汇编源程序一般用于系统最基本的初始化：初始化堆栈指针、设置页表、操作 ARM的协处理器等。这些初始化工作完成后就可以跳转到C代码main函数中执行。

**1、  GNU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**

         instruction为指令

         directive为伪操作

         pseudo-instruction为伪指令

         **<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。

         comment为语句的注释

下面定义一个"add"的函数，最终返回两个参数的和：

.section .text, “x”

.global add @ give the symbol “add” external linkage

add:

ADD r0, r0, r1 @ add input arguments

MOV pc, lr @ return from subroutine

@ end of program

注意：

          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。

          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、  GNU汇编程序中的标号symbol（或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。

Symbol的本质：代表它所在的地址,因此也可以当作变量或者函数来使用。

          段内标号的地址值在汇编时确定；

          段外标号的地址值在连接时确定。

Symbol的分类：3类（依据标号的生成方式）。

**<1>** 基于PC的标号。基于PC的标号是位于目标指令前的标号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，常用于表示跳转指令**”b”**等的目标地址，或者代码段中所嵌入的少量数据。

**<2>** 基于寄存器的标号。基于寄存器的标号常用MAP和FIELD来定义，也可以用EQU来定义。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，常用于访问数据段中的数据。

**<3>** 绝对地址。绝对地址是一个32位数据。它可以寻址的范围为[0，232-1]即可以直接寻址整个内存空间。

**特别说明：**局部标号Symbol

局部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。

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）。

局部变量引用的语法格式：**%{F|B}{A|T}N{routname}**

          %：表示引用操作

          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：使用局部符号的例子，一段循环程序

1:

subs r0, r0, #1 @每次循环使r0=r0-1

bne 1F @跳转到1标号去执行

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、  GNU汇编程序中的分段**

**<1>** .section伪操作

.section <section\_name> {,”<flags>”}

Starts a new code or data section. Sections in GNU are called .text, a code section, .data, an initialized data section, and .bss, an uninitialized data section.

These sections have default flags, and the linker understands the default names(similar directive to the armasm directive AREA).The following are allowable **.section flags** for ELF format files:

<Flag> Meaning

a allowable section

w writable section

x executable section

中文解释：

用户可以通过.section伪操作来自定义一个段,格式如下：

**.section section\_name [, "flags"[, %type[,flag\_specific\_arguments]]]**

每一个段以段名为开始, 以下一个段名或者文件结尾为结束。这些段都有缺省的标志（flags），连接器可以识别这些标志。(与arm asm中的AREA相同)。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：定义一个“段”

.section .mysection @自定义数据段，段名为 “.mysection”

.align 2

strtemp:

.ascii "Temp string \n\0" @ 对这一句的理解，我觉得应该是：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

**<2>** 汇编系统预定义的段名

          .text @代码段

          .data @初始化数据段 .data Read-write initialized long data.

          .bss @未初始化数据段

          .sdata @ .sdata Read-write initialized short data.

          .sbss @

注意：源程序中.bss段应该在.text段之前。

**4、  GNU汇编语言定义入口点**

汇编程序的缺省入口是**\_start**标号，**用户也可以在连接脚本文件中用ENTRY标志指明其它入口点**。

例：定义入口点

.section .data

< initialized data here>

.section .bss

< uninitialized data here>

.section .text

.globl \_start

\_start:

<instruction code goes here>

**5、  GNU汇编程序中的宏定义**

格式如下：

.macro 宏名 参数名列表 @伪指令.macro定义一个宏

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。可以使用.exitm伪指令来退出宏。

例：宏定义

.macro SHIFTLEFT a, b

.if \b < 0

MOV \a, \a, ASR #-\b

.exitm

.endif

MOV \a, \a, LSL #\b

.endm

**6、  GNU汇编程序中的常数**

**<1>** 十进制数以非0数字开头,如:123和9876；

**<2>** 二进制数以0b开头,其中字母也可以为大写；

**<3>** 八进制数以0开始,如:0456,0123；

**<4>** 十六进制数以0x开头,如:0xabcd,0X123f；

**<5>** 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”；

**<6>** 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；

**<7>** 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,其他的符号如:+、-、\*、 /、%、<、<<、>、>>、|、&、^、!、==、>=、<=、&&、|| 跟C语言中的用法相似。

**7、  GNU ARM汇编的常用伪操作**

在前面已经提到过了一些为操作，还有下面一些为操作：

          数据定义伪操作： .byte，.short，.long，.quad，.float，.string/.asciz/.ascii，重复定义伪操作.rept，赋值语句.equ/.set ；

          函数的定义；

          对齐方式伪操作 .align；

          源文件结束伪操作.end；

          .include伪操作；

          if伪操作；

          .global/ .globl 伪操作 ；

          .type伪操作 ；

          列表控制语句 ；

别于**GNU** **AS**汇编的通用伪操作,下面是ARM特有的伪操作：

.reg ，.unreq ，.code ，.thumb ，.thumb\_func ，.thumb\_set， .ltorg ，.pool

**<1>** 数据定义伪操作

         .byte:单字节定义，如：.byte 1,2,0b01,0x34,072,'s' ；

         .short:定义双字节数据，如:.short 0x1234,60000 ；

         .long:定义4字节数据，如:.long 0x12345678,23876565

         .quad:定义8字节，如:.quad 0x1234567890abcd

         .float：定义浮点数，如：.float **0f**-314159265358979323846264338327\

95028841971.693993751E-40 @ **- pi**

         .string/.asciz/.ascii：定义多个字符串，如:

.string "abcd", "efgh", "hello!"

.asciz "qwer", "sun", "world!"

.ascii "welcome\0"

注意：ascii伪操作定义的字符串需要自行添加结尾字符'\0'。

         .rept:重复定义伪操作, 格式如下：

.rept 重复次数

数据定义

.endr @结束重复定义

例：

.rept 3

.byte 0x23

.endr

         .equ/.set: 赋值语句, 格式如下：

.equ(.set) 变量名，表达式

例：

.equ abc, 3 @让abc=3

**<2>** 函数的定义伪操作

          函数的定义,格式如下：

函数名**:**

函数体

返回语句

一般的,**函数如果需要在其他文件中调用, 需要用到.global伪操作将函数声明为全局函数。**为了不至于在其他程序在调用某个C函数时发生混乱,对寄存器的使用我们需要遵循APCS准则。函数编译器将处理函数代码为一段.global的汇编码。

          函数的编写应当遵循如下规则：

**a.** a1-a4寄存器（参数、结果或暂存寄存器，r0到r3 的同义字）以及浮点寄存器f0-f3(如果存在浮点协处理器)在函数中是不必保存的；

**b.** 如果函数返回一个不大于一个字大小的值，则在函数结束时应该把这个值送到 r0 中；

**c.** 如果函数返回一个浮点数，则在函数结束时把它放入浮点寄存器f0中；

**d.** 如果函数的过程改动了sp（堆栈指针，r13）、fp（框架指针，r11）、sl（堆栈限制，r10）、lr（连接寄存器，r14）、v1-v8（变量寄存器，r4 到 r11）和 f4-f7,那么函数结束时这些寄存器应当被恢复为包含在进入函数时它所持有的值。

**<3>** .align .end .include .incbin伪操作

         .align:用来指定数据的对齐方式,格式如下:

.align [absexpr1, absexpr2]

以某种对齐方式,**在未使用的存储区域填充值**. **第一个值表示对齐方式,4, 8,16或 32.** **第二个表达式值表示填充的值。**

         .end:表明源文件的结束。

         .include:可以将指定的文件在使用.include 的地方展开,一般是头文件,例如:

.include “myarmasm.h”

         .incbin伪操作可以将原封不动的一个二进制文件编译到当前文件中,使用方法如下:

.incbin "file"[,skip[,count]]

skip表明是从文件开始跳过skip个字节开始读取文件,count是读取的字数.

**<4>** ..if伪操作

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。

.if有多个变种:

.ifdef symbol @判断symbol是否定义

.ifc string1,string2 @字符串string1和string2是否相等,字符串可以用单引号括起来

.ifeq expression @判断expression的值是否为0

.ifeqs string1,string2 @判断string1和string2是否相等,字符 串必须用双引号括起来

.ifge expression @判断expression的值是否大于等于0

.ifgt absolute expression @判断expression的值是否大于0

.ifle expression @判断expression的值是否小于等于0

.iflt absolute expression @判断expression的值是否小于0

.ifnc string1,string2 @判断string1和string2是否不相等, 其用法跟.ifc恰好相反。

.ifndef symbol, .ifnotdef symbol @判断是否没有定义symbol, 跟.ifdef恰好相反

.ifne expression @如果expression的值不是0, 那么编译器将编译下面的代码

.ifnes string1,string2 @如果字符串string1和string2不相 等, 那么编译器将编译下面的代码.

**<5>** .global .type .title .list

         .global/ .globl ：用来定义一个全局的符号，格式如下:

.global symbol 或者 .globl symbol

          .type：用来指定一个符号的类型是函数类型或者是对象类型, 对象类型一般是数据, 格式如下:

.type 符号, 类型描述

例：

.globl a

.data

.align 4

.type a, @object

.size a, 4

a:

.long 10

例：

.section .text

.type asmfunc, @function

.globl asmfunc

asmfunc:

mov pc, lr

**<6>** 列表控制语句:

.title：用来指定汇编列表的标题,例如:

.title “my program”

.list：用来输出列表文件.

**<7>** ARM特有的伪操作

          .reg: 用来给寄存器赋予别名,格式如下:

别名 .req 寄存器名

          .unreq: 用来取消一个寄存器的别名,格式如下:

　　　.unreq 寄存器别名

注意被取消的别名必须事先定义过,否则编译器就会报错,这个伪操作也可以用来取消系统预制的别名, 例如r0, 但如果没有必要的话不推荐那样做。

          .code伪操作用来选择ARM或者Thumb指令集,格式如下:

　　　　.code 表达式

　　如果表达式的值为16则表明下面的指令为Thumb指令,如果表达式的值为32则表明下面的指令为ARM指令.

          .thumb伪操作等同于.code 16, 表明使用Thumb指令, 类似的.arm等同于.code 32

          .force\_thumb伪操作用来强制目标处理器选择thumb的指令集而不管处理器是否支持

          .thumb\_func伪操作用来指明一个函数是thumb指令集的函数

          .thumb\_set伪操作的作用类似于.set, 可以用来给一个标志起一个别名, 比.set功能增加的一点是可以把一个标志标记为thumb函数的入口, 这点功能等同于.thumb\_func

          .ltorg用于声明一个数据缓冲池(literal pool)的开始,它可以分配很大的空间。

          .pool的作用等同.ltorg

          .space <number\_of\_bytes> {,<fill\_byte>}

分配number\_of\_bytes字节的数据空间，并填充其值为fill\_byte，若未指定该值，缺省填充0。（与armasm中的SPACE功能相同）

          .word <word1> {,<word2>} …

**插入一个32-bit的数据队列。**（与armasm中的DCD功能相同）。**可以使用.word把标识符作为常量使用。**

例：

Start:

valueOfStart:

.word Start

**这样程序的开头Start便被存入了内存变量valueOfStart中。**

          .hword <short1> {,<short2>} …

插入一个16-bit的数据队列。（与armasm中的DCW相同）

**8、** GNU ARM汇编特殊字符和语法

<1>    代码行中的注释符号: ‘@’

<2>    整行注释符号: ‘#’

<3>    语句分离符号: ‘;’

<4>    立即数前缀: ‘#’ 或 ‘$’