序为由左到右。
2575
2576
2577  2008-1-30
2578             (1).字符串表达式
2579                *字符串：由包含在双引号内的一系列字符组成。字符串的长度受到ARM汇编
2580                      语言语句长度的限制。"$$"表示"$",""表示"。
```

```
2581                示例：
2582                     STR1 SETS "this string contains only one"" double quote"
2583                     STR2 SETS "this string contains only one $$ dollar symbol"
2584
2585           *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
2586                        表达式相同
2587
2588           *操作符：  <1>.LEN
2589                        *返回字符串的长度。
2590                        语法格式：
2591                          :LEN: A
2592
2593                      <2>.CHR
2594                        *可以将0~255之间的整数作为含一个ASCII字符的字符串。当
2595                        有些ASCII字符不方便放在字符串中时，可以使用CHR将其放在
2596                        字符串表达式中。
2597                        语法格式：
2598                          :CHR: A    <----A为某一字符的ASCII值
2599
2600                      <3>.STR
2601                        *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
2602                        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
2603                        而言，STR将其转换成字符串T或者F。
2604                        语法格式：
2605                          :STR: A    <----A为数字量或逻辑表达式。
2606
2607                      <4>.LEFT
2608                        *返回一个字符串最左端一定长度的字符串。
2609                        语法格式：
2610                          A :LEFT: B    <----A为字符串，B为返回长度
2611
2612                      <5>.RIGHT
2613                        *返回一个字符串最右端一定长度的字符串：
2614                          A :RIGHT: B  <----A为字符串，B为返回长度
2615
2616                      <6>.CC
2617                        *用于连接两个字符串，B串接到A串后面：
2618                          A :CC: B  <----A为第一源字符串，B为第二源字符串。
2619
2620           *字符变量的声明和赋值
2621            *声明用GBLS或者LCLS。
2622            *赋值用SETS。
2623            示例：
2624                GBLS  string1
2625                GBLS  string2
2626                string1  SETS "aaaccc"
2627                string2  SETS "BB" :CC: (string2 :LEFT: 3); string2为：BBAAA
2628
2629
2630        (2).数字表达式
2631           *数字表达式通常由数字常量、数字变量、操作符和括号组成。
2632           *数字表达式表示一个32位的整数。当作为无符号数时，
2633           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
2634           *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
2635           时一样的。
2636           *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。
2637           <1>.整数数字量
2638              *decimal-digits
2639              *0x hexadecimal-digits
2640              *&  hexadecimal-digits
```

```
2641                *n_basse_n-digits
2642                示例：
2643                a       SETA 34906
2644                addr    DCD  0xA10E
2645                LDR     R4, &1000000F
2646                DCD     2_11001010        ;0xCA
2647                C3      DCD  8_74007
2648                DCQ     0x0123456789abcdef
2649             <2>.浮点数字量
2650                *{-}digits E{-}digits
2651                *{-}digits.digits E{-}digits
2652                *0x hexdigits
2653                *&  hexdigits
2654                *单精度浮点数表示的范围：3.40282347e+38 - 1.17549435e-38
2655                *双精度浮点数表示的范围：1.79769313486231571e+308
2656                                        - 2.22507385850720138e-308
2657                示例：
2658                DCFD        1E308,-4E-100
2659                DCFS        1.0
2660                DCFD        3.725E15
2661                LDFS        0x7FC00000
2662                LDFD        &FFF0 0000 0000 0000
2663
2664             <3>.数字变量
2665                *用GBLA/LCLA声明，用SETA赋值。
2666
2667             <4>.操作符
2668                *NOT    按位取反
2669                语法格式：
2670                 ：NOT: A    <----A为32位数字量
2671
2672                *+、-、*、/、MOD    加减乘除取余
2673                语法格式：
2674                A+B、A-B、A*B、A/B、A :MOD: B
2675
2676                *ROL/ROR/SHL/SHR位移
2677                语法格式：
2678                A :ROL: B       将整数A循环左移B位
2679                A :ROR: B       将整数A循环右移B位
2680                A :SHL: B       将整数A左移B位
2681                A :SHR: B       将整数A右移B位
2682
2683                *AND/OR/EOR   按位逻辑操作
2684                A :AND: B     A与B按位逻辑与操作
2685                A :OR: B      A与B按位逻辑或操作
2686                A :EOR: B     A与B按位逻辑异或操作
2687
2688          (3).基于寄存器和PC的表达式
2689             <1>.BASE
2690                *返回基于寄存器的表达式中的寄存器编号。
2691                语法格式：
2692                :BASE: A
2693             <2>.INDEX
2694                *返回基于寄存器的表达式相对于其基址寄存器的偏移量。
2695                语法格式：
2696                :INDEX: A
2697             <3>.+/-
2698                *正负号。它们可以放在数字表达式和基于PC的表达式前面。
2699
2700          (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
<3>.SB_OFFSET_11_0
*返回(label-SB)的bit[11:0]
语法格式:
:SB_OFFSET_11_0: label

CHAP4.4 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。

1.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:

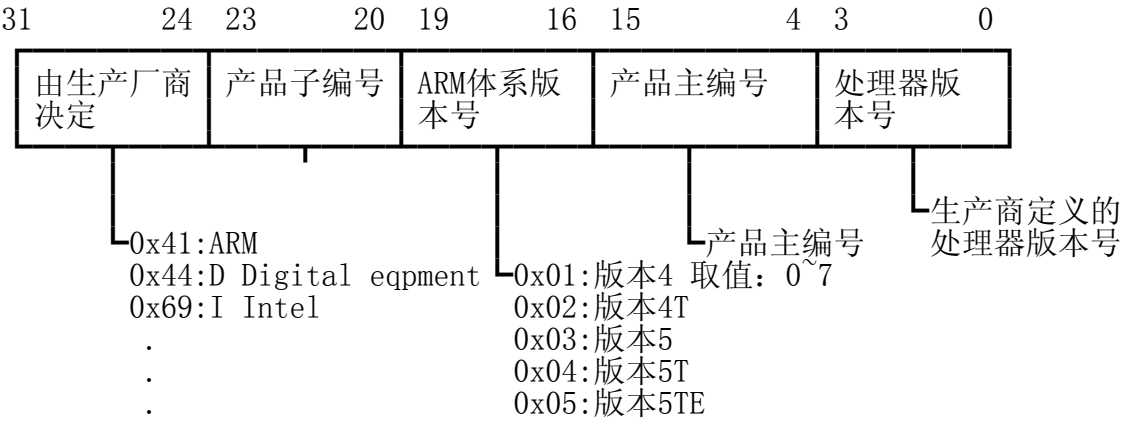
```
AREA EXAMPLE, CODE, READONLY
ENTRY
START   MOV R0, #10
        MOV R1, #3
        ADD R0, R0, R1
```

```
                END
CHAP4.5    ARM汇编编译器的使用

        ARMASM的各参数。
        *-16 告诉编译器所处理的源程序时THUMB指令的程序。其功能和在源程序用CODE16
             伪操作相同。
        *-32 告诉编译器所处理的源程序时ARM指令的程序。其功能和在源程序用CODE32
             伪操作相同。
        *-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：
             <1>./none                不使用任何ATPCS
             <2>./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。
             <3>./ropi                指定源程序是ROPI(只读位置无关)。默认。
                 /noropi
             <4>./pic                 是/ropi同义词。
                 /nopic               同/ropi。
             <5>./rwpi                指定源程序时读写位置无关。
                 /norwpi
             <6>./pid                 同/rwpi。
                ./nopid
             <7>./swstackcheck        指定源程序进行软件数据栈检查。
                 /noswstackcheck
             <8>./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。

        *-bigend 告诉源程序汇编成适合于BIG ENDIAN的模式。
        *-littleend 告诉源程序汇编成合适于LITTLE ENDIAN的模式。   默认。
        *-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。
        *-cpu 告诉ARMASM目标CPU的类型。
        *-depend 告诉ARMASM将源程序的依赖列表保存到文件dependfile。
        *-m 告诉ARMASM将源程序的依赖列表输出到标准输出。
        *-md 告诉ARMASM将源程序的依赖列表输出到文件inputfile.d。
        *-errors errorfile 告诉ARMASM将错误信息输出到文件errorfile中。
        *-fpu name 指定目标系统中的浮点运算单元的体系。可能的取值：
          <1>.none          指定没有浮点选项
          <2>.vfpv1         符合VFPV1的硬件向量浮点运算单元
          <3>.vfpv2         符合VFPV2的硬件向量浮点运算单元
          <4>.fpa           指定系统使用硬件的浮点加速器(float point accelerator)
          <5>.softvfp+vfp   指定系统中使用硬件向量浮点运算单元
          <6>.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
          <7>.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
        *-g 指示ARMASM产生DRAWF2格式的调试信息表。
        *-help 显示本汇编编译器的选项。
        *-i dir[,dir]...添加搜索路径。指定GET/INCLUDE中变量的搜索路径。
        *-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。
        *-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：
              <1>.-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。
              <2>.-width   指定列表文件中每行的宽度。
              <3>.-length  指定列表文件中每页的行数。
              <4>.-xref    指示ARMASM列出各符号的定义和引用情况。
        *-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
```

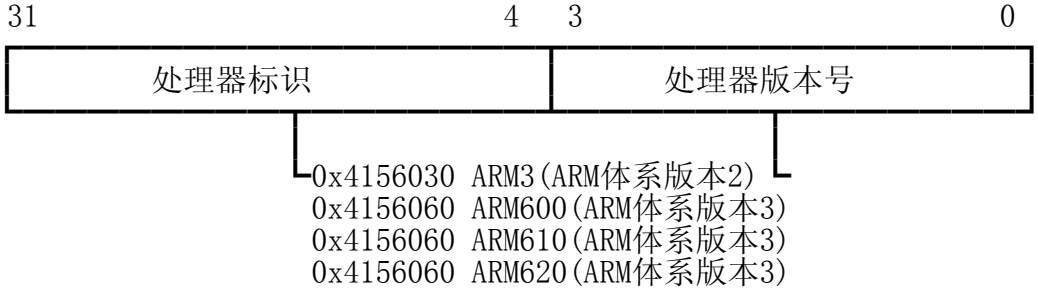
时将源程序缓存到内存中，在第二遍扫描时，从内存中读取该源文件)大小。
默认为8 MB。
*-memaccess attributes 指定目标系统的存储访问模式。默认的情况下时允许字节对齐、半字对齐、字对齐的读写访问。可以指定下面的访问属性。
<1>.+L41 允许非对齐的LDR访问
<2>.-L22 禁止半字的LOAD访问
<3>.-S22 禁止半字的STORE访问
<4>.-L22-S22 禁止半字的LOAD访问和STORE访问
*-nocache 禁止源程序cache。通常情况下ARMASM在第一遍扫描时把源程序保存到内存中，第二遍直接从内存中读取。
*-noesc 指示ARMASM忽略C语言风格的退出类的特殊字符。
*-noregs 指示ARMASM不要预定义寄存器名称。
*-nowarn 不产生警告信息。
*-o filename 指定输出的目标文件名称。
*-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。
*-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。
*-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。
*-via file 指示ARMASM从文件file中读取各选项信息。
*inputfile 为输入的源程序，必须为ARM汇编程序或者THUMB汇编程序。

/***/
/***/
第五章 ARM存储系统概述

1.ARM7之后的处理器主标识符编码格式(不完全)。



2.ARM7之前处理器主标识符。



CHAP5.3 MMU存储器管理单元

在ARM系统中,MMU主要完成以下工作：
1.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。
2.存储器访问权限的控制。
3.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，这个表就叫快表。TLB(translation lookaside buffer)。

/**********This chapter need REAL CPU type**********/

/***/
/***/
第六章 ATPCS介绍

CHAP6.1 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

CHAP6.2 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。

CHAP6.2.1 寄存器的使用规则
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。

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

CHAP6.2.2 数据栈使用规则

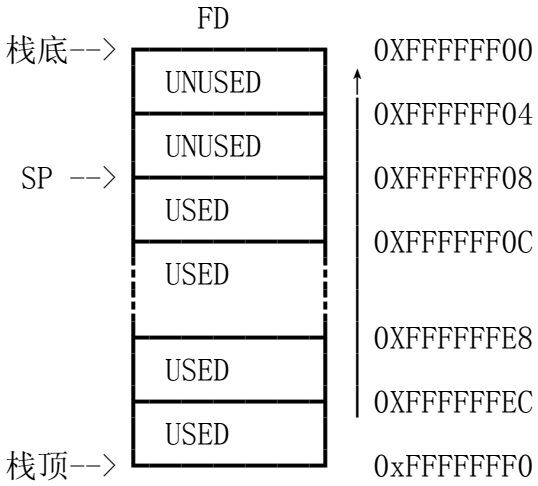
1.栈顶指针可以指向不同的位置。
*指向的栈顶不为空，称为FULL栈。反之，指向的为空数据单元，称为EMPTY栈。
栈顶指针增长方式也可以不同。
*向内存地址减小的方向增长时，称为DESCENDING。
向内存地址增加的方向增长时，称为ASCENDING。

	F(满)	E(空)
D(减)	FD(满减)	ED(空减)
A(增)	FA(满增)	EA(空增)

2008-1-31

2.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。
<1>.SP(stack pointer)最后一个写入栈的数据内存地址。
<2>.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。
<3>.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。
<4>.US(used stack)数据栈基址和栈指针之间的区域。
<5>.UUS(unused stack)数据栈界限到栈指针之间的区域。暂未使用部分。
<6>.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。



3.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:
<1>.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)。
<2>.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。

CHAP6.2.3 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。

CHAP6.3 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。

CHAP6.3.1 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：
<1>.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。
<2>.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。
<3>.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。
```

```
3121                CMP     IP, S1
3122                BLLO    __ARM_stack_overflow
3123                THUMB程序下的代码：
3124                LDR     WR, #-size
3125                ADD     WR, SP
3126                CMP     WR, S1
3127                BLO     __Thumb_stack_overflow
3128
3129  CHAP6.3.2     支持只读段位置无关(ROPI)的ATPCS
3130
3131          1.支持只读位置无关的ATPCS的应用场合。
3132           *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
3133           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
3134           常应用在以下一些场合：
3135           <1>.程序在运行期间动态加载到内存中。
3136           <2>.程序在不同的场合与不同的程序组合后加载到内存中。
3137           <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
3138               行时在建在到RAM中不同的地址。
3139
3140          2.遵守支持只读段位置无关的ATPCS的程序设计。
3141           这类程序设计规则：
3142           *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。
3143           *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
3144            ROPI段的位置关系必须固定。
3145           *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。
3146           *ROPI段移动后，对ROPI中的符号的引用要作相应的调整。
3147
3148
3149  CHAP6.3.3     支持可读写段位置无关(RWPI)的ATPCS。
3150
3151          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
3152          无关(RWPI)的ATPCS。
3153          *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
3154           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
3155           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
3156           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
3157           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
3158           加上SB即可生成该符号的地址。
3159
3160
3161  CHAP6.3.4     支持ARM程序和THUMB程序混合使用的ATPCS
3162
3163          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
3164          ARM,THUMB程序混合使用的ATPCS。它用在以下场合：
3165          *程序中存在ARM程序调用THUMB程序的情况。
3166          *程序中存在THUMB程序调用ARM程序的情况。
3167          *需要连接器进行ARM和THUMB状态切换的情况。
3168          *在下述情况，使用选项/nointerwork。本选项为默认选项。
3169           <1>.程序不包含THUMB程序。
3170           <2>.用户自己进行ARM状态和THUMB状态的切换。
3171          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。
3172
3173  CHAP6.3.5     处理浮点运算的ATPCS
3174
3175          *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
3176          的代码不兼容。
3177          *ADS的编译器和汇编器有下面6种与浮点数有关的选项：
3178          <1>.-fpu VFP
3179          <2>.-fpu FPA
3180          <3>.-fpu softVFP
```

<4>.-fpu softVFP+VFP
<5>.-fpu softFPA
<6>.-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：
<1>.如果要与FPA体系兼容，应选择-fpu softFPA
<2>.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
<3>.其他情况下选择选项-fpu softVFP。

/***/
/***/
第七章 ARM程序和THUMB程序混合使用

CHAP7.1 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：
<1>.编译器和汇编器将interwork的属性写入到目标文件中。
<2>.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)。
<3>.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。
<4>.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：
<1>.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。
<2>.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。
<3>.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。
<4>.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。

CHAP7.2 在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。

CHAP7.2.1 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，也可以时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)

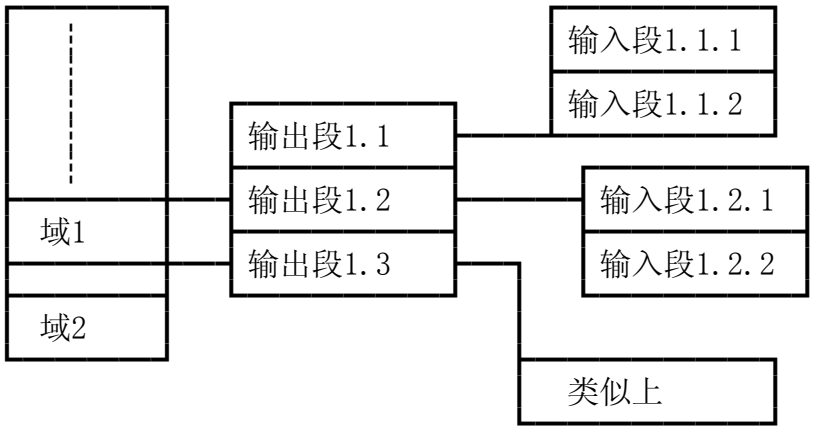
3.BLX(2)指令。
*这种格式的BLX指令从ARM指令跳转到指令中指定的目标地址，并将程序状态切换到THUMB状态，该指令同时将PC寄存器的内容复制到LR寄存器中。
语法格式：
BLX{<cond>} <Rm>
其中：<Rm>寄存器的bit[0]为0时，目标地址处的指令时ARM指令；当<Rm>寄存器的;bit[0]为1时，目标地址处的指令为THUMB指令。当<Rm>为R15时，会产生不可预知的结果。
指令的伪代码：
if cond passed then
LR = address of the instruction after the BLX instruction
T = Rm[0]
PC = Rm AND 0xFFFFFFFE

CAHP7.2.3 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
```

```
3301                ADD R2, R2, R3
3302                ADR R0, ARMprog          ;最低位为0，以便切换到ARM状态。
3303                BX  R0
3304
3305                CODE32
3306           ARMprog
3307                MOV R4, #4
3308                MOV R5, #5
3309                ADD R4, R4, R5
3310
3311           stop MOV R0, #0X18
3312                LDR R1, =0X20026
3313                SWI 0X123456
3314
3315                END
3316
3317  2008-2-2
3318  CHAP7.3   在C/C++程序中实现interwork
3319
3320           1.需要考虑interwork的场合。
3321            *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
3322             该C/C++程序时指定选项-apcs/interwork。
3323            *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
3324             用另一种指令系统的子程序时，需要在编译该C/C++程序时
3325             指定选项-apcs/interwork。
3326            *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，
3327             这时不要使用函数指针来调用该被调用程序。
3328            *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
3329             连接。
3330            *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。
3331
3332           2.编译选项-apcs/interwork的作用。
3333            * tcc -apcs/interwork
3334            * armcc -apcs/interwork
3335            * tcpp -apcs/interwork
3336            * armcpp -apcs/interwork
3337
3338            指定编译选项-apcs/interwork后，编译器会进行以下的处理：
3339            *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%。
3340            *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
3341             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。
3342            *对于非子叶程序，Thumb编译器将进行一些指令替换。如：
3343             POP{R4,R5,PC}
3344             替换为:
3345             POP{R4,R5}
3346             POP{R3}
3347             BX R3
3348            *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
3349             的用于程序状态切换的代码段。
3350
3351  /*****************************************************************************/
3352                          NULL FOR LATER
3353  /*****************************************************************************/
3354
3355
3356  第11章     ARM连接器
3357
3358  CAHP11.1.1     ARM映像文件的组成
3359           1.ARM映像文件的组成部分。
3360            *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(output
```

section)和输入段(input section)。各部分的关系如下:
<1>.一个映像文件由一个或多个域组成。
<2>.每个域包含一个或多个输出段。
<3>.每个输入段包含一个或多个输入段。
<4>.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

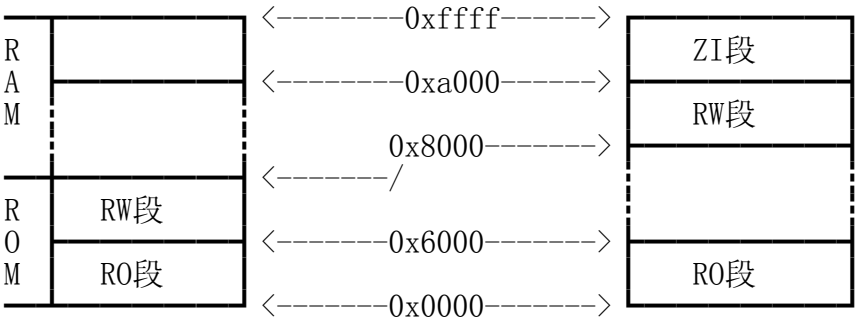
ARM连接器根据这些输入段的属性，再把这些输入段分组，再组成不同的输出段以及
这样我们可以理解:一个输出段就时一些同属性的输入段组成的。这个输出段的属性
组成它的同属性输入段的属性。

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
。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

2.ARM映像文件个组成部分的地址映射。

<1>.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
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F
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

<2>.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。

<3>.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
* 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
的连接器选项告诉连接器相关的地址映射关系。
* -ropi
* -rwpi
* -ro_base
* -rw_base
* -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

CHAP11.1.2 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
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
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:
<1>.初始入口点必须位于映像文件的运行时域内。
<2>.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
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
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，
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
初始入口点。

3481 * 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
3482 的初始入口点。
3483 * 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
3484 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:
3485 L6035W : Image does not have an entry point .(not specified of not set c
3486 to multiple choices)
3487
3488 4.普通入口点的使用方法。
3489 * 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
3490 断
3491 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
3492 * 一个映像文件可以有多个普通入口点。
3493 * 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
3494 普通入口点被连接器当成映像文件的初始入口点。
3495
3496
3497 CHAP11.1.3 输入段的排序规则
3498
3499 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
3500 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
3501 地址和该输出段中个输入段的排列顺序有关。
3502
3503 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
3504 用-fist和-last来改变这些规则。
3505 * 输入段的属性
3506 按照输入段的属性其排列顺序如下:
3507 *只读代码段 ---------- RO-CODE
3508 *只读数据段 ---------- RO-DATA
3509 *可读写代码段 ---------- RW-CODE
3510 *其他已经初始化的数据段 ----- I-DATA
3511 *未初始化的数据 ---------- UI-DATA
3512
3513 * 输入段的名称
3514 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
3515 按
3516 照ASCII码顺序进行排序。
3517
3518 * 个输入段在连接命令行的输入段列表中的排列顺序。
3519 <1>.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
3520 <2>.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
3521 使
3522 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。
3523 <3>.连接选项-first，-last选项不能改变根据段属性进行的排序。它只能改变相
3524 输
3525 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
3526 -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
3527 该输入段才能位于整个运行时域的开始位置。
3528 <4>.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
3529 使输入段满足地址对齐要求。
3530
3531
3532 CHAP11.2 ARM连接器介绍
3533
3534 1.ARM开发包中的连接器ARMLINK可以完成以下操作:
3535 * 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。
3536 * 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
3537 为
3538 部分连接。
3539 * 指定代码和数据在内存中的位置。
3540 * 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
  <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:
  <1>. 删除重复的调试信息段。
  <2>. 最小话符号表的大小。
  <3>. 保留那些未被解析的符号。
  <4>. 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。
  <1>. -output
  <2>. -partial
  <3>. -elf
* 指定选项文件，其中可以包含一些连接选项。
  <1>. -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。
  <1>. -first
  <2>. -last
  <3>. -debug/-nodebug
  <4>. -entry
  <5>. -keep
  <6>. -libpath
  <7>. -edit
  <8>. -locals/-nolocals
  <9>. -remove/-noremove
  <10>. -scanlib/-noscanlib

* 生成与映像文件相关的信息。
  <1>. -callgrah
  <2>. -infor
  <3>. -map
  <4>. -symbols
  <5>. -symdefs
  <6>. -xref
  <7>. -xreffrom
  <8>. -xrefto

* 控制ARMLINK生成相关的诊断信息。
  <1>. -errors
  <2>. -list
  <3>. -verbose
  <4>. -strict

<5>. -unsolved
<6>. -mangled
<7>. -unmangled

CHAP11.3 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$$。ARM连接器在生成映像文件时
它们来代表映像文件中各域的起始地址及存储区域界限，各输出段的起始地址和区域
，各输入段的起始地址和区域的界限。
如:
* load$$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。
* 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
引用。
* 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。
* 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。

CHAP11.3.1 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
。
* 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
region_name :
<1>. 对于只读的域，使用名称ER_RO。
<2>. 对于可读写的域，使用名称ER_RW。
<3>. 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:

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
生成另外的符号。这些符号为:

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)

CHAP11.3.2 连接器生成的与输出段相关的符号

1.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:

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

表: 连接器生成的与输出段相关的符号

2.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。

CHAP11.3.2 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

CHAP11.4 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THU
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
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
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。

CHAP11.5 运行时库的使用

/***/
待
/***/

CHAP11.6 从一个映像文件中使用另一个映像文件中的符号

/***/
待
/***/

CHAP11.7 隐藏或者重命名全局符号

/***/
待
/***/

CHAP11.8 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(Position Independent)。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
域是位置无关的。指定本操作，ARM连接器保证以下操作:

3781 * 检查并确保各RW属性的运行时域包含的各输入段PI。
3782 * 检查各段之间的重定位关系，保证其时合法的。
3783 * 在Region$$Table和ZIsection$$Table中添加基于静态寄存
3784 的条目。
3785 通常情况下可读写属性的输入段应该时PI的。
3786
3787 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
3788 。
3789 其中:
3790 * 一个加载时域包含所有的RO属性的输出段。可以使用-ro-
3791 来修改其加载时的地址。
3792 * 一个加载时域包含所有的RW属性的输出段。可以用-rw-bas
3793 修
3794 改其加载时地址。
3795
3796 11. -scatter file 指定分散加载文件。
3797
3798 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
3799 入
3800 段、符号表以及字符串表。
3801
3802 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
3803 别
3804 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
3805 文件)，但是不处理加载到存储器中的映像文件(.bin文件)。
3806 * ARM连接器进行部分连接，则生成的目标文件不包含调试信
3807 单依然包含了符号表以及字符串表。
3808 * 如果将来要使用fromELF来转换映像文件的格式，则在生成
3809 像文件时不要使用-nodebug选项。
3810
3811 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。
3812 -remove RO / -remove RW / -remov ZI。
3813
3814 15. -unremove 不删除映像文件中没有使用的段。
3815
3816 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
3817 的
3818 条件:
3819 * 必须位于映像文件的运行时域内。
3820 * 包含初始入口点的运行使用不能被覆盖，它的加载时地址
3821 行
3822 时地址必须相同。
3823 选项中的location可能的取值有以下几条:
3824 * 入口的点的地址。如：-entry address
3825 * 地址符号：symbol。如：-entry int-handler。如果被指
3826 符
3827 号在映像文件中多个定义，连接器报错。
3828 * 相对于某个目标中特定的一定的偏移量。如：-entry
3829 8+startup
3830
3831 17. -keep section-id 指定目标section-id不能被删除。section-id可能的取值有:
3832 * symbol。 包含symbol符号的输入段都将被保留。
3833 * object 目标文件中的段将被保留。如：-keep XXX.o。可
3834 ARM连接器命令行包含多个这样的-keep。
3835 * object。当目标文件中只有一个输入段时，该段被保留。
3836
3837 18. -first section-id 将输入段section-id放置在运行时域的开始位置。
3838
3839 19. -last section-id 将输入段section-id放置在运行时域的最后位置。
3840

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。

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。i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：
*被interworking的小代码段调用的函数。
*在本映像文件之外调用的函数。
*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：
*每次调用使用数据栈的大小。
*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sectic
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
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
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

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
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
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
列
表。

CHAP11.9 使用scatter文件定义映像文件的地址映射

CHAP11.9.1 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。
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：

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

表：BNF语法的基本元素

同是，在使用BNF语法描述scatter文件的格式时，还定义了一些元素。如表：

CHAP11.9.2 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
  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
  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
  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
  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
      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
  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*
}
```

```
4021        base_designator ::= base_address | (PLUS offset)
4022        详解:
4023         * 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。
4024         * 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:
4025           <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
4026                             齐的。
4027
4028           <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
4029                       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
4030                       起始地址即offset。
4031
4032           <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
4033           值     为ABSOLUTE。
4034              * PI       位置无关属性。
4035              * RELOC    重定位。
4036              * 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
4037                         重叠，需要用户自己提供地址空间重叠的管理机制。
4038              * ABSOLUTE
4039
4040           <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
4041           告错误。
4042
4043           <5>.input_region_description 含义后面有介绍。
4044
4045        3.输入段描述。
4046         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
4047         述，如下:
4048
4049         input_section_description ::= module_selector_pattern [{input_selectors}]
4050
4051         详解:
4052         * 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,
4053         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
4054         的。满足下面条件之一，认为该输入段是与module_selector_pattern匹配的。
4055             <1>.包含输入段的目标文件的名称与module_selector_pattern匹配。
4056             <2>.包含输入段的库成员名(不带前导路径)与module_selector_pattern匹配。
4057             <3>.包含输入段的库的代路径名称与module_selector_pattern匹配。
4058
4059         *input_selectiors含义后面介绍。
4060
4061        4.输入段选择符。
4062        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
4063        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+,
4064        符号+前面紧跟的逗号可以省略。描述如下:
4065
4066        input_selectiors ::= (PLUS input_selection_attrs | input_section_pat)
4067        ([COMMA] PLUS input_section_attrs | COMMA input_section_pat)*
4068
4069        详解:
4070        * 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
4071          的，包括:
4072          <1>. RO-CODE
4073          <2>. RO-DATA
4074          <3>. RO 包括了RO-CODE,RO-DATA
4075          <4>. RW-DATA
4076          <5>. RW 包括了RW-CODE,RW-DATA
4077          <6>. ZI
4078          <7>. CODE 同RO-CODE
4079          <8>. CONST 同RO-DATA
4080          <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
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，
号？代表单个字符。进行匹配时，所有字符是大小写无关的。

CHAP11.9.3 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
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
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
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
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
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
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。
    }
```

```
4141
4142            ER_RW 0x80004000 ;第二个运行时域的名称为：ER_RW 。
4143            {                ;它的起始地址为0x80004000
4144               *(+RW)        ;本域包含了所有RW属性的输入段，它们被连续放置。
4145            }
4146
4147            ER_ZI +0         ;第三个运行时域的名称为：ER_ZI 。
4148            {                ;它的起始地址为上个域结束位置的下一个地址。
4149               *(+ZI)        ;本域包含了所有ZI属性的输入段，它们被连续放置。
4150            }
4151        }
4152
4153
4154        3.2个加载时域和3个不连续的运行时域
4155
4156        scatter文件内容：
4157
4158        命令：  -split
4159                -ro-base 0x80000000
4160                -rw-base 0x80004000
4161
4162        LR_1 0X80000000      ;定义第一加载时域的名称为LR_1，起始地址为0x80000000
4163        {                    ;开始定义运行时域。
4164            ER_RO +0         ;第一个运行时域的名称为：ER_RO 。
4165            {                ;其起始地址为0X80000000 。
4166               *(+RO)        ;本域包含了所有的RO属性的输入段，它们被连续放置。
4167            }
4168        }
4169
4170        LR_2 0X80004000      ;定义第二加载时域的名称为LR_2，起始地址为0X80004000
4171        {
4172            ER_RW +0         ;第二个运行时域的名称为：ER_RW 。
4173            {                ;它的起始地址为0x80004000
4174               *(+RW)        ;本域包含了所有RW属性的输入段，它们被连续放置。
4175            }
4176
4177            ER_ZI +0         ;第三个运行时域的名称为：ER_ZI 。
4178            {                ;它的起始地址为上个域结束位置的下一个地址。
4179               *(+ZI)        ;本域包含了所有ZI属性的输入段，它们被连续放置。
4180            }
4181        }
4182
4183        4.固定的运行时域。
4184        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i
4185        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
4186        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
4187        错误信息：
4188        L6203E: Entry point(0x00000000)lies within non-root region 32bitARM
4189
4190        使用scatter文件时，可以有下面两种方法来设置固定域：
4191        第一种方法是设定一个加载时域的第一个运行时域的运行时地址，使其和该加载时域
4192        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：
4193        (1).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
4194        包含的第一个域。
4195        (2).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.
4196        (3).设置该运行时域属性为ABSOLUTE，这也是默认的值。
4197
4198    2008-2-3
4199        5.使用FIXED属性将某个域放置在ROM中固定位置。
4200        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
```

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
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
        .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。
   }
}
```

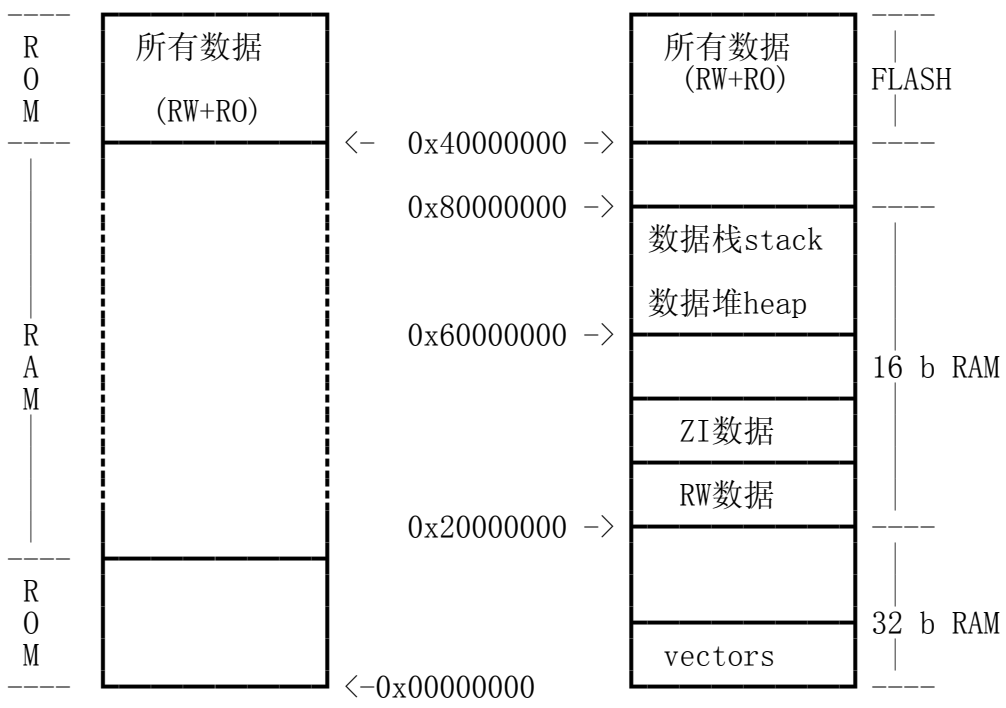
6.一个接近实际系统的例子。
* 在一个嵌入式系统中，为了保持好的性能价格比，通常存在多种存储器。

* 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
  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
  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
  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
  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
  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0x0000
  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：

1. 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
   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。
   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。
2. 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。经
   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
   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
   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
   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
   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F
    {
        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。
    }
}
```