GNU ARM 汇编语法

赵宏智 整理
北京交通大学计算机与信息技术学院

目录

GNU ARM 汇编语法

不同的汇编器对汇编语言的语法要求不一样。目前常用的 ARM 汇编环境有以下两种：

1，ARM ASM：ARM 公司的汇编器，适合在 Windows 平台下使用，其汇编语法可以参考文献 1.”

2，GNU ARM ASM：GNU 工具的 ARM 汇编器版本，适合于 Linux 开发平台，部分汇编语法可以参考文献 2.

本文主要介绍 GNU ARM 语法，其很多语法特征与 Windows 平台下 ARM ASM 汇编语法是不同的。本文所介绍的 GNU ARM 语法，借鉴了上述第二本参考书籍以及一些网络内容，同时结合作者在华为泰山服务器上基于 ArmV8 Aarch64 的编码实践而成。本文所举的汇编语法虽然适用于所有 GNU ARM 汇编编程，但是汇编代码例子都是针对 ArmV8 Aarch64 体系结构展开的。

1、GNU ARM 汇编语言语句格式

任何 Linux 汇编行都是如下结构：

[<label>:][<instruction or directive or pseudo-instruction>} //comment

- <label>: 为标号，GNU 汇编中，任何以冒号结尾的标识符都被认为是一个标号，而不一定非要在一行的开始。标号可以代表“标号处的代码”，也可以代表“标签处的数据的地址”；
- instruction 为 ARM 指令
- directive 为伪操作：伪操作均是以小数点开头，例如“.section”、“.text”等；
- pseudo-instruction 为伪指令
- //为代码行中间的注释符号，comment 为具体的注释语句，其它注释符号参见第 6 节“特殊字符”

下面定义一个"add"函数的例子：

```
.section .text          // 代码段开始,在一些 gnu arm 编译器中，此部分可选。
.global add             // 声明 add 为全局标号，这样它就可以在其它文件中使用
add:                    //  标号名为 add，使用“：”与其后的指令隔开；
    ADD w0, w0, w1      // add input arguments
    ret                 // return from add
// end of program
```

注意：

1>ARM 指令，伪指令，伪操作，寄存器名可以全部为大写字母，也可全部为小写字母，但不可大小写混用。

2>如果语句太长，可以将一条语句分几行来书写，在行末用“\”表示换行（即下一行与本行为同一语句）。“\”后不能有任何字符，包含空格和制表符（Tab)。

2、标号（label）

标号只能由 a～z，A～Z，0～9，″.″，_等（由点、字母、数字、下划线等组成，除局部标号外，不能以数字开头）字符组成，不能与系统预定义的符号、指令助记符和伪指令等同名。标号区分大小写，并且所有的字符都是有意义的。注意：除局部标号外，其它的标号都不能以数字开头。

标号的本质为其所在的地址,该地址既可以当作变量的地址，也可以作为函数的地址来使用。

1>段内标号的地址值在汇编时确定；

2>段外标号的地址值在连接时确定。

局部标号是一种特殊的标号，主要在局部范围内使用，**而且局部标号可以重复出现**。它由两部组成：开头是一个 0-99 直接的数字，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，也可以用 ROUT 来定义局部变量的作用范围。

1>局部变量定义的语法格式：N{routname}

N：为 0~99 之间的数字。

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

2>局部变量引用的语法格式：%{F|B}{A|T}N{routname}

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）

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

例：下面的代码使用了两个局部符号 1，

```
1:
subs w0, w0, #1 //每次循环使 r0=r0-1
cmp w0,0
b.ne 1b       //跳转到当前指令之前 1 标号处去执行，即执行指令 subs…
add w0,w0,1
cmp w0,0
b.ne 1f    //跳转到当前指令之后的 1 标号处去执行，即执行指令 mov w1,1
1: mov w1,1
```

注意：

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。

3、伪操作（directive）

伪操作的作用是告诉汇编器如何对汇编代码进行编译和连接，它不对应具体的机器代码或汇编指令，也不能控制 CPU 或内存的状态。所有的伪操作的名称都是以“.”开始，然后是字母组成的字符串，通常使用小写字母。伪操作可以分为如下几类：段定义类伪操作、标号属性定义类伪操作、数据定义类伪操作、控制类伪操作和杂项伪操作。

3.1 段定义类伪操作

<1>.section 伪操作

用于定义一个段,其语法格式如下：.section <section_name> {,”<flags>”}

GNU ARM 汇编系统预定义的 section_name（段名）有：

.text　　//定义一个代码段。在一些 gnu arm 编译器中，此部分可选。

.data　　//初始化数据段 .data。全局变量都被放在数据段上，数据段中保存的其实是变量的初始值。

.bss　　//未初始化数据段；在源程序中，.bss 段通常在.text 段之前。

这些系统预定义的段都有缺省的段标志 flags，无需用户指明。如果是用户自定义的 section_name，则有必要定义该段的段标识 flags。

Linux 系统中的可执行程序通常是 ELF 格式的二进制代码。一个 ELF 格式的可执行程序通常划分为如下几个部分：.text、.data 和 .bss，其中 .text 是只读的代码区，.data 是可读可写的数据区，而 .bss 则是可读可写且没有初始化的数据区。代码区和数据区在 ELF 中统称为 .section，根据实际需要你也可以添加自定义的.section，但一个 ELF 可执行程序至少应该有一个 .text 部分。下面是 ELF 格式允许的段标志 flags：

<标志>	含义
a	允许段
w	可写段
x	执行段

例：自定义一个段：.mysection：

```
.section .mysection, ”a”  //自定义数据段，段名为 “.mysection”,段标识为允许。
strtemp:
.ascii  "Temp string \n\0"  //将"Temp string \n\0"这个字符串存储在以标号 strtemp: 为起始地址的一段内存空间里;
```

在本例子中，也可以使用.data 来替代.mysection,”w”。

<2>.text 伪操作

用于定义一个代码段，其语法格式如下：

```
.text
codename: …  //codename 是代码段中定义的第一个标号或指令地址
          …
```

<3>.data 伪操作

用于定义一个可读可写的数据段，其语法格式如下：

```
.data
dataname: …   //dataname 是数据段中定义的第一个标号或变量地址
          …
```

<4>.bss 伪操作

用于定义一个可读可写且没有初始化的数据段，其语法格式如下：

```
.bss
bssname: …  //bssname 是 bss 数据段中定义的第一个标号或变量地址
         …
```

3.2 标号属性定义类伪操作

1>.global/.globl：将函数或变量声明为全局标号，使其能够被其它文件调用，如下两种格式都可以:

.global symbol 或者 .globl symbol

例如：在一个 copy_called.s 汇编文件中，定义了一个全局函数 stringcopy 如下：

```
.global stringcopy
stringcopy:  …
```

那么该全局函数 stringcopy 所对应的代码即可被同一工程目录下的其它代码文件如 copy_calling.s 或 copy_calling.c 等所调用。

2> .extern：用来声明一个函数名称或变量为外部标号，和 c 语言中的 extern 关键字是类似的，该外部函数名称对应的代码位于其它代码文件中，即在调用某外部函数之前，要先声明该函数来自外部代码文件。语法格式如下：

.extern symbol

例如：在一个 copy_calling.s 汇编文件中，需要调用一个在 copy_called.s 代码文件中定义的一个函数 stringcopy，那么在 copy_calling.s 中调用该函数之前，就需要先声明该函数为外部标号：

.extern stringcopy

一般说来，.global 与.extern 经常成对出现，在被调用函数所在代码文件中使用.global 来声明该被调用函数，在调用函数所在代码文件中使用.extern 来声明要调用某外部函数。

3> .equ 或.set：把某一个符号(symbol)定义成某一个值(expression).该指令并不分配空间. 语法格式如下：

.equ symbol, expression

例如：

```
.equ IRQ_mode1,0x03  //将符号 IRQ_mode1 定义为一个常数 0x03；
.set IRQ_mode2,0x04  //将符号 IRQ_mode2 定义为一个常数 0x04；
```

4> .type：用来指定一个符号的类型是函数类型或者是对象类型。

如果是定义对象类型，表示该符号是一个数据或变量，格式如下:

```
.type num,@object  //指定符号类型为对象(或者是一个变量)
```

示例：

```
.data
.globl a
.type a, @object
.size a, 2
a:
.short 8
```

这里的 a 为一个两字节的 short 类型的全局变量，其值为 8，用.type 伪操作突出其对象或变量的属性。

如果是定义函数类型，表示该符号是一个函数，格式如下:

```
.type func,@function  //指定符号类型为函数;
```

示例:

```
.text
.type afunc, @function
.globl afunc
afunc:
…
```

这里的 afunc 为一个函数，用.type 伪操作突出其函数属性。

3.3 数据定义类伪操作

数据定义伪操作一般用于在内存中开辟一段内存空间来存放数据，或者说是为一些变量分配内存单元。常用的数据定义伪操作有如下几类:

1>.byte：在内存中分配一个字节的内存单元，格式为:

```
.byte expr
```

expr 的值可以是一个 8 位的带符号数/无符号数/字符。

例如：.byte 0x01

其在内存中开辟的一个 1 字节的存储空间，存放的值为 0x01。

如果要连续定义多个字节，还可以简写为:

```
.byte expr1,expr2,…
```

2>.short：在内存中分配两个字节的内存单元，格式为:

```
.short expr
```

expr 的值可以是一个 16 位的带符号数/无符号数或者两个字符。如果要连续定义.short 存储单元，还可以简写为:

```
.short expr1,expr2,…
```

3>.word 或.long 或.int: 在内存中分配四个字节的内存单元，格式为:

```
.word expr 或 .long expr 或.int expr
```

expr 的值可以是一个 32 位的带符号数/无符号数或者四个字符。如果要连续定义.word(或.long，.int)存储单元，还可以简写为:

```
.word expr1,expr2,… 或 .long expr1,expr2,…或 .int expr1,expr2,…
```

.word 通常还有一种有趣的用法，可以把标识符作为常量使用。

例:

Start:

AddressOfStart:

.word Start

这样标号 Start（是一个 32bit 的内存地址）便被存入了内存变量 valueOfStart 中。

4>.quad: 在内存中分配八个字节的内存单元，格式为：

.quad expr

expr 的值可以是一个 64 位的带符号数/无符号数或者八个字符。如果要连续定义.quad 存储单元，还可以简写为：

.quad expr1,expr2,…

5>.float: 在内存中分配一个 4 字节的存储空间，并用指定的浮点数据对其进行初始化，格式为：

.float expr

例如：.float 2.11

其在内存中开辟的一个 4 字节的存储空间，存放的值为 0x40070a3d。

如果要连续定义.float 存储单元，还可以简写为：

.float expr1,expr2,…

6>.space：从当前位置开始，分配一片连续的存储区域，并将该区域所有存储单元的值初始化为指定的值，格式如下：

.space size[,expr]

其中 size 表示要分配一个 size 个字节的存储区域；expr 表示该存储区域存放的初始化值，该初始化值如果大于 0xff，则会截取低 8 位为一个字节长度的初始化值；如果“,expr”省略不写，则默认其值为一个字节长度的 0。

例如：

.space 100,1 //从当前位置开辟一个 100 个字节的连续内存空间，其中每个字节都存放一个 0x01；

.space 100 //从当前位置开辟一个 100 个字节的连续内存空间，其中每个字节都存放一个 0x00；

7>.ascii、.asciz 或.string：从当前位置开始，在内存中存放一个字符串。字符串用双引号“”包含在内，其语法格式如下：

.string "string1" 或 .asciz "string1" 或 .ascii "string1\0"

注意：ascii 伪操作定义的字符串需要自行添加结尾字符'\0'以表明字符串结束。

示例：

```
.string "abcd"
.asciz "qwer"
.ascii "welcome\0"
```

如果要连续定义多个字符串，还可以简写为：

.string/.asciz/.ascii string1,string2,…

8>.rept 和.endr: 重复定义伪操作，格式如下：

```
.rept 重复次数
   code/data definition
.endr
```

.rept 表示重复定义的开始，.endr 表示重复定义的结束。整个重复定义伪操作的含义是将 code 代码或数据定义重复若干次，相当于循环展开。

例 1：

```
 .rept 3
.byte 0x04
.endr
```

相当于:

```
    .byte 0x04,0x04,0x04
```

例 2：

```
mov w0,1
 .rept 3
add w0,w0,1
.endr
```

相当于执行 add 指令三次，之后 w0 的值为 4。

需要说明的是，在基于 Aarch64 处理器的 centos 和 GCC 4.8.5 版本下（鲲鹏云）， .align 伪操作不起作用。

3.4 控制类伪操作

控制类伪操作用于控制汇编代码的编译过程，常用的控制类伪操作有：

1> 条件类伪操作

.if、.else 和.endif，支持条件预编译，可以根据一个表达式的值来决定是否要编译下面的代码, 用.endif 伪操作来表示条件判断的结束, 中间可以使用.else 来决定.if 的条件不满足的情况下应该编译哪一部分代码。语法结构如下：

```
.if logical-expression
   code 1;
.else
   code 2;
.endif
```

如果分支较多，还可以使用.elseif 来增加更多的分支：

```
 .if logical-expression1
   code 1;
.elseif  logical-expression2
   code 2;
.elseif  logical-expression3
   code 3;
    …
.else
    code n;
.endif
```

2> 宏定义类: .macro 和.endm

.macro 和.endm 可以将一段代码定位为一个整体，.macro 表示代码的开始，.endm 表示代码的结束，之后就可以在程序中通过宏调用的方式来对其进行多次调用。格式如下：

```
.macro 宏名 参数名列表    //.macro 定义一个宏的开始
    code    //宏体
.endm                    //.endm 表示宏结束
```

如果宏使用参数,那么在宏体中使用该参数时添加前缀“\”。宏定义时的参数还可以使用默认值。当宏操作被展开时，这些参数将被相应的值替换。宏操作的使用方式、功能与子程序类似，可以提供模块化的程序设计。但是与子程序方式相比，宏定义方式不需要包含程序现场，开销较小。

示例：宏定义

```
.macro SHIFTLEFT a, b
    MOV \a, \a, ASR #-\b
.endm
```

3.4 杂项伪操作

1> .ltorg:用于声明一个数据缓冲池(literal pool)的开始,它可以分配很大的内存空间，用于存放一些常量数据。

.ltorg 用于声明一个文字池（即内存中的一段数据存储区。.data 所定义的数据存储区是组成 ELF 格式的一部分，属于一个进程空间的一个 section，是与.text section 相互独立的；而.ltorg 所定义的数据存储区则可以看做是.text section 内部的一部分，多用于存放代码段中的一些常量数据。）

在汇编代码中使用 LDR 伪指令来对寄存器赋值时，形如“LDR X1,=const”，const 值往往需要放到文字池中。在阐明文字池的用法之前，有必要先了解一下汇编器中 LDR 伪指令的寻址原理：

①如果 const 在本 LDR 指令之前就定义过的文字池（即本伪指令之前的位置已经有用过 ltorg 定义过上一或上几个文字池了）里，那么就去这些文字池里寻找 const 的内容；

②如果在本 LDR 伪指令之前没有定义过文字池，或者之前定义的文字池里没有对 const 进行定义，那么就在本 LDR 伪指令之后再寻找一个文字池。如果本 LDR 伪指令之后有这么一个定义了 const 的文字池，并且该文字池距离当前 LDR 伪指令的地址距离在 LDR 伪指令的寻址范围内（在 arm32 中，从当前 LDR 伪指令的 PC 值到 const 地址处的距离必须小于 4KB），那么就去该文字池里寻找 const 的内容即可；

③如果在本 LDR 伪指令之后没有文字池，或者之后的第一个文字池距离当前伪指令的地址距离超出了 LDR 伪指令的寻址范围（比如在 arm32 中，该距离大于 4KB），那么就需要在本 LDR 伪指令之后的一个合适位置处用“.ltorg”新定义一个文字池，并且在该文字池中添加对 const 内容的定义。所谓的合适位置，一是要确保新定义文字池与当前 LDR 伪指令之间的距离不超过 LDR 伪指令的寻找范围，二是要确保该处不能被指令执行到（即 PC 寄存器能够访问到的地址）。在一个代码段中，不能被指令执行到的地方有：第一条指令之前的位置，比如在.text 之前的位置；在 ret、无条件跳转指令之后以及下一条指令之前的位置等。如果文字池放到了能够被 PC 寄存器访问到位置，那么 const 数据就会当做指令来执行。

④经过前 3 个步骤，汇编器就可以告诉该 LDR 伪指令其所寻找的 const 所在的文字池位置以及 const 在其中的偏移量，从而正确取出 const 处所存储的内容。

2>.arch 伪操作：指定指令集版本

例如： .arch armv8-a ，表明当前代码用到的是 armv8-a 架构下的指令集。

4、伪指令（pseudo-instruction）

GNU ARM 汇编器提供 LDR,ADR 和 ADRL 三个伪指令。汇编器在编译汇编代码的时候，会根据当前代码所处的环境（即 A64/A32/T32/T16）将这些伪指令替换为合适的 ARM64（或 A64）/ARM32(或 A32)/Thumb32(或 T32)/Thumb16(或 T16)指令(或指令和文字池)。

4.1 LDR 伪指令

本伪指令通常用来获取**一个变量的地址**（该变量可以是一个字节、字、字符串等，该变量的值通常放在一个文字池中。该变量的地址，即下面的 expression，可以用一个立即数来表示其低位，也可以用一个标号来表示），并将该地址放到一个寄存器中。其语法格式为：

ldr <register>,=<expression>

在用法上，该伪指令经常与其它指令配合，实现对一个数据段或文字池中数据的操作。

举个例子，变量 aa 的值为 50，其所在的地址为 0x0000 0000 0000 01200（该地址在一个文字池或数据段中）；那么 expression 就是一个存放值 0x0000 0000 0000 01200 的地址，比如为 0x400278。那么：伪指令 ldr x1,=aa 就是相当于指令 ldr x1,0x400278.

<1> **如果 ldr 伪指令中的表达式是立即数(imm)，**不同的指令中对立即数长度的限制是不同的，例如 A32 中 mov 指令限制了立即数的长度不能超过 8 位，A64 中 mov 指令限制了立即数的长度不能超过 16 位。而 ldr 伪指令对立即数的长度是没有这个限制的，将根据立即数的长度做不同的处理，以 A64 下的“LDR X4,=imm”为例：

- 如果 imm 没有超过 X4 寄存器的表示范围，那么在实际汇编的时候该 imm 值将直接被放入文字池中。该 LDR 伪指令则被替换为指令“LDR X4,0x400278”，地址 0x…400278（其…表示高位，高位部分与当前指令 LDR 的地址的高位相同）中存放了文字池中 imm 值所在内存地址。
- 如果 imm 超过了 X4 寄存器（X4 是 64 位寄存器），比如为“LDR X4, =0x123456789abcdef0123456”，其长度超长了 64 位，则汇编器将该 imm 的低 64 位放入**文字池**中，超长 64 位之外的部分被舍弃。在该文字池中的存放 imm 值的地址（比如为 0x400278）则被赋值给 x4。该 LDR 伪指令则被替换为指令“LDR X4,0x400278”，并且在离当前伪指令地址最近的 64 位内存地址 0X…400278 处（其…表示高位，高位部分与当前指令 LDR 的地址的高位相同）存放低 32 位数据 0xf0123456，在 64 位内存地址 0X…40027c 处存放较高的 32 位数据 0x789abcde；至于更高位的数值 0x123456 则被舍弃。

<2>**如果 ldr 伪指令中的表达式是标号，**汇编器则将标号替换为存储标号所对应的内存地址的内存地址（即存储标号对应地址值的地址！）例如，“ldr x1,=msg”，msg 为字符串“hello!”的标号，该字符串放在地址为 0x00420000 处。该伪指令将被替换为指令“ldr x1,0x400280”，而在地址 0x…400280（…表示内存地址的高位，该高位与当前 ldr 指令的高位内存地址相同）处存放的内容为 0x00420000，即 msg 标号所在的内存地址。

简而言之，伪指令 ldr <register>,=<expression>虽然看上去只是一条伪指令，但是其要占用 2 个 32bit 的空间：第一个 32bit 空间是指令“ldr <register>,expression 所在地址的地址”； 第二个 32bit 空间是文字池中的一个 32bit 空间，用于存放 expression 所在的地址。

需要注意的是，如果形如“ldr <register>,<expression>”，即缺失一个“=”，那么这里的ldr 就是一个指令。

4.2 ADR 伪指令

本伪指令通常用来获取代码段中**某条指令的地址**（该指令的地址，即下面的 label。该地址与当前 PC 指向的地址间的差值必须比较小，例如地址差值不能超过 add 指令中立即数的表示范围。比如，add register,pc,#imm 中立即数 imm 的范围不能超过-255~255byte 时，那么 ADR 指令的 PC 值距离 label 地址的差值就不能超过-255~255byte），并将该地址放到一个寄存器中。其语法格式为：

adr <register>,<label>

该伪指令通常会被编译器替换为一条 add 指令或 sub 指令，形如“add register,pc,#offeset_to_label”或“sub register,pc,#offeset_to_label”。

在实际使用中，该伪指令经常与其它指令配合，实现代码的小范围跳转或者获取指令的机器码。需要注意的是，label 与当前 adr 伪指令必须在同一代码段中。

4.3 ADRL 伪指令

ADRL 指令的功能与 ADR 指令基本相同，区别在于：当 ADR 伪指令中 label 地址与当前伪指令的 PC 值地址间的地址差值较大，超出了 ADR 伪指令所能够表示的地址范围时，就需要使用 ADRL 伪指令。ADRL 指令其语法格式为：

adrl <register>,<label>

该伪指令通常会被编译器替换为两条 add 指令或两条 sub 指令，形如“add register1,pc,#offeset_to_label_1”和“add register1,register1,#offeset_to_offeset_to_label_1”。即两条 add 指令中的偏移量相加，即可得到 label 地址与当前伪指令的 PC 值地址间的地址差值。需要注意的是，label 与当前 adr 伪指令必须在同一代码段中。

* 关于 NOP 指令，在 ArmV8 架构下，NOP 是指令而不是伪指令，其反汇编之后是有机器码的。而在 arm32 架构中，NOP 则是伪指令。

5、常数定义

<1>立即数定义符号：#，如：#64； 在面向 ARM64 的汇编中，在立即数之前添加或不添加#，不影响赋值的正确性。

<2>十进制数以非 0 数字开头,如:64 和 9876;

<3>二进制数以 0b 开头,其中字母 b 大写或小写都可以，例如 0b01000000(或者 0B01000000),相当于 0x40 或者 64）;

<4>八进制数以 0 开始,如:0100（相当于 0x40 或者 64）； 注意：是 0 不是 o!

<5>十六进制数以 0x 开头,如:0x40,0x123f;

<6>字符串常量需要用引号括起来,中间也可以使用转义字符,如: “You are welcome!\n”;

<7>当前地址以“.”表示,在 GNU 汇编程序中可以使用这个符号代表当前指令的地址；例如表达式 len=.-msg 所表达的意思是：常数 len 的值是当前地址减去 msg 标号所在的地址的差值;

<8>表达式：在汇编程序中的表达式可以使用常数或者数值，“-”表示取负数，“~”表示取补，“<>”表示不相等。

6、特殊字符

1,代码行中的注释符号：与 C 语言中的用法相似，有：// 、 /* */ 和 #

例如：

<1> “.text // hello” 中的//符号仅仅将“hello”注释掉！

<2> “# hello！”中的#是将本行的内容都注释掉！

<3> /* This is a comment! */

需要说明的是，@不能作为注释符号。

2，; 行分隔符（或者语句分离符号）

例如：

```
.section  .text ;  .global add  ; add:
ADD w0,w0,w1
```

等价于如下代码：

```
.section  .text
.global add
add:
ADD w0,w0,w1
```

3，其他的符号如:+、-、*、 /、%、<、<<、>、>>、|、&、^、!、==、>=、<=、&&、|| 跟 C 语言中的用法相似。

7、程序入口函数定义

GNU 汇编程序的入口函数命名有两种：一，如果使用 gcc 编译器来对汇编程序进行编译的话，由于_start 入口函数已经被 crt1.o 文件所定义，则入口函数不能再使用_start 名称，需要自定义其它名称；二，如果使用 as 汇编器和 ld 连接器来编译链接汇编程序的话，则入口函数名称可以使用_start 名称。

例： 使用 gcc 编译器时，汇编程序入口函数可以如下：

```
.text
.global st1
st1:
      mov w0, #1
      ……
```

入口函数通常用“.global”来标记为是一个全局函数。w0 是 Aarch64 架构下的寄存器名称。

8、GNU 内嵌(inline)ARM 汇编代码

在很多实际的开发场景中，C 是无法完全代替汇编语言的，一方面是其效率比 C 要高，另一方面是某些特殊的指令在 C 语法中是没有等价的语法的。例如操作某些特殊的 CPU 寄存器如状态寄存器、操作主板上的某些 IO 端口或者对性能要求极其苛刻的场景等，我们都可以通过在 C 中内嵌汇编来满足要求。

8.1 内嵌汇编语句的格式

在 C 代码中内嵌的汇编语句的基本格式为：

```
__asm__ __volatile__ ("asm code"
        : 输出操作数列表
        : 输入操作数列表
        : clobber 列表
);
```

说明：

1，__asm__前后各两个下划线，并且两个下划线之间没有空格，用于声明这行代码是一个内嵌汇编表达式，是内嵌汇编代码时必不可少的关键字。

2，关键字 volatile 前后各两个下划线，并且两个下划线之间没有空格。该关键字告诉编译器不要优化内嵌的汇编语句，如果想优化可以不加 volatile； 在很多时候，如果不使用该关键字的话，汇编语句有可能被编译器修改而无法达到预期的执行效果。

3，括号里面包含四个部分：汇编代码（asm code）、输出操作数列表（output）、输入操作数列表（input）和 clobber 列表（破坏描述符）。这四个部分之间用“：”隔开。其中，输入操作数列表部分和 clobber 列表部分是可选的，如果不使用 clobber 列表部分，则格式可以简化为：

```
__asm__ __volatile__ ("asm code"：output：input);
```

如果不使用输入部分，则格式可以简化为：

```
__asm__ __volatile__ ("asm code"：output::changed);
```

此时，即使输入部分为空，输出部分之后的“：”也是不能省略的。

另外，输入部分和 clobber 列表部分是可选的，如果都为空，则格式可以简化为：

```
__asm__ __volatile__ ("asm code"：output);
```

关于这四个部分的具体语法将在后面的小节中加以介绍。

4，括号之后要以“;”结尾。

8.2 输出操作数列表

输出部分则可以理解为内嵌汇编代码给 C 代码的返回值。输出部分的典型格式为：

```
__asm__ __volatile__ (
  "asm code"
  :[result]“constraint”（C 变量名）
);
```

分为三个部分：[ ]内的符号名，””里面的约束字符串和()里面的 C 变量名部分。

在[ ]中的符号名，可以使用类似 result1 或 output2 等名称来表示这是一个输出参数，方便程序员记忆。在汇编代码将其作为某指令的目的操作数时，可以使用“%[result1]”形式来代替该目的操作数。例如：__asm__(”mov %[result1],onearray”:[result1]”=r”(y))。其中 y 是 C 代码中的待赋值变量，其值存放在编译器分配给其的一个寄存器中。该分配过程是隐式进行的，程序员可以利用这个特性少写一条从 onearray 处取出一个数据放入某寄存器中的指令。

在””中的约束字符串部分由两部分组成：C 变量名的存放位置类型和修饰符。

- C 变量的存放位置类型有两类：

  r　存放在某个通用寄存器中，即在汇编代码里用一个寄存器代替()部分中定义的 c 变量。

  m　存放在内存地址中

- 修饰符部分有三类：

  +　可读可写，既可以取变量值，也可改变变量的值。

  =　只写，即在汇编代码里只能改变 C 变量的值，而不能取它的值。

  &　该输出操作数不能使用输入部分使用过的寄存器，只能+& 或 =& 方式使用

例如：

″=&r″（result），″=&r″（tmp）

表示汇编代码部分有两个输出值，对应两个 C 变量 result 和 tmp。这两个输出值分别存放在汇编代码的两个寄存器中。这两个寄存器只能修改这两个变量的值，但是不能读取这两个变量的值，而且在 input 输入部分用过的寄存器是不能用作这两个寄存器的。至于这两个寄存器是哪两个寄存器，则由编译器自行分配。一般说来，根据 Arm 公司的 AAPCS64，即 Aarch64 程序调用标准，Aarch64 标准提供了 8 个 64bit 通用寄存器（x0-x7）用于传递函数参数，依次对应于参数 1、参数 2、参数 3…参数 8，当存放位置类型为 r 类型时，编译器默认地从这 8 个 64bit 通用寄存器中选出一个未被分配过的寄存器来分配给 C 变量 result 或 tmp。

如果某操作数既作为输入参数，也作为输出参数，那么在输出操作数列表部分可以使用约束符号“+”来表明其可读可写。

()里面的 C 变量名来自于 C 代码。C 变量的类型及其对应的字节数目应当与汇编指令中与其对应的寄存器的宽度保持一致。

8.3 输入操作数列表

输入操作数列表用于从 C 代码向内嵌汇编代码传入参数。

```
__asm__ __volatile__ (
        "asm code"
        :
        :[value]“constraint”（variable / immediate）
    );
```

输入操作数列表分为三个部分：[ ]内的符号名，””里面的约束字符串和()里面的 C 变量名部分。

在[ ]中的符号名，可以使用类似 value1 或 input1 等名称来表示这是一个输入参数，方便程序员记忆。在汇编代码将其作为某指令的源操作数时，可以使用“%[value1]”形式来代替该源操作数。例如：__asm__(… ”mov onearray,%[input1]”: … :[input1]”=r”(x))。其中 x 是 C 代码中的一个变量，其值存放在编译器分配给其的一个寄存器中。该分配过程是隐式进行的，

程序员可以利用这个特性少写一条将 x 赋值给一个寄存器的指令。

constraint 定义 variable / immediate 的存放位置：

r	使用任何可用的通用寄存器（变量和立即数都可以）
m	使用变量的内存地址（不能用立即数）
i	使用立即数（不能用变量）
0	与第一个输出参数共用同一个寄存器

( )里面的 C 变量名来自于 C 代码，也可以是一个常量。如果是变量的话，则该变量的类型及其对应的字节数目应当与汇编指令中与其对应的寄存器的宽度保持一致。

注意 1：输入操作数是只读的，不能对其进行写操作，即在汇编代码部分不能将其作为某汇编指令的目标操作数。一个特例就是：如果某操作数既作为输入参数，也作为输出参数，那么其是可写的，但是需要在输出操作数列表部分使用约束符号“+”来表明其可读可写，不应在输入操作数列表中对其进行描述。

注意 2：使用限制符 0 来约束 variable / immediate 的存放位置时，如果本输入参数所对应的寄存器作为输出参数对应的寄存器已经被修改过了，那么就会发生错误。此时需要仔细安排内嵌汇编代码的指令顺序，避免这种情况发生。

注意 3：在 Arm64 架构下，输入参数如果是存放在寄存器中，那么所用的寄存器都是 Xn 而不是 Wn。

8.4 Clobber 列表

Clobber 列表也称为破坏描述符列表或修改描述符列表，描述了在内嵌汇编代码在执行过程中，将被修改的寄存器、内存空间或标志寄存器等告诉编译器，以便编译器在编译内嵌代码时保持这些内容不变。一个设计精细的优化器可能重新排列代码顺序，导致寄存器和内存空间被修改的顺序发生预期之外的改变，即这些值被意外修改。通过使用 clobber 列表，则可以避免其被编译器安排的其它代码按非预期顺序所修改。如果此部分省略，则这些将被修改的寄存器、内存空间或标志寄存器等则有可能被编译器安排的其它代码所修改。

- **寄存器修改通知定义**：当一个内联汇编中包含影响某寄存器 Xn 或 Wn 的条件,为了不改变该寄存器的值，需要在本部分中使用"Xn"或”Wn”来向 GCC 声明这一点。如果需要声明多个寄存器，则必须使用逗号“,”将它们分隔开。
- **内存修改通知定义**：相当于定义一个内存的临界区，保护其不被编译器安排的本内嵌代码之外的其它部分代码所修改。如果在内嵌代码中需要修改某部分内存单元，为了避免编译器安排其它的代码先对该部分内存单元进行修改，那么需要在本部分中使用"memory"来向 GCC 声明这一点。比如：如果某个内存单元的内容被装入了寄存器，那么在这个内联汇编之后,如果需要使用这个内存单元处的内容，就会直接到这个内存单元处重新读取,而不是使用被存放在寄存器中的拷贝，因为这个时候寄存器中的拷贝很可能已经和该内存单元处的内容不一致了。简而言之，如果使用”memory”定义，那么在编译该内嵌汇编代码前后，该内存单元处的值将保持不变。
- **标志寄存器修改通知定义**：当一个内联汇编中包含影响标志寄存器的条件，为了不改变标志寄存器的值，那么需要在本部分中使用"cc"来向 GCC 声明这一点。如果使用”cc”定义，那么在执行内嵌汇编代码前后，该标志寄存器处的值将保持不变。

通过本部分定义，可以将被修改的寄存器、内存空间或标志寄存器等作为 C 代码的局部临时变量来使用。为了保持原值不变，不同的编译器有着不同的做法，既可以要求编译器不要将所定义的寄存器 Xn 或内存单元分配给%0 和%1 等，也可以在内嵌汇编代码之前压栈保存以及代码之后出栈恢复。

需要注意的是：如果在本部分指定了太多的通用寄存器，编译器可能在产生代码时就将可用的寄存器耗尽，从而无法为 C 代码分配可用的寄存器。

下面是摘自 https://blog.csdn.net/tidyjiang/article/details/52138598 的一个例子：

下面一段代码的作用是计算 b 和 c 的乘积。其中，b 和/或 c 的值可能会被中的例程修改。因此，我们在访问变量前先禁止中断，并在完成计算后再重新使能中断。

```
__asm__ __volatile__("mrs r12, cpsr\n\t"
    "orr r12, r12, #0xC0\n\t"
    "msr cpsr_c, r12\n\t" ::: "r12", "cc");
c *= b; /* This may fail. */
__asm__ __volatile__("mrs r12, cpsr\n"
    "bic r12, r12, #0xC0\n"
    "msr cpsr_c, r12" ::: "r12", "cc");
```

不幸的是，优化器可能会让乘积指令先执行，再执行两个内联汇编指令，或者相反。这会让我们的汇编代码毫无意义。

对于这个问题，我们可以借助于 clobber 列表。该例程中的 clobber 列表如下：

```
"r12", "cc"
```

这条 colbber 列表将给编译器传达如下信息：我这段汇编代码修改了寄存器 r12 的值，并更新了程序状态寄存器的标志位。顺便说一下，直接指明使用的寄存器，将有可能阻止了最好的优化结果。一般情况下，你应该传递一个变量，让编译器自己选择寄存器。clobber 列表中的关键字，除了寄存器名和 cc(状态寄存器标志位)，还包括 memory。memory 关键字用来指示编译器，该汇编指令改变了内存中的值。这将强制编译器在执行汇编指令前将所有缓存的值保存起来，并在汇编指令执行完后再将这些值加载进去。此外，编译器还必须保留执行顺序，因为在执行完带有 memory 关键字的 asm 语句后，所有变量的内容都是无法预测的。

```
__asm__ __volatile__("mrs r12, cpsr\n\t"
    "orr r12, r12, #0xC0\n\t"
    "msr cpsr_c, r12\n\t" :: : "r12", "cc", "memory");
c *= b; /* This is safe. */
__asm__ __volatile__("mrs r12, cpsr\n"
    "bic r12, r12, #0xC0\n"
```

```
    "msr cpsr_c, r12" ::: "r12", "cc", "memory");
```

使所有缓存值无效可能是次优化的。你也可以添加一个虚拟操作数来创建一个虚拟依赖：

```
__asm__ __volatile__("mrs r12, cpsr\n\t"
    "orr r12, r12, #0xC0\n\t"
    "msr cpsr_c, r12\n\t" : "=X" (b) :: "r12", "cc");
c *= b; /* This is safe. */
__asm__ __volatile__("mrs r12, cpsr\n"
    "bic r12, r12, #0xC0\n"
    "msr cpsr_c, r12" :: "X" (c) : "r12", "cc");
```

上述代码中，第一条汇编语句尝试去修改变量 b ，第二条汇编语句尝试使用变量 c，使得这三个部分存在数据依赖关系，编译器据此将保留三个语句的执行顺序。

8.5 汇编代码部分

汇编代码部分用于声明这行代码是一个内嵌汇编表达式。C 语言中嵌入汇编需要显式的标明寄存器的分配情况、与 C 程序的融合情况等。主要靠内嵌汇编表达式说明。

- **指令以字符串形式出现**：所有的汇编指令必须放在一个以双引号“”定义的字符串内。但是为了增加程序的可读性，最好将每一个汇编指令单独放一行，每一行都以一个字符串的形式存在，但是字符串中间是不能直接按回车键换行的，需在字符串的结尾处添加：\n 换行符，相当于回车键；还可以再添加\t 制表符，相当于 tab 键。
- **指令的操作数**：本部分的 arm 汇编指令，除去汇编指令的操作数用法与常规汇编指令可能有所不同之外，其它部分与常规 arm 汇编指令都完全相同。关于汇编指令的操作数部分，除了常规的寄存器和变量标号形式之外，最典型的就是其表示为“%n”的形式（n 为非负整数）。假设在输出操作数列表中如果定义了 n1 个输出参数值，在输入操作数列表中定义了 n2 个输入参数值，那么在 asm code 中的指令的形如“%n”的操作数含义为：如果 0<=n<=n1-1，那么表示该操作数为输出参数列表中定义的第 n 个输出参数（按在列表中出现的顺序，编号从 0 到 n1-1）；如果 n1<=n<=n1+n2-1，那么表示该操作数为输入参数列表中定义的第 n-n1 个输出参数（按在列表中出现的顺序，编号从 n1-1 到 n1+n2-1）。如果某操作数既作为输入参数，也作为输出参数，那么在输出操作数列表部分可以使用约束符号“+”来表明其可读可写。

8.6 例子

下面的例子主要来自 http://wenku.baidu.com/view/b7bb0e116c175f0e7cd13733.html ，本文在其基础

上做了一些改进。

例 1：将 C 代码中的一个变量值赋值给另外一个变量，

```
int x = 5, y = 0;
__asm__("mov %[output], %[input]\n":[output] "=r"(y):[input] "r" (x));
```

在本例中，y 作为被赋值变量，x 作为赋值变量。在输出操作数列表定义部分，编译器将变量 y 存放在一个寄存器中；在输入操作数列表部分，编译器将变量 x 存放在另外一个寄存器中；在代码部分，则编译器使用这两个寄存器来实现赋值功能。编译器分配的是哪两个寄存器程序员是无需了解的。

例 2：内嵌汇编代码中有多个输出操作数 a，b 和 result，有一个输入操作数为立即数的情况

```
int a = 100,b = 200;
int result;
__asm__ __volatile__ (
        “mov     %0,%3/n/t”     //mov w3,#123，%0 代表 result，%3 代表 123（编译
                                //器会自动加#号）
        “ldr     w0,%1/n/t”      //ldr w0,[fp, #-12]，%1 代表 a 的地址
        “ldr     w1,%2/n/t”      //ldr w1,[fp, #-16]，%2 代表 b 的地址
        “str     w0,%2/n/t”      //str w0,[fp, #-16]，因为%1 和%2 是地址所以只能用
                                //ldr 或 str 指令
        “str     w1,%1/n/t”     //str w1,[fp, #-12]，如果用错指令编译时不会报错，要
                                //到汇编时才会
        ：“=r”(result),“+m”(a),“+m”(b)  //第一个输出操作数 result 是%0，编译器将
                                //其存放到一个寄存器中，只写；第二个输出操作数
                            //a 是%1，编译器将其存放到一个内存单元中，可读可写；
                            //第三个输出操作数 b 是%2，编译器将其存放到一个内存
                            //单元中，可读可写；
        ：“i”(123)      //第一个输入操作数是%3；
);
```

注释部分的 fp 相关地址都是编译器自动分配的。

例 3：内嵌汇编代码中一个变量既可读又可写的情况

```
int num = 100;
__asm__ __volatile__ (
    "add     %0,%1,#100/n/t"
    : "=r"(a)       //变量 a 作为可写的输出操作数，用%0 表示，其值被编译器存放到
                    //一个寄存器中；
    : "0"(a)        //变量 a 作为可读的输入操作数，"0"即%0，表示其将和第 1 个输出
                    //操作数共用一个寄存器
    );
```

例 4：内嵌汇编代码中只有一个输入操作数和一个输出操作数，编译器为其自动分配同一个寄存器的情况

```
int num;
```

```
__asm__ __volatile__ (          //mov r3, #123；本指令是编译器自动加的
   "mov     %0,%1/n/t"          //mov r3,r3；编译器为输入和输出操作数分配了一个
                                //相同的寄存器
   : "=r"(num)      //num 为第一个输出操作数，即%0，存放在编译器自动分配的一
                    //个寄存器中
   : "r"(123)       //常数 123 为第一个输入操作数，即%1，存放在编译器自动分配
                    //的一个寄存器中
    );
```

如果内嵌汇编代码中仍然只有一个输入操作数和一个输出操作数，但是不想编译器为其自动分配同一个寄存器的情况，那么可以使用&符合来做到这一点：

```
    int num;
__asm__ __volatile__ (              //mov     r3, #123
   "mov     %0,%1/n/t"              //mov     r2,r3
   : "=&r"(num)                    //mov     r3, r2 ；加了&后编译器为输入操作数和输
                          //出操作数分配的寄存器不一样了
   : "r"(123)
  );
```

参考文献

1， 文全刚,郝志刚,张荣高.《汇编语言程序设计：基于 ARM 体系结构》(第 3 版). 北京航空航天大学出版社. 2016).

2， 刘洪涛,秦山虎,武立鑫等. 《ARM 嵌入式体系结构与接口技术》: Cortex-A9 版: 微课版》. 人民邮电出版社.2017.

注：上述两本参考书籍讲的是 Arm32 或者 Aarch32 汇编程序的编写，其编写的代码无法直接在 ArmV8 鲲鹏处理器上运行，有些指令如 Add r1,r2,r3 等中的寄存器 Rn 标识是不能被 64bit OS 下的 GCC 编译器所能够识别的。在 ArmV8 鲲鹏处理器上只能运行使用 Xn 或 Wn 寄存器的指令，不能使用 Rn 寄存器。