第 4 章　ARM 汇编语言程序设计

本章介绍如何编写 ARM 和 Thumb 汇编语言程序。同时介绍 ARM 汇编编译器 armasm 的使用方法。

4.1　伪　操　作

ARM 汇编语言源程序中语句由指令、伪操作和宏指令组成。在 ARM 中伪操作称为 derective，这里为保持和国内在 IBM PC 汇编语言中对名词翻译的一致性 derective 称为伪操作；同样在 ARM 中宏指令被称为 pseudo-instruction，这里将其称为宏指令，宏指令也是通过伪操作定义的。本节介绍伪操作和宏指令。伪操作不像机器指令那样在计算机运行期间由机器执行，它是在汇编程序对源程序汇编期间由汇编程序处理的。宏是一段独立的程序代码。在程序中通过宏指令调用该宏。当程序被汇编时，汇编程序将对每个宏调用作展开，用宏定义体取代源程序中的宏指令。本节介绍以下类型的 ARM 伪操作和宏指令。

- 符号定义(Symbol definition)伪操作。
- 数据定义(Data definition)伪操作。
- 汇编控制(Assembly control)伪操作。
- 框架描述(Frame description)伪操作。
- 信息报告(Reporting)伪操作。
- 其他(Miscellaneous)伪操作。

4.1.1　符号定义伪操作

符号定义(Symbol definition)伪操作用于定义 ARM 汇编程序中的变量，对变量进行赋值以及定义寄存器名称。包括以下伪操作。

- 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 及 GBLS

GBLA, GBLL 及 GBLS 伪操作用于声明一个 ARM 程序中的全局变量，并将其初始化。

GBLA 伪操作声明一个全局的算术变量，并将其初始化成 0。

GBLL 伪操作声明一个全局的逻辑变量，并将其初始化成{FALSE}。

GBLS 伪操作声明一个全局的串变量，并将其初始化成空串""。

语法格式

```
<gblx> variable
```

其中：

<gblx> 是后面 3 种伪操作之一： GBLA, GBLL 或者 GBLS。

variable 是所说明的全局变量的名称。在其作用范围内必须惟一。

使用说明

如果用这些伪操作重新声明已经声明过的变量，则变量的值将被初始化成后一次声明语句中的值。

全局变量的作用范围为包含该变量的源程序。

示例

```
GBLA objectsize          ; 声明一个全局的算术变量
objectsize SETA 0xff     ; 向该变量赋值
SPACE objectsize         ; 引用该变量

GBLL statusB             ; 声明一个全局的逻辑变量 statusB
statusB SETL {TRUE}      ; 向该变量赋值
```

2. LCLA, LCLL 及 LCLS

LCLA, LCLL 及 LCLS 伪操作用于声明一个 ARM 程序中的局部变量，并将其初始化。

LCLA 伪操作声明一个局部的算术变量，并将其初始化成 0。

LCLL 伪操作声明一个局部的逻辑变量，并将其初始化成{FALSE}。

LCLS 伪操作声明一个局部的串变量，并将其初始化成空串""。

语法格式

```
<lclx> variable
```

其中，

<lclx> 是后面 3 种伪操作之一： LCLA, LCLL 或者 LCLS。

variable 是所说明的局部变量的名称。在其作用范围内必须惟一。

使用说明

如果用这些伪操作重新声明已经声明过的变量，则变量的值将被初始化成后一次声明语句中的值。

局部变量的作用范围为包含该局部变量的宏代码的一个实例。

示例

```
MACRO                    ; 声明一个宏
$label  message $a       ; 宏的原型
LCLS err                 ; 声明一个局部串变量 err
```

```
err     SETS "error no: "    ; 向该变量赋值
$label                       ; 代码
INFO 0, "err":CC::STR:$a     ; 使用该串变量
MEND                         ; 宏定义结束
```

3. SETA, SETL 及 SETS

SETA, SETL 及 SETS 伪操作用于给一个 ARM 程序中的变量赋值。

SETA 伪操作给一个算术变量赋值。

SETL 伪操作给一个逻辑变量赋值。

SETS 伪操作给一个串变量赋值。

语法格式

```
<setx> variable expr
```

其中：

<setx> 是后面 3 种伪操作之一： SETA, SETL 或者 SETS。

variable 是使用 GBLA、GBLL、 GBLS 、LCLA、LCLL 或 LCLS 说明的变量的名称。在其作用范围内必须惟一。

expr 为表达式，即赋予变量的值。

使用说明

在向变量赋值前，必须先声明该变量。

示例

```
GBLA objectsize          ; 声明一个全局的算术变量
objectsize SETA 0xff     ; 向该变量赋值
SPACE objectsize         ; 引用该变量
GBLL statusB             ; 声明一个全局的逻辑变量 statusB
statusB SETL {TRUE}      ; 向该变量赋值
```

4. RLIST

RLIST 为一个通用寄存器列表定义名称。

语法格式

```
name RLIST {list-of-registers}
```

其中：

name 是寄存器列表的名称。

{list-of-registers} 为通用寄存器列表。

使用说明

RLIST 伪操作用于给一个通用寄存器列表定义名称。定义的名称可以在 LDM/STM 指令中使用。

在 LDM/STM 指令中，寄存器列表中的寄存器的访问次序总是先访问编号较低的寄存器，再访问编号较高的寄存器，而不管寄存器列表中各寄存器的排列顺序。

示例

```
Context RLIST {r0-r6,r8,r10-r12,r15}  ; 将寄存器列表名称定义为 Context
```

5. CN

CN 为一个协处理器的寄存器定义名称。

语法格式

```
name CN expr
```

其中：

name 是该寄存器的名称。

expr 为协处理器的寄存器的编号，数值范围为 0～15。

使用说明

CN 伪操作用于给一个协处理器的寄存器定义名称。方便程序员记忆该寄存器的功能。

示例

```
Power CN 6        ; 将协处理器的寄存器 6 名称定义为 Power
```

6. CP

CP 为一个协处理器定义名称。

语法格式

```
name CP expr
```

其中：

name 是该协处理器的名称。

expr 为协处理器的编号，数值范围为 0～15。

使用说明

CP 伪操作用于给一个协处理器定义名称，方便程序员记忆该协处理器的功能。

示例

```
Dmu CN 6  ; 将协处理器 6 名称定义为 Dmu
```

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。

语法格式

```
name DN expr
name SN expr
```

其中：

name 是该 VFP 寄存器的名称。

expr 为 VFP 双精度寄存器编号(0～15)或者 VFP 单精度寄存器编号(0～31)。

使用说明

DN 和 SN 伪操作用于给一个 VFP 寄存器定义名称，方便程序员记忆该寄存器的功能。

示例

```
height DN 6  ; 将 VFP 双精度寄存器 6 名称定义为 height
width SN 16  ; 将 VFP 单精度寄存器 16 名称定义为 width
```

8. FN

FN 为一个 FPA 浮点寄存器定义名称。

语法格式

```
name FN expr
```

其中：

name 是该浮点寄存器的名称。

expr 为浮点寄存器的编号，数值范围为 0～7。

使用说明

FN 伪操作用于给一个浮点寄存器定义名称，方便程序员记忆该协处理器的功能。

示例

```
Height FN 6  ; 将浮点寄存器 6 名称定义为 Height
```

4.1.2　数据定义伪操作

数据定义(Data definition)伪操作包括以下的伪操作。

- LTORG　声明一个数据缓冲池(literal pool)的开始。
- MAP　定义一个结构化的内存表(storage map)的首地址。
- FIELD　定义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 及 DCDU　分配一段字的内存单元，并用指定的数据初始化。
- DCDO　分配一段字的内存单元，并将个单元的内容初始化成该单元相对于静态基值寄存器的偏移量。
- DCFD 及 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 及 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是代码，而不是数据。
- DCQ 及 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 及 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。现已不再使用，仅用于保持向前兼容。

1. LTORG

LTORG 用于声明一个数据缓冲池(literal pool)的开始。

语法格式

```
LTORG
```

使用说明

通常 ARM 汇编编译器把数据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，这样处理器就不会错误地将数据缓冲池中的数据当作指令来执行。

示例

```
AREA    Example, CODE, READONLY
start   BL      func1
func1                       ; 子程序
; code
LDR r1,=0x55555555         ; => LDR R1, [pc, #offset to Literal Pool 1}
; code
MOV pc,lr                   ; 子程序结束
LTORG                       ; 定义数据缓冲池 &55555555
data    SPACE 4200          ; 从当前位置开始分配 4200 字节的内存单元
END                         ; 默认的数据缓冲池为空
```

2. MAP

MAP 用于定义一个结构化的内存表(storage map)的首地址。此时，内存表的位置计数器{VAR}设置成该地址值。^是 MAP 的同义词。

语法格式

```
MAP expr{,base-register}
```

其中：

expr 为数字表达式或者是程序中的标号。当指令中没有 base-register 时，expr 即为结构化内存表的首地址。此时，内存表的位置计数器{VAR}设置成该地址值。当 expr 为程序中的标号时，该标号必须是已经定义过的。

base-register 为一个寄存器。当指令中包含这一项时，结构化内存表的首地址为 expr 和 base-register 寄存器值的和。

使用说明

MAP 伪操作和 FIELD 伪操作配合使用来定义结构化的内存表结构。具体使用方法在 FIELD 中详细介绍。

示例

```
MAP    0x80,R9              ; 内存表的首地址为 R9+0x80
```

3. FIELD

FIELD 用于定义一个结构化内存表中的数据域。#是 FIELD 的同义词。

语法格式

```
{label} MAP expr
```

其中：

{label}为可选的。当指令中包含这一项时，label 的值为当前内存表的位置计数器{VAR}的值。汇编编译器处理了这条 FIELD 伪操作后，内存表计数器的值将加上 expr。

expr 表示本数据域在内存表中所占的字节数。

使用说明

MAP 伪操作和 FIELD 伪操作配合使用来定义结构化的内存表结构。MAP 伪操作定义内存表的首地址；FIELD 伪操作定义内存表中各数据域的字节长度，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 寄存器值对于其后所有的 FIELD 伪操作定义的数据域是默认使用的，直至遇到新的包含 base-register 项的 MAP 伪操作。

MAP 伪操作和 FIELD 伪操作仅仅是定义数据结构，它们并不实际分配内存单元。

示例

例 1：　下面的伪操作序列定义一个内存表，其首地址为固定地址 4096(0x1000)，该内存表中包含 5 个数据域：consta 长度为 4 个字节；constb 长度为 4 个字节；x 长度为 8 个字节；y 长度为 8 个字节；string 长度为 256 个字节。这种内存表称为基于绝对地址的内存表。

```
MAP 4096                          ; 内存表的首地址为 4096(0x1000)
consta      FIELD   4             ; consta 长度为 4 个字节，相对位置为 0
constb      FIELD   4             ; constb 长度为 4 个字节，相对位置为 5000
x           FIELD   8             ; x 长度为 4 个字节，相对位置为 5004
y           FIELD   8             ; y 长度为 4 个字节，相对位置为 5012
string      FIELD   256           ; string 长度为 256 字节，相对位置为 5020
                                  ; 在指令中可以这样引用内存表中的数据域：

LDR   R6,consta
```

上面的指令仅仅可以访问 LDR 指令前面(或后面)4 KB 地址范围的数据域。

例 2：　下面的伪操作序列定义一个内存表，其首地址为 0，该内存表中包含 5 个数据域：consta 长度为 4 个字节；constb 长度为 4 个字节；x 长度为 8 个字节；y 长度为 8 个字节；string 长度为 256 个字节。这种内存表称为基于相对地址的内存表。

```
MAP 0                             ; 内存表的首地址为 0
consta      FIELD   4             ; consta 长度为 4 个字节，相对位置为 0
```

```
constb      FIELD   4       ; constb长度为4个字节，相对位置为4
x           FIELD   8       ; x长度为4个字节，相对位置为8
y           FIELD   8       ; y长度为4个字节，相对位置为16
string      FIELD   256     ; string长度为256字节，相对位置为24
```

可以通过下面的指令方便地访问地址范围超过4 KB的数据。

```
MOV R9,#4096
LDR R5,[R9,constb]          ; 将内存表中数据域constb读取到R5中
```

在这里，内存表中各数据域的实际内存地址不是基于一个固定的地址，而是基于LDR指令执行时R9寄存器中的内容。这样通过上面方法定义的内存表结构可以在程序中有多个实例(通过在LDR指令中指定不同的基址寄存器值来实现)。

在ARM-Thumb的过程调用标准中，通常用R9作为静态基址寄存器。

例3：下面的伪操作序列定义一个内存表，其首地址为0与R9寄存器值的和，该内存表中包含5个数据域：consta长度为4个字节；constb长度为4个字节；x长度为8个字节；y长度为8个字节；string长度为256个字节。这种内存表称为基于相对地址的内存表。

```
MAP 0 ,R9                   ; 内存表的首地址为0与R9寄存器值的和
consta      FIELD   4       ; consta长度为4个字节，相对位置为0
constb      FIELD   4       ; constb长度为4个字节，相对位置为4
x           FIELD   8       ; x长度为4个字节，相对位置为8
y           FIELD   8       ; y长度为4个字节，相对位置为16
string      FIELD   256     ; string长度为256字节，相对位置为24
```

可以通过下面的指令方便地访问地址范围超过4 KB的数据。

```
ADR R9,DATASTART
LDR R5, constb              ; 相当于LDR R5,[R9,#4]
```

在这里，内存表中各数据域的实际内存地址不是基于一个固定的地址，而是基于LDR指令执行时R9寄存器中的内容。这样通过上面方法定义的内存表结构可以在程序中有多个实例(通过在LDR指令前指定不同的基址寄存器R9值来实现)。

例4：下面的伪操作序列定义一个内存表，其首地址为PC寄存器的值，该内存表中包含5个数据域：consta长度为4个字节；constb长度为4个字节；x长度为8个字节；y长度为8个字节；string长度为256个字节。这种内存表称为基于PC的内存表。

```
Datastruc   SPACE   280     ; 分配280字节的内存单元
MAP Datastruc               ; 内存表的首地址为Datastruc内存单元
consta      FIELD   4       ; consta长度为4个字节，相对位置为0
constb      FIELD   4       ; constb长度为4个字节，相对位置为4
x           FIELD   8       ; x长度为4个字节，相对位置为8
y           FIELD   8       ; y长度为4个字节，相对位置为16
string      FIELD   256     ; string长度为256字节，相对位置为24
```

可以通过下面的指令方便地访问地址范围不超过4KB的数据。

```
LDR R5, constb              ; 相当于LDR R5, [PC, offset]。
```

在这里，内存表中各数据域的实际内存地址不是基于一个固定的地址，而是基于 PC 寄存器的值。这样在使用 LDR 指令访问内存表中的数据域时，不必使用基值寄存器。

例 5：　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，汇编编译器处理该条伪操作后，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断当前内存的使用没有超过程序分配的可用内存。下面的伪操作序列定义一个内存表，其首地址为 PC 寄存器的值，该内存表中包含 5 个数据域：consta 长度为 4 个字节；constb 长度为 4 个字节；x 长度为 8 个字节；y 长度为 8 个字节；string 长度为 maxlen 个字节，未防止 maxlen 的取值使得内存使用越界，可以利用 endofstru 监视内存的使用情况，保证其不超过 endofmem。

```
startofmem  EQU   0x1000      ; 分配的内存首地址
endofmem    EQU   0x2000      ; 分配的内存末地址
MAP startofmem                ; 内存表的首地址为 startofmem 内存单元
consta      FIELD   4         ; consta 长度为 4 个字节，相对位置为 0
constb      FIELD   4         ; constb 长度为 4 个字节，相对位置为 4
x           FIELD   8         ; x 长度为 4 个字节，相对位置为 8
y           FIELD   8         ; y 长度为 4 个字节，相对位置为 16
string      FIELD   maxlen    ; string 长度为 maxlen 字节，相对位置为 24
endofstru   FIELD   0
ASSERT  endofstru<=endofmem
```

4. SPACE

SPACE 用于分配一块内存单元，并用 0 初始化。%是 SPACE 的同义词。

语法格式

```
{label} MAP expr
```

其中：

{label}为可选的。

expr 表示本伪操作分配的内存字节数。

示例

```
Datastruc   SPACE   280       ; 分配 280 字节的内存单元，并将内存单元内容初始化成 0
```

5. DCB

DCB 用于分配一段字节内存单元，并用伪操作中的 expr 初始化之。=是 DCB 的同义词。

语法格式

```
{label} MAP expr{,expr} …
```

其中：

{label}为可选的。

expr 可以为–128～255 的数值或者为字符串。

示例

```
Nullstring   DCB   "Null string",0   ; 构造一个以 NULL 结尾的字符串
```

6. DCD 及 DCDU

DCD 用于分配一段字内存单元(分配的内存都是字对齐的)，并用伪操作中的 expr 初始化之。&是 DCD 的同义词。

DCDU 与 DCD 的不同之处在于 DCDU 分配的内存单元并不严格字对齐。

语法格式

```
{label} DCD expr{,expr} …
```

其中：

{label}为可选的。

expr 可以为数字表达式或者为程序中的标号。

使用说明

DCD 伪操作可能在分配的第一个内存单元前插入填补字节(padding)以保证分配的内存是字对齐的。

DCDU 分配的内存单元则不需要字对齐。

示例

```
data1 DCD 1,5,20          ; 其值分别为 1、5 和 20
data2  DCD memaddr+ 4     ; 分配一个字单元，其值为程序中标号 memaddr 加 4 个字节
```

7. DCDO

DCDO 用于分配一段字内存单元(分配的内存都是字对齐的)，并将个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

```
{label} DCDO expr{,expr} …
```

其中：

{label}为可选的。

expr 可以为数字表达式或者为程序中的标号。

使用说明

DCD 伪操作为基于静态基址寄存器 R9 的偏移量分配内存单元。

示例

```
IMPORT  externsym
DCDO    externsym    ; 32 位的字单元，其值为标号 externsym 基于 R9 的偏移量
```

8. DCFD 及 DCFDU

DCFD 用于为双精度的浮点数分配字对齐的内存单元，并将个字单元的内容初始化为 fpliteral 表示的双精度浮点数。每个双精度的浮点数占据两个字单元。

DCFD 与 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

```
{label} DCFD{U} fpliteral{,fpliteral}...
```

其中，{label}为可选的。

fpliteral 为双精度的浮点数。

使用说明

DCFD 伪操作可能在分配的第一个内存单元前插入填补字节以保证分配的内存是字对齐的。

DCFDU 分配的内存单元则不需要字对齐。

如何将 fpliteral 转换成内存单元的内部表示形式是由浮点运算单元控制的。

示例

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

9. DCFS 及 DCFSU

DCFS 用于为单精度的浮点数分配字对齐的内存单元，并将个字单元的内容初始化成 fpliteral 表示的单精度浮点数。每个单精度的浮点数占据 1 个字单元。

DCFS 与 DCFSU 的不同之处在于 DCFSU 分配的内存单元并不严格字对齐。

语法格式

```
{label} DCFS{U} fpliteral{,fpliteral}...
```

其中，{label}为可选的。

fpliteral 为单精度的浮点数。

使用说明

DCFS 伪操作可能在分配的第一个内存单元前插入填补字节以保证分配的内存是字对齐的。

DCFSU 分配的内存单元则不需要字对齐。

示例

```
DCFS        1E3,-4E-9
DCFSU       1.0,-.1,3.1E6
```

10. DCI

在 ARM 代码中，DCI 用于分配一段字内存单元(分配的内存都是字对齐的)，并用伪操作中的 expr 将其初始化。

在 Thumb 代码中，DCI 用于分配一段半字内存单元(分配的内存都是半字对齐的)，并用伪操作中的 expr 将其初始化。

语法格式

```
{label} DCI expr{,expr} …
```

其中，{label}为可选的。

expr 可以为数字表达式。

使用说明

DCI 伪操作和 DCD 伪操作非常类似，不同之处在于 DCI 分配的内存中数据被标识为指令，可用于通过宏指令来定义处理器指令系统不支持的指令。

在 ARM 代码中，DCI 可能在分配的第一个内存单元前插入最多 3 个字节的填补字节以保证分配的内存是字对齐的。在 Thumb 代码中，DCI 可能在分配的第一个内存单元前插入 1 个字节的填补字节以保证分配的内存是半字对齐的。

示例

```
MACRO                          ; 这个宏指令将指令 newinstr Rd,Rm 定义为相应的机器指令
newinst  $Rd,$Rm
DCI  0xe16f0f10 :OR: ($Rd:SHL:12) :OR: $Rm  ; 这里存放的是指令 MEND
```

11. DCQ 及 DCQU

DCQ 用于分配一段以 8 个字节为单位的内存(分配的内存都是字对齐的)，并用伪操作中的 literal 初始化。

DCQU 与 DCQ 的不同之处在于 DCQU 分配的内存单元并不严格字对齐。

语法格式

```
{label} DCQ{U} {-}literal{,{-}literal}...
```

其中，{label}为可选的。

literal 为 64 位的数字表达式。其取值范围为 0～2^{64}-1。当在 literal 前加上“-”时 literal 的取值范围为-2^{63}～-1。在内存中，2^{64}-n 与-n 具有相同的表达形式。

使用说明

DCQ 伪操作可能在分配的第一个内存单元前插入多达 3 个字节的填补字节以保证分配的内存是字对齐的。

DCQU 分配的内存单元则不需要字对齐。

示例

```
AREA MiscData, DATA, READWRITE
data    DCQ -225,2_101        ; 2_101 指的是二进制的 101
DCQU number+4                 ; number 必须是已经定义过的数字表达式
```

12. DCW 及 DCWU

DCW 用于分配一段半字内存单元(分配的内存都是半字对齐的)，并用伪操作中的 expr 初始化。

DCWU 与 DCW 的不同之处在于 DCWU 分配的内存单元并不严格半字对齐。

语法格式

```
{label} DCW expr{,expr} …
```

其中，{label}为可选的。

expr 为数字表达式，其取值范围为-32768～65535。

使用说明

DCW 伪操作可能在分配的第一个内存单元前插入 1 字节的填补字节以保证分配的内存是半字对齐的。

DCWU 分配的内存单元则不需要半字对齐。

示例

```
data1    DCW -235,num1+8
```

4.1.3　汇编控制伪操作

汇编控制(Assembly control)伪操作包括下面的伪操作。

- IF，ELSE 及 ENDIF
- WHILE 及 WEND
- MACRO 及 MEND
- MEXIT

1. IF，ELSE 及 ENDIF

IF，ELSE 及 ENDIF 伪操作能够根据条件把一段源代码包括在汇编语言程序内或者将其排除在程序之外。[是 IF 伪操作的同义词，| 是 ELSE 伪操作的同义词，] 是 ENDIF 伪操作的同义词。

语法格式

```
IF logical expression
instructions or derectives
{ELSE
instructions or derectives
      }
       ENDIF
```

其中，ELSE 伪操作为可选的。

使用说明

IF，ELSE 及 ENDIF 伪操作可以嵌套使用。

示例

```
IF Version = "1.0"
; 指令
; 伪指令
ELSE
; 指令
; 伪指令
ENDIF
```

2. WHILE 及 WEND

WHILE 及 WEND 伪操作能够根据条件重复汇编相同的或者几乎相同的一段源代码。

语法格式

```
WHILE logical expression
instructions or derectives
        WEND
```

使用说明

IF，ELSE 及 ENDIF 伪操作可以嵌套使用。

示例

```
count   SETA    1          ; 设置循环计数变量 count 初始值为 1
WHILE count <= 4           ; 由 count 控制循环执行的次数
count   SETA count+1       ; 将循环计数变量加 1
; code                     ; 代码
WEND
```

3. MACRO 及 MEND

MACRO 伪操作标识宏定义的开始，MEND 标识宏定义的结束。用 MACRO 及 MEND 定义一段代码，称为宏定义体，这样在程序中就可以通过宏指令多次调用该代码段。

语法格式

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

其中：

$label 在宏指令被展开时，label 可被替换成相应的符号，通常是一个标号。在一个符号前使用$表示程序被汇编时将使用相应的值来替代$后的符号。

Macroname 为所定义的宏的名称。

$parameter 为宏指令的参数。当宏指令被展开时将被替换成相应的值，类似于函数中的形式参数。可以在宏定义时为参数指定相应的默认值。

使用说明

使用子程序可以节省存储空间及程序设计所花的时间，可以提供模块化的程序设计，可以使程序的调试和维护简单。但是，使用子程序也有一些缺点，例如，使用子程序时要保存和恢复相关的寄存器及子程序现场，这些增加了额外的开销。在子程序比较短，而需要传递的参数比较多的情况下可以使用宏汇编技术。

首先使用 MACRO 和 MEND 等伪操作定义宏。包含在 MACRO...MEND 之间的代码

段称为宏定义体。在 MACRO 伪操作之后的一行声明宏的原型，其中包含了该宏定义的名称，需要的参数。在汇编程序中可以通过该宏定义的名称来调用它。当源程序被汇编时，汇编编译器将展开每个宏调用，用宏定义体代替源程序中的宏定义的名称，并用实际的参数值代替宏定义时的形式参数。

宏定义中的$label 是一个可选参数。当宏定义体中用到多个标号时，可以使用类似$label.$internallabel 的标号命名规则使程序易读。下面的例 1 说明了这种用法。

对于 ARM 程序中的局部变量来说，如果该变量在宏定义中被定义，在其作用范围即为该宏定义体。

宏定义可以嵌套。

示例

例 1：　在下面的例子中，宏定义体包括两个循环操作和一个子程序调用。

```
MACRO                          ; 宏定义开始
$label      xmac $p1,$p2       ; 宏的名称为 xmac，有两个参数$p1，$p2
                               ; 宏的标号$label 可用于构造宏定义体内的
                               ; 其他标号名称
; code
$label.loop1    ; code         ; $label.loop1 为宏定义体的内部标号
; code
BGE $label.loop1
$label.loop2    ; code         ; $label.loop2 为宏定义体的内部标号
BL $p1                         ; 参数$p1 为一个子程序的名称
BGT $label.loop2
; code
ADR $p2
; code
MEND                           ; 宏定义结束

; 在程序中调用该宏
abc xmac subr1,de              ; 通过宏的名称 xmac 调用宏，其中宏的标号为 abc，
                               ; 参数 1 为 subr1，参数 2 为 de

; 程序被汇编后，宏展开的结果
; code
abcloop1 ; code                ; 用标号$label 实际值 abc 代替$label 构成标号
                               ; abcloop1
; code
BGE abcloop1
abcloop2 ; code
BL subr1                       ; 参数 1 的实际值为 subr1
BGT abcloop2
; code
ADR de                         ; 参数 2 的实际值为 de
```

例 2：　在 ARM 中完成测试-跳转操作需要两条指令，下面定义一条宏指令完成测试-跳转操作。

```
MACRO                                  ; 宏定义开始
```

```
$label  TestAndBranch $dest, $reg, $cc ; 宏的名称为 TestAndBranch ，有 3
                                       ; 个参数$dest,$reg,$cc: $dest
                                       ; 为跳转的目标地址，$reg 为测试的
                                       ; 寄存器，$cc 为测试的条件。宏的标
                                       ; 号$label 可用于构造宏定义体内的
                                       ; 其他标号名称
$label  CMP $reg, #0
B$cc $dest
MEND

; 在程序中调用该宏
test    TestAndBranch NonZero, r0, NE  ; 通过宏的名称 TestAndBranch 调用
                                       ; 宏，其中宏的标号为 test，参数 1
                                       ; 为 NonZero，参数 2 为 r0，参数 3
                                       ; 为 NE
...
...
NonZero

; 程序被汇编后，宏展开的结果
test    CMP r0, #0
BNE NonZero
...
...
NonZero
```

4. MEXIT

MEXIT 用于从宏中跳转出去。

语法格式

```
MEXIT
```

使用说明

IF，ELSE 及 ENDIF 伪操作可以嵌套使用。

示例

```
MACRO
$abc        macroabc $param1,$param2
; code
WHILE condition1
; code
IF condition2
; code
MEXIT                ; 从宏中跳转出去
ELSE
; code
ENDIF
WEND
; code
```

```
MEND
```

4.1.4　栈中数据帧描述伪操作

栈中数据帧描述伪操作主要用于调试，这里不介绍这部分内容。感兴趣的读者可以参考 ARM 的相关资料。

4.1.5　信息报告伪操作

信息报告(Reporting)伪操作包括伪操作。

- ASSERT
- INFO
- OPT
- TTL 及 SUBT

1. ASSERT

在汇编编译器对汇编程序的第二遍扫描中，如果其中 ASSERTION 中条件不成立，ASSERT 伪操作将报告该错误信息。

语法格式

```
ASSERT logical expression
```

其中，logical expression 为一个逻辑表达式。

使用说明

ASSERT 伪操作用于保证源程序被汇编时满足相关的条件，如果条件不满足，ASSERT 伪操作报告错误类型，并终止汇编。

示例

```
ASSERT          ; 测试 a>0x10 条件是否满足
```

2. INFO

INFO 伪操作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：

string-expression 为一个串表达式。

numeric-expression 为一个数字表达式。如果 numeric-expression 的值为 0，则在汇编处理中，第二遍扫描时，伪操作打印 string-expression；如果 numeric-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。

使用说明

INFO 伪操作用于用户自定义的错误信息。

示例

```
INFO     0, "Version 1.0"              ; 在第二遍扫描时，报告版本信息
IF endofdata <= label1
INFO 4, "Data overrun at label1"       ; 如果 endofdata <= label1 成立在第一遍扫
                                       ; 描时报告错误信息，并终止汇编
ENDIF
```

3. OPT

通过 OPT 伪操作可以在源程序中设置列表选项。

语法格式

```
OPT  n
```

其中，n 为所设置的选项的编码。具体含义如表 4.1 所示。

表 4.1 OPT 伪操作选项的编码

选项编码 n	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
8192	设置选项，不显示条件汇编伪操作
16384	设置选项，显示 MEND 伪操作
32768	设置选项，不显示 MEND 伪操作

使用说明

使用编译选项-list 将使编译器产生列表文件。

默认情况下，-list 选项生成常规的列表文件，包括变量声明、宏展开、条件汇编伪操作以及 MEND 伪操作，而且列表文件只是在第二遍扫描时给出。通过 OPT 伪操作，可以在源程序中改变默认的选项。

示例

在 func1 前插入 OPT 4 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 ; places a page break before func1
func1   ; code
```

4. TTL 及 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。

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 伪操作将作用在其后的每一页，直到遇到新的 SUBT 伪操作。

语法格式

```
TTL title
SUB subtitle
```

其中，title 为标题。subtitle 为子标题。

使用说明

TTL 伪操作在列表文件的页顶部显示一个标题。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。

当使用 TTL 伪操作改变页标题时，新的标题将在下一页开始起作用。

SUBT 伪操作在列表文件的页标题的下面显示一个子标题。如果要在列表文件的第一页显示子标题，SUBT 伪操作要放在源程序的第一行。

当使用 SUBT 伪操作改变页标题时，新的标题将在下一页开始起作用。

示例

```
TTL First Title          ; 在列表文件的第一页及后面的各页显示标题
SUBT First Subtitle      ; 在列表文件的第二页及后面的各页显示标题
```

4.1.6　其他的伪操作

这些杂类的伪操作包括：

- ALIGN
- AREA
- CODE16 及 CODE32
- END
- ENTRY
- EQU
- EXPORT 或 GLOBAL

- EXTERN
- GET 或 INCLUDE
- IMPORT
- INCBIN
- KEEP
- NOFP
- REQUIRE
- REQUIRE8 及 PRESERVE8
- RN
- ROUT

1. CODE16 及 CODE32

CODE16 伪操作告诉汇编编译器后面的指令序列为 16 位的 Thumb 指令。
CODE32 伪操作告诉汇编编译器后面的指令序列为 32 位的 ARM 指令。

语法格式

```
CODE16
CODE32
```

使用说明

当汇编源程序中同时包含 ARM 指令和 Thumb 指令时，使用 CODE16 伪操作告诉汇编编译器后面的指令序列为 16 位的 Thumb 指令；使用 CODE32 伪操作告诉汇编编译器后面的指令序列为 32 位的 ARM 指令。但是，CODE16 伪操作和 CODE32 伪操作只是告诉编译器后面指令的类型，该伪操作本身并不进行程序状态的切换。

示例

在下面的例子中，程序先在 ARM 状态下执行，然后通过 BX 指令切换到 Thumb 状态，并跳转到相应的 Thumb 指令处执行。在 Thumb 程序入口处用 CODE16 伪操作标识下面的指令为 Thumb 指令。

```
AREA ChangeState, CODE, READONLY
CODE32                    ; 指示下面的指令为 ARM 指令
LDR r0,=start+1
BX r0                     ; 切换到 Thumb 状态，并跳转到 start 处执行

CODE16                    ; 指示下面的指令为 Thumb 指令
start MOV r1,#10
```

2. EQU

EQU 伪操作为数字常量、基于寄存器的值和程序中的标号(基于 PC 的值)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 为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。

name 为 EQU 伪操作为 expr 定义的字符名称。

Type 当 expr 为 32 位常量时，可以使用 type 指示 expr 表示的数据的类型。Type 有下面 3 种取值。

- CODE16
- CODE32
- DATA

使用说明

EQU 伪操作的作用类似于 C 语言中的#define，用于为一个常量定义字符名称。

示例

```
abcd EQU 2                ; 定义 abcd 符号的值为 2
abcd EQU label+16         ; 定义 abcd 符号的值(label+16)
addr1 EQU 0x1C, CODE32  ; 定义 addr1 符号值为绝对地址值 0x1C，而且该处为 ARM 指令
```

3. AREA

AREA 伪操作用于定义一个代码段或者数据段。

语法格式

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

其中，sectionname 为所定义的代码段或者数据段的名称。如果该名称是以数字开头的，则该名称必须用“|”括起来，如|1_datasec|。还有一些代码段具有约定的名称，如|.text|表示 C 语言编译器产生的代码段或者是与 C 语言库相关的代码段。

Attr 是该代码段(或者程序段)的属性。在 AREA 伪操作中，各属性间用逗号隔开。下面列举所有的可能的属性：

- ALIGN=expression。默认的情况下，ELF 的代码段和数据段是 4 字节对齐的。Expression 可以取 0～31 的数值，相应的对齐方式为($2^{expression}$)字节对齐。如 expression=3 时为 8 字节对齐。
- ASSOC=section。指定与本段相关的 ELF 段。任何时候连接 section 段也必须包括 sectionname 段。
- CODE 定义代码段。默认属性为 READONLY。
- COMDEF 定义一个通用的段。该段可以包含代码或者数据。在个源文件中，同名的 COMDEF 段必须相同。
- COMMON 定义一个通用的段。该段不包含任何用户代码和数据，连接器将其初始化为 0。各源文件中同名的 COMMON 段公用同样的内存单元，连接器为起分配合适的尺寸。

- DATA 定义数据段。默认属性为 READWRITE。
- NOINIT 指定本数据段仅仅保留了内存单元，而没有将各初始值写入内存单元，或者将个内存单元值初始化为 0。
- READONLY 指定本段为只读，代码段的默认属性为 READONLY。
- READWRITE 指定本段为可读可写，数据段的默认属性为 READWRITE。

使用说明

通常可以用 AREA 伪操作将程序分为多个 ELF 格式的段。段名称可以相同，这时这些同名的段被放在同一个 ELF 段中。

一个大的程序可以包括多个代码段和数据段。一个汇编程序至少包含一个段。

示例

下面的伪操作定义了一个代码段，代码段的名称为 Example，属性为 READONLY。

```
AREA Example,CODE,READONLY
; code
```

4. ENTRY

ENTRY 伪操作指定程序的入口点。

语法格式

```
ENTRY
```

使用说明

一个程序(可以包含多个源文件)中至少要有一个 ENTRY(可以有多个 ENTRY)，但一个源文件中最多只能有一个 ENTRY(可以没有 ENTRY)。

示例

```
AREA example CODE, READONLY
ENTRY                          ; 应用程序的入口点
```

5. END

END 伪操作告诉编译器已经到了源程序结尾。

语法格式

```
END
```

使用说明

每一个汇编源程序都包含 END 伪操作，以告诉本源程序的结束。

示例

```
AREA example CODE, READONLY
...
END
```

6. ALIGN

ALIGN 伪操作通过添加补丁字节使当前位置满足一定的对齐方式。

语法格式

```
ALIGN {expr{,offset}}
```

其中，expr 为数字表达式，用于指定对齐方式。可能的取值为 2 的次幂，如 1、2、4、8 等。如果伪操作中没有指定 expr，则当前位置对齐到下一个字边界处。offset 为数字表达式。当前位置对齐到下面形式的地址处：offset+n*expr。

使用说明

下面的情况中，需要特定的地址对齐方式：

- Thumb 的宏指令 ADR 要求地址是字对齐的，而 Thumb 代码中地址标号可能不是字对齐的。这时就要使用伪操作 ALIGN 4 使 Thumb 代码中的地址标号字对齐。
- 由于有些 ARM 处理器的 CACHE 采用了其他对齐方式，如 16 字节的对齐方式，这时使用 ALIGN 伪操作指定合适的对齐方式可以充分发挥该 CACHE 的性能优势。
- LDRD 及 STRD 指令要求内存单元是 8 字节对齐的。这样在为 LDRD/STRD 指令分配的内存单元前要使用 ALIGN 8 实现 8 字节对齐方式。
- 地址标号通常自身没有对齐要求。而在 ARM 代码中要求地址标号是字对齐的，在 Thumb 代码中要求字节对齐。这样需要使用合适的 ALIGN 伪操作来调整对齐方式。

示例

例 1：　在 AREA 伪操作中的 ALIGN 与 ALIGN 伪操作中 expr 含义是不同的。

```
AREA cacheable, CODE, ALIGN=3   ; 指定下面的指令是 8 字节对齐的
rout1       ; code
; code
MOV pc,lr                       ; 程序跳转后变成 4 字节对齐的
ALIGN 8                         ; 指定下面的指令只 8 字节对齐的
rout2       ; code
```

例 2：　将两个字节数据放在同一个字的第一个字节和第四个字节中。

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

例 3：　在下面的例子中通过 ALIGN 伪操作使程序中地址标号字对齐。

```
AREA Example, CODE, READONLY
start   LDR r6,=label1
; code
MOV pc,lr
```

```
label1  DCB 1          ; 本伪操作使字对齐被破坏
ALIGN                  ; 重新使数据字对齐
subroutine1
MOV r5,#0x5
```

7. EXPORT 及 GLOBAL

EXPRORT 声明一个符号可以被其他文件引用。相当于声明了一个全局变量。GLOBAL 是 EXPORT 的同义词。

语法格式

```
EXPORT symbol{[WEAK]}
```

其中，symbol 为声明的符号的名称。它是区分大小写的。

[WEAK]选项声明其他的同名符号优先于本符号被引用。

使用说明

使用 EXPORT 伪操作声明一个源文件中的符号，使得该符号可以被其他源文件引用。

示例

```
AREA  Example,CODE,READONLY
EXPORT  DoAdd                ; 下面的函数名称 DoAdd 可以被其他源文件引用
DoAdd  ADD r0,r0,r1
```

8. IMPORT

IMPORT 伪操作告诉编译器当前的符号不是在本源文件中定义的，而是在其他源文件中定义的，在本源文件中可能引用该符号，而且不论本源文件是否实际引用该符号，该符号都将被加入到本源文件的符号表中。

语法格式

```
IMPORT symbol{[WEAK]}
```

其中，symbol 为声明的符号的名称。它是区分大小写的。

[WEAK] 指定这个选项后，如果 symbol 在所有的源文件中都没有被定义，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。

使用说明

使用 IMPORT 伪操作声明一个符号是在其他源文件中定义的。如果连接器在连接处理时不能解析该符号，而 IMPORT 伪操作中没有指定[WEAK]选项，则连接器将会报告错误。如果连接器在连接处理时不能解析该符号，而 IMPORT 伪操作中指定了[WEAK]选项，则连接器将不会报告错误，而是进行下面的操作：

- 如果该符号被 B 或者 BL 指令引用，则该符号被设置成下一条指令的地址，该 B 或者 BL 指令相当于一条 NOP 指令。
- 其他情况下该符号被设置为 0。

9. EXTERN

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

语法格式

```
EXTERN symbol{[WEAK]}
```

其中，symbol 为声明的符号的名称。它是区分大小写的。

[WEAK]　指定该选项后，如果 symbol 在所有的源文件中都没有被定义，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。

使用说明

使用 EXTERN 伪操作声明一个符号是在其他源文件中定义的。如果连接器在连接处理时不能解析该符号，而 EXTERN 伪操作中没有指定[WEAK]选项，则连接器将会报告错误。如果连接器在连接处理时不能解析该符号，而 EXTERN 伪操作中指定了[WEAK]选项，则连接器将不会报告错误，而是进行下面的操作：

- 如果该符号被 B 或者 BL 指令引用，则该符号被设置成下一条指令的地址，该 B 或者 BL 指令相当于一条 NOP 指令。
- 其他情况下该符号被设置为 0。

示例

下面的代码测试是否连接了 C++库，并根据结果执行不同的代码

```
AREA    Example, CODE, READONLY
EXTERN  __CPP_INITIALIZE[WEAK]   ; 如果连接了 C++库则读取函数__CPP_INITIALIZE
                                 ; 地址
LDR  r0,__CPP_INITIALIZE
CMP  r0,#0 ; Test if zero.
BEQ  nocplusplus                 ; 如果没有连接 C++库，则跳转到 nocplusplus
```

10. GET 及 INCLUDE

GET 伪操作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 是 GET 的同义词。

语法格式

```
GET filename
```

其中，filename 为被包含的源文件的名称。这里可以使用路径信息。

使用说明

通常可以在一个源文件中定义宏，用 EQU 定义常量的符号名称，用 MAP 和 FIELD 定义结构化的数据类型，这样的源文件类似于 C 语言中的.H 文件。然后用 GET 伪操作将这个源文件包含到它们的源文件中，类似于在 C 源程序的“include *.h”。

编译器通常在当前目录中查找被包含的源文件。可以使用编译选项-I 添加其他的查找目录。同时，被包含的源文件中也可以使用 GET 伪操作，即 GET 伪操作可以嵌套使用。如在源文件 A 中包含了源文件 B，而在源文件 B 中包含了源文件 C。编译器在查找 C 源文件时将把源文件 B 所在的目录作为当前目录。

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。

示例

```
AREA Example, CODE, READONLY
GET file1.s                        ; 包含源文件 file1.s
GET c:\project\file2.s             ; 包含源文件 file2.s,可以包含路径信息
GET c:\Program files\file3.s       ; 包含源文件 file3.s，路径信息中可以包含空格
```

11. INCBIN

INCBIN 伪操作将一个文件包含到(INCLUDE)当前源文件中，被包含的文件不进行汇编处理。

语法格式

```
INCBIN filename
```

其中，filename 为被包含的文件的名称。这里可以使用路径信息。

使用说明

通常可以使用 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地放到当前文件中。编译器从 INCBIN 伪操作后面开始继续处理。

编译器通常在当前目录中查找被包含的源文件。可以使用编译选项-I 添加其他的查找目录。同时，被包含的源文件中也可以使用 GET 伪操作，即 GET 伪操作可以嵌套使用。如在源文件 A 中包含了源文件 B，而在源文件 B 中包含了源文件 C。编译器在查找 C 源文件时将把源文件 B 所在的目录作为当前目录。

这里所包含的文件名称及其路径信息中都不能有空格。

示例

```
AREA Example, CODE, READONLY
INCBIN file1.dat                   ; 包含文件 file1.dat
INCBIN c:\project\file2.txt        ; 包含文件 file2.txt
```

12. KEEP

KEEP 伪操作告诉编译器将局部符号包含在目标文件的符号表中。

语法格式

```
KEEP {symbol}
```

其中，symbol 为被包含在目标文件的符号表中的符号。如果没有指定 symbol，则除

了基于寄存器外的所有符号将被包含在目标文件的符号表中。

使用说明

默认情况下，编译器仅将下面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。

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

示例

```
label1  ADC r2,r3,r4
KEEP label1          ; 将标号 label1 包含到目标文件的符号表中
ADD r2,r2,r5
```

13. NOFP

使用 NOFP 伪操作禁止源程序中包含浮点运算指令。

语法格式

```
NOFP
```

使用说明

当系统中没有硬件或软件仿真代码支持浮点运算指令时，使用 NOFP 伪操作禁止在源程序中使用浮点运算指令。这时如果源程序中包含浮点运算指令，编译器将会报告错误。同样如果在浮点运算指令的后面使用 NOFP 伪操作，编译器同样将会报告错误。

14. REQUIRE

REQUIRE 伪操作指定段之间的相互依赖关系。

语法格式

```
REQUIRE lable
```

其中，label 为所需要的标号的名称。

使用说明

当进行连接处理时包含了有 REQUIRE lable 伪操作的源文件，则定义 label 的源文件也将被包含。

15. REQUIRE8 及 PRESERVE8

REQUIRE8 伪操作指示当前代码中要求数据栈 8 字节对齐。

PRESERVE8 伪操作指示当前代码中数据栈是 8 字节对齐的。

语法格式

```
REQUIRE8
PRESERVE8
```

使用说明

LDRD 及 STRD 指令要求内存单元地址是 8 字节对齐的。当在程序中使用这些指令在数据栈中传送数据时，要求该数据栈是 8 字节对齐的。

连接器要保证要求 8 字节对齐的数据栈代码只能被数据栈是 8 字节对齐的代码调用。

16. RN

RN 伪操作为一个特定的寄存器定义名称。

语法格式

```
name  RN expr
```

其中，expr 为某个寄存器的编码。

name 为本伪操作给寄存器 expr 定义的名称。

使用说明

RN 伪操作用于给一个寄存器定义名称。方便程序员记忆该寄存器的功能。

17. ROUT

ROUT 伪操作用于定义局部变量的有效范围。

语法格式

```
{name}ROUT
```

其中，name 为所定义的作用范围的名称。

使用说明

当没有使用 ROUT 伪操作定义局部变量的作用范围时，局部变量的作用范围为其所在的段(AREA)。ROUT 伪操作作用的范围为本 ROUT 伪操作和下一个 ROUT(指同一个段中的 ROUT 伪操作)伪操作之间。

4.2 ARM 汇编语言伪指令

ARM 中伪指令不是真正的 ARM 指令或者 Thumb 指令，这些伪指令在汇编编译器对源程序进行汇编处理时被替换成对应的 ARM 或者 Thumb 指令(序列)。ARM 伪指令包括 ADR、ADRL、LDR 和 NOP。

1. ADR (小范围的地址读取伪指令)

该指令将基于 PC 的地址值或基于寄存器的地址值读取到寄存器中。

语法格式

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

其中，cond 为可选的指令执行的条件。

register 为目标寄存器。

expr 为基于 PC 或者基于寄存器的地址表达式，其取值范围如下：

- 当地址值不是字对齐时，其取值范围为-255～255。
- 当地址值是字对齐时，其取值范围为-1020～1020。
- 当地址值是 16 字节对齐时，其取值范围将更大。

使用说明

在汇编编译器处理源程序时，ADR 伪指令被编译器替换成一条合适的指令。通常，编译器用一条 ADD 指令或 SUB 指令来实现该 ADR 伪指令的功能。如果不能用一条指令来实现 ADR 伪指令的功能，编译器将报告错误。

因为 ADR 伪指令中的地址是基于 PC 或者基于寄存器的，所以 ADR 读取到的地址为位置无关的地址。当 ADR 伪指令中的地址是基于 PC 时，该地址与 ADR 伪指令必须在同一个代码段中。

示例

```
start MOV r0,#10          ; 因为 PC 值为当前指令地址值加 8 字节
ADR r4,start              ; 本 ADR 伪指令将被编译器替换成 SUB r4,pc,#0xc
```

2. ADRL(中等范围的地址读取伪指令)

该指令将基于 PC 或基于寄存器的地址值读取到寄存器中。ADRL 伪指令比 ADR 伪指令可以读取更大范围的地址。ADRL 伪指令在汇编时被编译器替换成两条指令。

语法格式

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

其中，cond 为可选的指令执行的条件。

register 为目标寄存器。

expr 为基于 PC 或者基于寄存器的地址表达式，其取值范围如下：

- 当地址值不是字对齐时，其取值范围为-64 KB～64 KB。
- 当地址值是字对齐时，其取值范围为-256 KB～256 KB。
- 当地址值是 16 字节对齐时，其取值范围将更大。

使用说明

在汇编编译器处理源程序时，ADRL 伪指令被编译器替换成两条合适的指令，即使一条指令可以完成该伪指令的功能，编译器也将用两条指令来替换该 ADRL 伪指令。如果不能用两条指令来实现 ADRL 伪指令的功能，编译器将报告错误。

示例

```
start   MOV r0,#10        ; 因为 PC 值为当前指令地址值加 8 字节
ADRL r4,start+60000       ; 本 ADRL 伪指令将被编译器替换成下面两条指令
ADD r4,pc,#0xe800
ADD r4,r4,#0x254
```

3. LDR 大范围的地址读取伪指令

LDR 伪指令将一个 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，cond 为可选的指令执行的条件。

register 为目标寄存器。

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：

- 当 expr 表示的地址值没有超过 MOV 或 MVN 指令中地址的取值范围时，编译器用合适的 MOV 或者 MVN 指令代替该 LDR 伪指令。
- 当 expr 表示的地址值超过了 MOV 或 MVN 指令中地址的取值范围时，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 的地址表达式或者是外部表达式。当 Label-expr 为基于 PC 的地址表达式时，编译器将 label-expr 表示的数值放在数据缓冲区中，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。

使用说明

LDR 伪指令主要有以下两种用途：

- 当需要读取到寄存器中的数据超过了 MOV 及 MVN 指令可以操作的范围时，可以使用 LDR 伪指令将该数据读取到寄存器中。
- 将一个基于 PC 的地址值或者外部的地址值读取到寄存器中。由于这种地址值是在连接时确定的，所以这种代码不是位置无关的。同时 LDR 伪指令处的 PC 值到数据缓冲区中的目标数据所在的地址的偏量要小于 4 KB。

示例

例 1：　将 0xff0 读取到 R1 中。

```
LDR R1,=0XFF0
```

汇编后将得到：

```
MOV R1,0XFF0
```

例 2：　将 0xfff 读取到 R1 中。

```
LDR R1,=0XFFF
```

汇编后将得到：

```
LDR R1,[PC,OFFSET_TO_LPOOL]
…
LPOOL  DCD 0XFFF
```

例 3：　将外部地址 ADDR1 读取到 R1 中。

```
LDR R1,=ADDR1
```

汇编后将得到：

```
LDR R1,[PC,OFFSET_TO_LPOOL]
…
LPOOL  DCD ADDR1
```

4. NOP 空操作伪指令

NOP 伪指令在汇编时将被替换成 ARM 中的空操作，比如可能为 MOV R0 和 R0 等。

语法格式

```
NOP
```

使用说明

NOP 伪指令不影响 CPSR 中的条件标志位。

4.3　ARM 汇编语言语句格式

ARM 汇编语言语句格式如下所示：

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

其中，instruction 为指令。在 ARM 汇编语言中，指令不能从一行的行头开始。在一行语句中，指令的前面必须有空格或者符号。

directive 为伪操作。

pseudo-instruction 为伪指令。

symbol 为符号。在 ARM 汇编语言中，符号必须从一行的行头开始，并且符号中不能包含空格。在指令和伪指令中符号用作地址标号(label)；在有些伪操作中，符号用作变量或者常量。

comment 为语句的注释。在 ARM 汇编语言中注释以分号(；)开头。注释的结尾即为一行的结尾。注释也可以单独占用一行。

在 ARM 汇编语言中，各个指令、伪指令及伪操作的助记符必须全部用大写字母，或者全部用小写字母，不能在一个伪操作助记符中既有大写字母又有小写字母。

源程序中，语句之间可以插入空行，以源代码的可读性更好。

如果一条语句很长，为了提高可读性，可以将该长语句分成若干行来写。这时在一行的末尾用“\”表示下一行将续在本行之后。注意，在“\”之后不能再有其他字符，空格和制表符也不能有。

4.3.1　ARM 汇编语言中的符号

在 ARM 汇编语言中，符号(symbols)可以代表地址(addresses)、变量(variables)和数字常量(numeric constants)。当符号代表地址时又称为标号(label)。当标号以数字开头时，其作用范围为当前段(当没有使用 ROUT 伪操作时)，这种标号又称为局部标号(local lable)。符号包括变量、数字常量、标号和局部标号。

符号的命名规则如下：

- 符号由大小写字母、数字以及下划线组成。
- 局部标号以数字开头，其他的符号都不能以数字开头。
- 符号是区分大小写的。
- 符号中的所有字符都是有意义的。
- 符号在其作用范围内必须惟一，即在其作用范围内不可有同名的符号。
- 程序中的符号不能与系统内部变量或者系统预定义的符号同名。
- 程序中的符号通常不要与指令助记符或者伪操作同名。当程序中的符号与指令助记符或者伪操作同名时，用双竖线将符号括起来，如||require||，这时双竖线并不是符号的组成部分。

1. 变量

程序中变量的值在汇编处理过程中可能会发生变化。在 ARM 汇编语言中变量有数字变量、逻辑变量和串变量 3 种类型。变量的类型在程序中是不能改变的。

数字变量的取值范围为数字常量和数字表达式所能表示的数值的范围。关于数字常量和数字表达式在后面有介绍。

逻辑变量的取值范围为{true}及{false}。

串变量的取值范围为串表达式可以表示的范围。

在 ARM 汇编语言中，使用 GBLA、GBLL 及 GBLS 声明全局变量；使用 LCLA、LCLL 及 LCLS 声明局部变量；使用 SETA、SETL 及 SETS 为这些变量赋值。

2. 数字常量

数字常量是 32 位的整数。当作为无符号整数时，其取值范围为 0 到 $2^{32}-1$；当作为有符号整数时，其取值范围为 -2^{31}～$2^{31}-1$。汇编编译器并不区分一个数是无符号的还是有符号的，事实上-n 与 2^{32}-n 在内存中是同一个数。

进行大小比较时，认为数字常量都是无符号数。按照这种规则有：0<-1。

在 ARM 汇编语言中，使用 EQU 来定义数字常量。数字常量一经定义，其数值就不能再修改。

3. 汇编时的变量替换

如果在串变量前面有一个$字符，在汇编时编译器将用该串变量的数值取代该串变量。

例 1： 如果 STR1 的值为 pen.，则汇编后 STR2 值为 This is a pen.。

```
GBLS STR1
GBLS STR2
STR1   SETS  "pen"
STR2   SETS  "This is a $STR1"
```

对于数字变量来说，如果该变量前面有一个 $ 字符，在汇编时编译器将该数字变量的数值转换成十六进制的串，然后用该十六进制的串取代 $ 字符后的数字变量。

对于逻辑变量来说，如果该逻辑变量前面有一个 $ 字符，在汇编时编译器将该逻辑

变量替换成它的取值(T 或者 F)。

如果程序中需要字符 $，则用 $$ 来表示，编译器将不进行变量替换，而是将 $$ 当作$。

例 2：　本例说明数字变量的替换和$$的用法。汇编后得到 STR1 值为 abcB0000000E。

```
        GBLS  STR1
        GBLS  B
        GBLA  NUM1
NUM1    SETA   14
B       SETS   "CHANGED"
STR1    SETS    "abc$$B$NUM1"
```

通常情况下，包含在两个竖线(|)之间的$并不表示进行变量替换。但是如果竖线(|)是在双引号内，则将进行变量替换。

使用“.”来表示变量名称的结束。

例 3：　本例说明使用“.”来分割出变量名的用法。汇编后 STR2 值为 bbbAAACCC。

```
            GBLS STR1
            GBLS STR2
    STR1    SETS  "AAA"
STR2        SETS  "bbb$STR1.CCC"
```

4. 标号

标号是表示程序中的指令或者数据地址的符号。根据标号的生成方式可以有以下 3 种：

- 基于 PC 的标号

  基于 PC 的标号是位于目标指令前或者程序中数据定义伪操作前的标号。这种标号在汇编时将被处理成 PC 值加上(或减去)一个数字常量。它常用于表示跳转指令的目标地址，或者代码段中所嵌入的少量数据。

- 基于寄存器的标号

  基于寄存器的标号通常用 MAP 和 FILED 伪操作定义，也可以用 EQU 伪操作定义。这种标号在汇编时将被处理成寄存器的值加上(或减去)一个数字常量。它常用于访问位于数据段中的数据。

- 绝对地址

  绝对地址是一个 32 位的数字量。它可以寻址的范围为 $0\sim2^{32}-1$，即直接可以寻址整个内存空间。

5. 局部标号

局部标号主要用于在局部范围使用。它由两部分组成：开头是一个 0～99 之间的数字；后面紧接一个通常表示该局部变量作用范围的符号。

局部变量的作用范围通常为当前段，也可用伪操作 ROUT 来定义局部变量的作用范围。

局部变量定义的语法格式如下。

```
N{routname}
```

其中，N为0～99之间的数字。

routname 为符号，通常为该变量作用范围的名称(用 ROUT 伪操作定义的)。

局部变量引用的语法格式如下：

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

其中，N 为局部变量的数字号。

routname 为当前作用范围的名称(用 ROUT 伪操作定义的)。

%表示引用操作。

F 指示编译器只向前搜索。

B 指示编译器只向后搜索。

A 指示编译器搜索宏的所有嵌套层次。

T 指示编译器搜索宏的当前层次。

如果 F 和 B 都没有指定，编译器先向前搜索，再向后搜索。

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

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

4.3.2 ARM 汇编语言中的表达式

表达式是有符号、数值、单目或多目操作符以及括号组成的。在一个表达式各种元素的优先级如下所示：

- 括号内的表达式优先级最高。
- 各种操作符有一定的优先级。
- 相邻的单目操作符的执行顺序为由右到左，单目操作符优先级高于其他操作符。
- 优先级相同的双目操作符执行顺序为由左到右。

下面分别介绍表达式中的各元素。

1. 字符串表达式

字符串表达式由字符串、字符串变量、操作符以及括号组成。字符串的最大长度为 512 字节，最小长度为 0。下面介绍字符串表达式的组成元素。

- 字符串

  字符串由包含在双引号内的一系列的字符组成。字符串的长度受到 ARM 汇编语言语句长度的限制。

  当在字符串中包含美元符号$或者引号"时，用$$表示一个$，用""表示一个"。

例 1： 本例说明字符串中包含$及"的方法。

```
abc SETS "this string contains only one "" double quote"
def SETS "this string contains only one $$ dollar symbol"
```

- 字符串变量

  字符串变量用伪操作 GBLS 或者 LCLS 声明，用 SETS 赋值。取值范围与字符表

达式相同。

- 操作符

  与字符串表达式相关的操作符有下面一些。

  - LEN

    LEN 操作符返回字符串的长度。其语法格式如下：

    ```
    : LEN: A
    ```

    其中，A 为字符串变量。

  - CHR

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

    ```
    : CHR: A
    ```

    其中，A 为某一字符的 ASCII 值。

  - STR

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

    ```
    : STR: A
    ```

    其中，A 为数字量或者逻辑表达式。

  - LEFT

    LEFT 返回一个字符串最左端一定长度的子串。其语法格式如下：

    ```
    A: LEFT: B
    ```

    其中，A 为源字符串。

    B 为数字量，表示 LEFT 将返回的字符个数。

  - RIGHT

    RIGHT 返回一个字符串最右端一定长度的子串。其语法格式如下：

    ```
    A: RIGHT: B
    ```

    其中，A 为源字符串。

    B 为数字量，表示 RIGHT 将返回的字符个数。

  - CC

    CC 用于连接两个字符串。其语法格式如下：

    ```
    A: CC: B
    ```

    其中，A 为第 1 个源字符串。

    B 为第 2 个源字符串。CC 操作符将字符串 B 连接在字符串 A 的后面。

- 字符变量的声明和赋值

字符变量的声明使用 GBLS 或者 LCLS 伪操作。

字符变量的赋值使用 SETS 伪操作。

- 字符串表达式应用举例

```
GBLS  STRING1                                    ; 声明字符串变量 STRING1
GBLS  STRING2                                    ; 声明字符串变量 STRING2
STRING1 SETS"AAACCC"                             ; 变量 STRING1 赋值为"AAACCC"
STRING2 SETS"BB": CC: (STRING2: LEFT: 3);  为变量 STRING2 赋值
                                                 ; 为变量 STRING2 值为"BBAAA"
```

2. 数字表达式

数字表达式由数字常量、数字变量、操作符和括号组成。

数字表达式表示的是一个 32 位的整数。当作为无符号整数时，其取值范围为 0～2^{32}–1；当作为有符号整数时，其取值范围为–2^{31}～2^{31}–1。汇编编译器并不区分一个数是无符号的还是有符号的，事实上–n 与 2^{32}–n 在内存中是同一个数。

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。

- 整数数字量

  在 ARM 汇编语言中，整数数字量有以下几种格式：

  - decimal-digits　十进制数。
  - 0xhexadecimal-digits　十六进制数。
  - &hexadecimal-digits　十六进制数。
  - n_base-n-digits　n 进制数。

  当使用 DCQ 或者 DCQU 伪操作声明时，该数字量表示的数的范围位 0～2^{64}–1。

  其他情况下数字量表示的数的范围位 0～2^{32}–1。

  例：本例列举一些数字量。

```
a       SETA    34906
addr    DCD 0xA10E
LDR r4,&1000000F
DCD 2_11001010
c3      SETA    8_74007
DCQ 0x0123456789abcdef
```

- 浮点数字量

  浮点数字量有以下几种格式：

  - {-}digits E{-}digits
  - {-}{digits}.digits{E{-}digits}
  - 0xhexdigits
  - &hexdigits

  其中，digits 为十进制的数字，hexdigits 为十六进制的数。

  单精度的浮点数表示范围为：最大值为 3.40282347e+38；最小值为 1.17549435e–38。

  双精度的浮点数表示范围为：最大值为 1.79769313486231571e+308；最小值为 2.22507385850720138e–308。

例：本例列举一些浮点数。

```
DCFD 1E308,-4E-100
DCFS 1.0
DCFD 3.725e15
LDFS 0x7FC00000
LDFD &FFF0000000000000
```

- 数字变量

  数字变量用伪操作 GBLA 或者 LCLA 声明，用 SETA 赋值，它代表一个 32 位的数字量。

- 操作符

  与数字表达式相关的操作符有下面一些。

  - NOT 按位取反

    NOT 将一个数字量按位取反。其语法格式如下：

    ```
    : NOT: A
    ```

    其中，A 为一个 32 位数字量。

  - +、–、×、/及 MOD 算术操作符

    +、–、×、/及 MOD 这些算术运算符含义即语法格式如下。其中，A 和 B 均为数字表达式。

    A+B 表示 A、B 的和。

    A–B 表示 A、B 的差。

    A×B 表示 A、B 的积。

    A/B 表示 A 除以 B 的商。

    A：MOD：B 表示 A 除以 B 的余数。

  - ROL、ROR、SHL 及 SHR 移位(循环移位操作)

    ROL、ROR、SHL 及 SHR 操作符的格式及含义如下。其中，A 和 B 为数字表达式。

    A：ROL：B 将整数 A 循环左移 B 位。

    A：ROR：B 将整数 A 循环右移 B 位。

    A：SHL：B 将整数 A 左移 B 位。

    A：SHR：B 将整数 A 右移 B 位，这里为逻辑右移，不影响符号位。

  - AND、OR 及 EOR 按位逻辑操作符

    AND、OR 及 EOR 逻辑操作符都是按位操作的，其语法格式及含义如下。其中，A 和 B 为数字表达式。

    A：AND：B 将数字表达式 A 和 B 按位作逻辑与操作。

    A：OR：B 将数字表达式 A 和 B 按位作逻辑或操作。

    A：EOR：B 将数字表达式 A 和 B 按位作逻辑异或操作。

3. 基于寄存器和基于 PC 的表达式

基于寄存器的表达式表示了某个寄存器的值加上(或减去)一个数字表达式。

基于 PC 的表达式表示了 PC 寄存器的值加上(或减去)一个数字表达式。基于 PC 的表

达式通常由程序中的标号与一个数字表达式组成。相关的操作符有以下几种：

- BASE

  BASE 操作符返回基于寄存器的表达式中的寄存器编号。其语法格式和含义如下。其中，A 为基于寄存器的表达式。

  ```
  : BASE: A
  ```

- INDEX

  INDEX 操作符返回基于寄存器的表达式相对于其基址寄存器的偏移量。其语法格式和含义如下。其中，A 为基于寄存器的表达式。

  ```
  : INDEX: A
  ```

- +、-

  +、-为正负号。它们可以放在数字表达式或者基于 PC 的表达式前面。其语法格式和含义如下。其中，A 为基于 PC 的表达式或者数字表达式。

  ```
  +A
  -A
  ```

4. 逻辑表达式

逻辑表达式由逻辑量、逻辑操作符、关系操作符以及括号组成。取值范围为{FALSE}和{TURE}。

- 关系操作符

  关系操作符用于表示两个同类表达式之间的关系。关系操作符和它的两个操作数组成一个逻辑表达式，其取值为{FALSE}或{TURE}。

  关系操作符的操作数可以是以下类型。

  - 数字表达式　这里数字表达式均视为无符号数。
  - 字符串表达式　字符串比较时，依据串中对应字符的 ASCII 顺序比较。
  - 基于寄存器的表达式。
  - 基于 PC 的表达式。

  A 和 B 是上述的 4 类表达式之一。下面列出 A 和 B 比较的关系操作符。

  - A=B　表示 A 等于 B。
  - A>B　表示 A 大于 B。
  - A>=B　表示 A 大于或者等于 B。
  - A<B　表示 A 小于 B。
  - A<=B　表示 A 小于或者等于 B。
  - A/=B　表示 A 不等于 B。
  - A<>B　表示 A 不等于 B。

- 逻辑操作符

  逻辑操作符进行两个逻辑表达式之间的基本逻辑操作。操作的结果为{FALSE}或{TURE}。

  A 和 B 是两个逻辑表达式。下面列出各逻辑操作符语法格式及其含义。

- ：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：LEOR：B　逻辑表达式 A 和 B 的逻辑异或。

5. 其他的一些操作符

ARM 汇编语言中的操作符还有下面一些。

- ?

  ? 操作符的语法格式及含义如下，其中 A 为一个符号：

  ```
  ?A
  ```

  返回定义符号 A 的代码行所生成的可执行代码的字节数。

- DEF

  DEF 操作符判断某个符号是否已定义。其语法格式及含义如下，其中 A 为一个符号：

  ```
  : DEF: A
  ```

  如果符号 A 已经定义，上述结果为{TURE}，否则上述结果为{FALSE}。

- SB_OFFSET_19_12

  SB_OFFSET_19_12 语法格式及含义如下，其中 label 为一个标号。

  ```
  : SB_OFFSET_19_12: label
  ```

  返回(label－SB)的 bits[19：12]。

- SB_OFFSET_11_0

  SB_OFFSET_11_0 语法格式及含义如下，其中 label 为一个标号。

  ```
  : SB_OFFSET_11_0: label
  ```

  返回(label－SB)的 bits[11：0]。

4.4　ARM 汇编语言程序格式

本小节介绍 ARM 汇编语言程序的基本格式以及子程序间调用的格式。

4.4.1　汇编语言程序格式

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

ARM 汇编语言源程序经过汇编处理后生成一个可执行的映像文件(类似于 windows 系统下的 EXE 文件)。该可执行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。这些数据段通常是可读写的。
- 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的映像文件中段之间的相邻关系并不一定相同。

下面通过一个简单的例子，说明 ARM 汇编语言源程序的基本结构。

```
AREA EXAMPLE1, CODE, READONLY
ENTRY
start
MOV r0, #10
MOV r1, #3
ADD r0, r0, r1

END
```

在 ARM 汇编语言源程序中，使用伪操作 AREA 定义一个段。AREA 伪操作表示了一个段的开始，同时定义了这个段的名称及相关属性。在本例中定义了一个只读的代码段，其名称为 EXAMPLE1。

ENTRY 伪操作标识了程序执行的第一条指令。一个 ARM 程序中可以有多个 ENTRY，至少要有一个 ENTRY。初始化部分的代码以及异常中断处理程序中都包含了 ENTRY。如果程序包含了 C 代码，C 语言库文件的初始化部分也包含了 ENTRY。

本程序的程序体部分实现了一个简单的加法运算。

END 伪操作告诉汇编编译器源文件的结束。每一个汇编模块必须包含一个 END 伪操作，指示本模块的结束。

4.4.2 汇编语言子程序调用

在 ARM 汇编语言中，子程序调用是通过 BL 指令完成的。BL 指令的语法格式如下：

```
BL subname
```

其中，subname 是调用的子程序的名称。

BL 指令完成两个操作：将子程序的返回地址放在 LR 寄存器中，同时将 PC 寄存器值设置成目标子程序的第一条指令地址。

在子程序返回时可以通过将 LR 寄存器的值传送的 PC 寄存器中来实现。

子程序调用时通常使用寄存器 R0～R3 来传递参数和返回结果，这些在后面编程模型中还会又详细介绍。

下面是一个子程序调用的例子。子程序 DOADD 完成加法运算，操作数放在 R0 和 R1 寄存器中，结构放在 R0 中。

```
AREA EXAMPLE2, CODE, READONLY
ENTRY
start   MOV r0, #10          ; 设置输入参数 R0
MOV r1, #3                   ; 设置输入参数 R1
BL doadd                     ; 调用子程序 doadd
```

```
doadd   ADD r0, r0, r1          ; 子程序
MOV pc, lr                      ; 从子程序中返回
END
```

4.5　ARM 汇编编译器的使用

本节介绍 ARM 汇编编译器 ARMASM。内嵌的 ARM 汇编编译器是 ARM 中 C/C++编译器的一部分，它没有自己的命令行格式。在 ARMASM 命令中，除了文件名区分大小写之外，其他的参数都不区分大小写。

ARMASM 的语法格式如下所示：

```
armasm [-16|-32] [-apcs [none|{/qualifier[/qualifier[...]]]]]
[-bigend|-littleend] [-checkreglist] [-cpu cpu] [-depend dependfile|-m|-
md]
[-errors errorfile] [-fpu name] [-g] [-help] [-i dir [,dir]…] [-keep]
[-list
[listingfile] [options]] [-maxcache n] [-memaccess attributes] [-nocache]
[-noesc] [-noregs] [-nowarn] [-o filename] [-predefine "directive"] [-
split_ldm]
[-unsafe] [-via file] inputfile
```

下面详细介绍 ARMASM 的各参数。

- –16　告诉汇编编译器所处理的源程序是 Thumb 指令的程序。其功能与在源程序开头使用伪操作 CODE16 相同。
- –32　告诉汇编编译器所处理的源程序是 ARM 指令的程序。这是 ARMASM 的默认选项。
- –apcs [none|[/qualifier[/qualifier[...]]]]　用于指定源程序所使用的 ATPCS。使用何种 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，并连接到适当类型的库文件。ATPCS 选项可能的取值有：
  - /none　指定源程序不使用任何 ATPCS。
  - /interwork　指定源程序中有 ARM 指令和 Thumb 指令混合使用。
  - /nointerwork　指定源程序中没有 ARM 指令和 Thumb 指令混合使用。这是 ARMASM 默认的选项。
  - /ropi　指定源程序是 ROPI(只读位置无关)。ARMASM 默认的选项是 /noropi。
  - /pic　是/ropi 的同义词。
  - /nopic　是/noropi 的同义词。
  - /rwpi　指定源程序是 RWPI(读写位置无关)。ARMASM 默认的选项是 /norwpi。
  - /pid　是/rwpi 的同义词。
  - /nopid　是/norwpi 的同义词。

- /swstackcheck 指定源程序进行软件数据栈限制检查。
- /noswstackcheck 指定源程序不进行软件数据栈限制检查，这是 ARMASM 默认的选项。
- /swstna 指定源程序既与进行软件数据栈限制检查的程序兼容，也与不进行软件数据栈限制检查的程序兼容。

- –bigend 告诉 ARMASM 将源程序汇编成适合于 BIG ENDIAN 的模式。
- –littleend 告诉 ARMASM 将源程序汇编成适合于 LITTLE ENDIAN 的模式。这是 ARMASM 默认的选项。
- –checkreglist 告诉 ARMASM 检查指令 RLIST、LDM、STM 中的寄存器列表，保证寄存器列表中的寄存器是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 cpu 告诉 ARMASM 目标 CPU 的类型。合法的取值为 ARM 体系名称，如 3、4T、5TE，或者也可以为 CPU 的类型编号，如 ARM7TDMI 等。
- –depend dependfile 告诉 ARMASM 将源程序的依赖列表(dependency lists)保存到文件 dependfile 中。
- –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 本选项指定目标系统中的浮点运算单元的体系。其可能的取值如下：
  - none 指定没有浮点选项。这样目标程序与所有其他目标程序都是兼容的。
  - vfpv1 指定系统中使用符合 VFPv1 的硬件向量浮点运算单元。
  - vfpv2 指定系统中使用符合 VFPv2 的硬件向量浮点运算单元。
  - fpa 指定系统使用硬件的浮点加速器(Float Point Accelerator)。
  - softvfp+vfp 指定系统中使用硬件向量浮点运算单元。
  - softvfp 指定系统使用软件的浮点运算库，这时使用单一的内存模式(endianess 格式)。这是 ARMASM 默认的选型。
  - 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(endianess 格式)。
- –g 指示 ARMASM 产生 DRAWF2 格式的调试信息表。
- –help 指示 ARMASM 显示本汇编编译器的选项。
- –i dir[,dir]... 添加搜索路径。指定搜索伪操作 GET/INCLUDE 中的变量的范围。
- –keep 指示 ARMASM 将局部符号保留在目标文件的符号表中，供调试器进行调试时使用。
- –list [listingfile] [option] 指示 ARMASM 将其产生的汇编程序列表保存到列表文件 listingfile 中。如果没有指定 listingfile，则保存到文件 inputfile.lst 中。下面一些选项控制列表文件的格式：
  - –noterse 源程序中由于条件汇编被排除的代码也将包含在列表文件中。

  - –width　指定列表文件中每行的宽度，默认为 79 个字符。
  - –length　指定列表文件中每页的行数，默认为 66 行，0 表示不分页。
  - –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 cache 是指 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，从内存中读取该源程序)大小，默认为 8 MB。
- –memaccess attributes　指定目标系统的存储访问模式。默认的情况是允许字节对齐、半字对齐、字对齐的读写访问。可以指定下面的访问属性。
  - +L41　允许非对齐的 LDR 访问。
  - –L22　禁止半字的 LOAD 访问。
  - –S22　禁止半字的 STORE 访问。
  - –L22-S22　禁止半字的 LOAD 访问和 STORE 访问。
- –nocache　禁止源程序 cache。通常情况下，ARMASM 在第一遍扫描时将源程序保存在内存中(称为源程序 cache)，第二遍扫描时从内存中读取该源程序。
- –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　指示 ARMASM 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 和 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 将认为该指令错误。
- –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。

4.6　汇编程序设计举例

在本节中通过一些例子来说明 ARM 中伪操作以及指令的用法。4.6.1 中举了一些伪操作的实例，4.6.2 中是一些 ARM 汇编程序的实例。

4.6.1　ARM 中伪操作使用实例

程序 4.1　ARM 中伪操作的使用实例。

```
; 声明两个字符变量，用以存放两个函数参数
    GBLS    _arg0
    GBLS    _arg1

; 宏_spaces_remove
; 删除全局变量 wstring 开头和结尾的空格
```

```
        MACRO
        _spaces_remove $wstring
        WHILE (  ("*" :CC: $wstring) :RIGHT: 1 = " ")
$wstring SETS ($wstring :LEFT: (:LEN: $wstring - 1))
        WEND
        WHILE ( ($wstring :CC: "*") :LEFT: 1 = " ")
$wstring SETS ($wstring :RIGHT: (:LEN: $wstring - 1))
        WEND
        MEND

; 宏_lbracket_remove
; 删除一起左括号 - 如果不存在左括号则报错
    MACRO
    _lbracket_remove $s
    ASSERT  $s:LEFT:1 = "("
$s  SETS  $s:RIGHT:(:LEN:$s-1)
    _spaces_remove $s
    MEND

; 宏_rbracket_remove
; 删除一起左括号 - 如果不存在左括号则报错
; 然后删除多余的空格
    MACRO
    _rbracket_remove $s
    ASSERT  $s:RIGHT:1 = ")"
$s  SETS    $s:LEFT:(:LEN:$s-1)
    _spaces_remove $s
    MEND

; 宏_comment_remove
; 删除行末的所有注释及空格
    MACRO
    _comment_remove $s
    _spaces_remove $s
    IF ("**":CC:$s):RIGHT:2) = "*/"
      WHILE ($s:RIGHT:2) <> "/*"
$s      SETS $s:LEFT:(:LEN:$s-1)
      WEND
$s  SETS $s:LEFT:(:LEN:$s-2)
    _spaces_remove $s
    ENDIF
    MEND

; 宏_arg_remove
; 从一个用空格分割的串中获取一个变量
    MACRO
    _arg_remove $s,$arg
    LCLA    _arglen
    LCLL    _ok
_arglen SETA    0
_ok SETL    {TRUE}
```

```
    WHILE   _ok
      IF _arglen>=:LEN:$s
_ok     SETL {FALSE}; break if used up input string
      ELSE
$arg        SETS ($s:LEFT:(_arglen+1)):RIGHT:1  ; 下一个字符
        IF $arg=" "
_ok       SETL {FALSE}
        ELSE
_arglen       SETA _arglen+1
        ENDIF
      ENDIF
    WEND
$arg    SETS $s:LEFT:_arglen
$s  SETS $s:RIGHT:(:LEN:$s-_arglen)
    _spaces_remove $s
    MEND

; 宏 define
; 作用: 使用 #defines 定义 C/Assembler 变量
; 语法格式如下:#<space/tab>define<spaces><symbol><spaces><value></*comment*/>

    MACRO
$la define $a
_arg0   SETS    "$a"
    ASSERT  "$la"="#"
    _comment_remove _arg0
    _arg_remove _arg0,_arg1
 IF "$_arg0" /= ""
$_arg1  EQU $_arg0
 ELSE
$_arg1 EQU 1
 ENDIF
    MEND

; ifndef and endif 宏
    MACRO
$la    ifndef  $a
    MEND

    MACRO
$la    endif $a
    MEND

; COMMENT
; 作用: 用于注释
; 语法格式 : COMMENT <anything you like!>
    MACRO
    COMMENT $a,$b,$c,$d,$e,$f,$g,$h
    MEND
```

4.6.2 ARM中汇编程序实例

本节列举一些ARM汇编程序的实例。

1. 数据块复制

本程序将数据从源数据区src复制到目标数据区dst。复制时，以8个字为单位进行。对于最后所剩不足8个字的数据，以字为单位进行复制，这时程序跳转到copywords处执行。在进行以8个字为单位的数据复制时，保存了所用的8个工作寄存器。程序的清单如程序4.2所示。

程序4.2 数据块复制

```
; 设置本段程序的名称(Block)及属性
AREA Block, CODE, READONLY
; 设置将要复制的字数
num  EQU  20
; 标识程序入口点
       ENTRY

Start
; r0 寄存器指向源数据区 src
       LDR     r0, =src
; r1 寄存器指向目标数据区 dst
       LDR     r1, =dst
; r2 指定将要复制的字数
       MOV     r2, #num


; 设置数据栈指针(r13)，用于保存工作寄存器数值
       MOV     sp, #0x400
; 进行以 8 个字为单位的数据复制
blockcopy
; 需要进行的以 8 个字为单位的复制次数
       MOVS    r3,r2, LSR #3
; 对于剩下不足 8 个字的数据，跳转到 copywords，以字为单位复制
       BEQ     copywords

; 保存工作寄存器
       STMFD   sp!, {r4-r11}
Octcopy
; 从源数据区读取 8 个字的数据，放到 8 个寄存器中，并更新目标数据区指针 r0
       LDMIA   r0!, {r4-r11}
; 将这 8 个字数据写入到目标数据区中，并更新目标数据区指针 r1
       STMIA   r1!, {r4-r11}
; 将块复制次数减 1
       SUBS    r3, r3, #1
; 循环，直到完成以 8 个字为单位的块复制
       BNE     octcopy
; 恢复工作寄存器值
       LDMFD   sp!, {r4-r11}
```

```
Copywords
; 剩下不足 8 个字的数据的字数
        ANDS    r2, r2, #7
; 数据复制完成
        BEQ     stop
Wordcopy
; 从源数据区读取 18 个字的数据，放到 r3 寄存器中，并更新目标数据区指针 r0
        LDR     r3, [r0], #4
; 将这 r3 中数据写入到目标数据区中，并更新目标数据区指针 r1
        STR     r3, [r1], #4
; 将字数减 1
        SUBS    r2, r2, #1
 ; 循环，直到完成以字为单位的数据复制
        BNE     wordcopy

Stop
; 调用 angel_SWIreason_ReportException
; ADP_Stopped_ApplicationExit
; ARM semihosting SWI
; 从应用程序中退出
        MOV     r0, #0x18
        LDR     r1, =0x20026
        SWI     0x123456
; 定义数据区 BlockData
        AREA BlockData, DATA, READWRITE
; 定义源数据区 src 及目标数据区 dst
src     DCD     1,2,3,4,5,6,7,8,1,2,3,4,5,6,7,8,1,2,3,4
dst     DCD     0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
; 结束汇编
        END
```

2. ADR 伪操作的使用实例

在 ARM 中，地址标号的引用对于不同的地址标号有所不同，伪操作 ADR/ADRL 用来引用地址标号，本实例说明其用法。具体程序清单如程序 4.3 所示。

程序 4.3　伪指令 ADR/ADR2 使用实例。

```
; 设置本段程序的名称(adrlabel)及属性
AREA adrlabel, CODE, READONLY
; 标识程序入口点
        ENTRY
Start
; 跳转到子程序 func 执行
        BL func
; 调用 angel_SWIreason_ReportException
; ADP_Stopped_ApplicationExit
; ARM semihosting SWI
; 从应用程序中退出
stop
        MOV     r0, #0x18
```

```
        LDR     r1, =0x20026
        SWI     0x123456

; 定义一个数据缓冲区，用于生成地址标号相对于 PC 的偏移量
        LTORG

Func
; 下面的伪指令 ADR 被汇编成： SUB r0, PC, #offset to Start
        ADR     r0, Start
; 下面的伪指令 ADR 被汇编成： ADD r1, PC, #offset to DataArea
        ADR     r1, DataArea
; 下面的伪指令 ADR 时错误的，因为第二个操作数不能用 DataArea+4300 表示
        ; ADR r2, DataArea+4300
; 下面的伪指令 ADRL 被汇编成两条指令：
; ADD r2, PC, #offset1
;    ADD r2, r2, #offset2
ADRL    r3, DataArea+4300

; 从子程序 func 返回
        MOV     pc, lr

; 从当前位置起保存 8000 字节存储单元
; 并将其清 0
DataArea SPACE  8000

; 结束汇编
END
```

3. 利用跳转表实现程序跳转

在程序中常常需要根据一定的参数选择执行不同的子程序。本例演示通过跳转表实现程序跳转。跳转表中存放的是各子函数的地址，选择不同子程序的参数是该子程序在跳转表中的偏移量。在本实例中，r3 寄存器中存放的是跳转表的基地址(首地址，其中存放的是第一个子程序的地址)，r0 寄存器的值用于选择不同的子程序，当 r0 为 0 时，选择的是子程序 DoAdd，当 r0 为 1 时，选择的是子程序 DoSub。子程序的清单如程序 4.4 所示。

程序 4.4　利用跳转表实现程序跳转

```
; 设置本段程序的名称(Jump)及属性
AREA    Jump, CODE, READONLY
; 跳转表中的子程序个数
num     EQU     2

; 程序执行的入口点
        ENTRY
Start
; 设置 3 个参数，然后调用子程序 arithfunc，进行算术运算
        MOV     r0, #0
        MOV     r1, #3
        MOV     r2, #2
; 调用子程序 arithfunc
```

```
        BL      arithfunc

Stop
; 调用 angel_SWIreason_ReportException
; ADP_Stopped_ApplicationExit
; ARM semihosting SWI
; 从应用程序中退出
        MOV     r0, #0x18
        LDR     r1, =0x20026
        SWI     0x123456

; 子程序 arithfunc 入口点
arithfunc
; 判断选择子程序的参数是否在有效范围之内
        CMP     r0, #num
        MOVHS   pc, lr
; 读取跳转表的基地址
        ADR     r3, JumpTable
; 根据参数 r0 的值跳转到相应的子程序
        LDR     pc, [r3,r0,LSL#2]

; 跳转表 JumpTable 中保存了各个子程序的地址
; 在这里有两个子程序 DoAdd 和 DoSub
; 当参数 r0 为 0 时上面的代码将选择 DoAdd
; 当参数 r0 为 1 时上面的代码将选择 DoSub
JumpTable
        DCD     DoAdd
        DCD     DoSub

; 子程序 DoAdd 执行加法操作
DoAdd
        ADD     r0, r1, r2
        MOV     pc, lr

; 子程序 DoSub 执行减法操作
DoSub
        SUB     r0, r1, r2
        MOV     pc,lr

; 结束汇编
        END
```

4. 伪指令 LDR 的使用实例

通过伪指令 LDR 可以将基于 PC 的地址标号值或者外部地址值读取到寄存器中。利用这种方式读取的地址值在连接时已经被固定，因而这种代码不是位置无关的代码。当遇到 LDR 伪指令时，汇编编译器将该地址值保存到一个数据缓冲区(literal pool)中，然后将该 LDR 伪指令处理成一条基于 PC 到该数据缓冲区单元的 LDR 指令，从而将该地址值读取到寄存器中。这时，要求该数据缓冲区单元到 PC 的距离小于 4 KB。如果该目标地址值为一个外部地址值或者不在本数据段内，则汇编编译器在目标文件中插入一个地址重定位

伪操作，当连接器进行连接时生成该地址值。

程序4.5中演示了伪指令LDR的用法，并给出了该伪指令汇编后的结果。

程序4.5 伪指令LDR的用法

```
; 设置本段程序的名称(Jump)及属性
AREA LDRlabel, CODE, READONLY
; 程序执行的入口点
      ENTRY

start
; 跳转到子程序 func1 及 func2 执行
       BL      func1
       BL      func2

; 跳转表 JumpTable 中保存了各个子程序的地址
; 在这里有两个子程序 DoAdd 和 DoSub
; 当参数 r0 为 0 时上面的代码将选择 DoAdd
; 当参数 r0 为 1 时上面的代码将选择 DoSub
Stop
       MOV     r0, #0x18
       LDR     r1, =0x20026
       SWI     0x123456

func1
; 下面伪指令被汇编成 : LDR R0,[PC, #offset to Litpool1]
       LDR     r0, =start
; 下面伪指令被汇编成 :  LDR R1,[PC, #offset to Litpool 1]
       LDR     r1, =Darea +12
; 下面伪指令被汇编成 : LDR R2, [PC, #offset to Litpool1]
       LDR     r2, =Darea + 6000
; 程序返回
       MOV     pc,lr
; 字符串缓冲区: Literal Pool 1
       LTORG

func2
; 下面伪指令被汇编成 : LDR r3, [PC, #offset to Litpool 1]
; 共有前面的字符串缓冲区
       LDR     r3, =Darea +6000
; 下面的伪指令如果不注释掉，汇编器将会产生错误信息
; 因为字符串缓冲区 Litpool 2 超出了被伪指令可以达到的范围
       ; LDR   r4, =Darea +6004
; 程序返回
       MOV     pc, lr

; 从单前地址开始，保留 8000 字节的存储单元，
; 并将其内容清除成 0
Darea   SPACE   8000
; 字符串缓冲区 Litpool 2 应该从这里开始，
; 它超出了前面被注释掉的伪指令所能够到达的范围
       END
```