

KF32 系列

ChipON IDE 汇编语言开发手册 V1.0

2019 年 10 月

目录

目录	2
1 工具链基础知识	6
1.1 简介	6
1.2 文件命名约定	6
1.3 数据存储	6
1.3.1 字节序	6
1.3.2 整型类型与范围	6
1.3.3 默认字节 <i>char</i> 变量的符号说明	7
1.3.4 浮点类型与长度	7
1.3.5 指针类型与长度	7
1.4 属性	7
1.4.1 <i>Interrupt</i> 中断函数	7
1.4.2 <i>section</i> 指定类型段	7
1.4.3 配合伪指令的完整汇编代码函数示例	8
1.5 命令行格式	8
1.5.1 设置环境变量	8
1.5.2 命令行选项	8
2 汇编语言与库说明	9
2.1 简介	9
2.2 汇编伪指令	9
2.2.1 <i>.abort</i>	10
2.2.2 <i>.align abs-expr, abs-expr, abs-expr</i>	10
2.2.3 <i>.ascii "string"</i>	10
2.2.4 <i>.asciz "string"</i>	10
2.2.5 <i>.balign[wl] abs-expr, abs-expr, abs-expr</i>	10
2.2.6 <i>.byte expressions</i>	11
2.2.7 <i>.comm symbol, length</i>	11
2.2.8 <i>.data subsection</i>	11
2.2.9 <i>.desc symbol, abs-expression</i>	12
2.2.10 <i>.double flonums</i>	12
2.2.11 <i>.else</i>	12
2.2.12 <i>.elseif</i>	12
2.2.13 <i>.end</i>	12
2.2.14 <i>.endfunc</i>	12
2.2.15 <i>.endif</i>	12
2.2.16 <i>.equ symbol, expression</i>	12
2.2.17 <i>.err</i>	13

2.2.18 .exitm.....	13
2.2.19 .export symbol.....	13
2.2.20 .extern.....	13
2.2.21 .fail expression.....	13
2.2.22 .fill repeat, size, value.....	13
2.2.23 .float flonums.....	14
2.2.24 .func name[, label].....	14
2.2.25 .global symbol, .globl symbol.....	14
2.2.26 .hidden names.....	14
2.2.27 .hword expressions.....	14
2.2.28 .ident.....	14
2.2.29 .if absolute expression.....	14
2.2.30 .incbin "file"[, skip[, count]].....	15
2.2.31 .include "file".....	16
2.2.32 .int expressions.....	16
2.2.33 .internal names.....	16
2.2.34 .irp symbol, values	16
2.2.35 .irpc symbol, values.	17
2.2.36 .lcomm symbol , length.....	17
2.2.37 .line line-number.....	17
2.2.38 .ln line-number.....	17
2.2.39 .list.....	17
2.2.40 .long expressions.....	18
2.2.41 .macro.....	18
2.2.42 .nolist.....	19
2.2.43 .octa bignums.....	19
2.2.44 .org new-lc , fill.....	19
2.2.45 .p2align[w1] abs-expr, abs-expr, abs-expr.....	19
2.2.46 .previous.....	20
2.2.47 .popsection.....	20
2.2.48 .print string.....	20
2.2.49 .protected names.....	20
2.2.50 .psize lines , columns.....	21
2.2.51 .purgem name.....	21
2.2.52 .pushsection name , subsection.....	21
2.2.53 .quad bignums.....	21
2.2.54 .rept count.....	21
2.2.55 sbttl "subheading".....	22
2.2.56 .section name.....	22
2.2.57 .set symbol, expression.....	22
2.2.58 .short expressions.....	22
2.2.59 .single flonums.....	22
2.2.60 .size name , expression.....	23

2.2.61 .sleb128 expressions.....	23
2.2.62 .skip size , fill.....	23
2.2.63 .space size , fill.....	23
2.2.64 .stabd, .stabn, .stabs.....	23
2.2.65 .string "str".....	24
2.2.66 .struct expression.....	24
2.2.67 .symver.....	24
2.2.68 .text subsection.....	25
2.2.69 .title "heading".....	25
2.2.70 .type name , type description.....	25
2.2.71 .version "string".....	26
2.2.72 .vtable_entry table, offset.....	26
2.2.73 .vtable_inherit child, parent.....	26
2.2.74 .weak names.....	26
2.2.75 .word expressions.....	26
2.3 汇编自定义伪指令.....	27
2.3.1 特殊伪指令 LD Ra, #label.....	27
2.3.2 特殊伪指令 MOV Ra, #data.....	27
2.3.3 相对跳转 SJMP/LJMP/JMP.....	28
2.3.4 条件跳转 JZ JNZ 等编写.....	28
2.4 汇编指令.....	28
2.4.1 条件执行编码.....	30
2.4.2 指令功能分类.....	30
2.5 RAM 函数.....	38
3 中断函数.....	38
3.1 简介.....	38
3.2 中断处理函数.....	39
3.2.1 中断函数现场保护.....	39
3.2.2 中断处理程序.....	39
3.2.3 中断向量配置.....	39
4 特殊功能寄存器的操作.....	40
4.1 特殊功能寄存器的操作.....	40
4.2 寄存器使用约定.....	41
附录 1：中断向量表实现示例与说明.....	42
附录 2：示例变量与函数格式书写.....	43
对照 C 代码示例.....	43
汇编文件代码编写示例.....	43
引入宏的常规信息简化.....	45
C 文件使用汇编文件变量与函数.....	45

在使用本手册前，请您认真阅读以下使用许可协议。只有在同意以下使用许可协议的情况下，方能使用本手册中介绍的产品。

版权公告

未经上海芯旺微电子科技有限公司书面允许，任何公司、个人不得以任何形式复制本使用手册的全部或部分内容。

重要声明

上海芯旺微电子科技有限公司努力使本手册中提供的信息准确和适用，然而，产品及手册可能包括技术或印刷上的错误。上海芯旺微电子科技有限公司保留在不事先通知的情况下改变本使用手册全部或部分内容的权力。

1 工具链基础知识

1.1 简介

C 编译器(kf32-gcc)对 C 语言模块及库文件进行编译；汇编器(kf32-as)对汇编语言模块及库文件进行汇编；链接器(kf32-ld)对目标文件及库文件进行链接，同时生成 HEX 文件。

1.2 文件命名约定

KF32 编译工具链识别如下文件扩展名：

表 2-1 文件名

扩展名	定义
*.c	C 语言源文件。
*.cpp	C++语言源文件。
*.asm	汇编语言源文件。
*.h	C 语言头文件。
*.inc	汇编头文件。
*.i	已经过预处理的 C 源文件。
*.s	C 编译生成的汇编文件。
*.o	目标文件。
*.elf	ELF 文件(可执行可链接文件)。
*.map	MAP 文件。
*.lst	反汇编文件。
*.hex	Intel HEX 文件。
其他	要传递给链接器的文件。

1.3 数据存储

1.3.1 字节序

KF32 编译器以小端字节序格式存储数据。最低有效字节存储在最低地址。
举例地址 0x10000000 处开始 32 位数值 0x12345678，存储格式如下：

地址	0x10000000	0x10000001	0x10000002	0x10000003
数值	0x78	0x56	0x34	0x12

1.3.2 整型类型与范围

KF32 编译器基本数据类型与范围如下：

表 2-2 整型类型

类型	位	字节	最小值	最大值
signed char	8	1	-128	127
char,unsigned char	8	1	0	255

short, signed short	16	2	-32768	32767
unsigned short	16	2	0	65535
int, signed int	32	4	-2^{31}	$2^{31}-1$
unsigned int	32	4	0	$2^{32}-1$
long, signed long	32	4	-2^{31}	$2^{31}-1$
unsigned long	32	4	0	$2^{32}-1$
long long, signed long long	64	8	-2^{63}	$2^{63}-1$
unsigned long long	64	8	0	$2^{64}-1$

1.3.3 默认字节 char 变量的符号说明

默认情况下,不带修饰符的 char 类型的值被 KF32 编译器定义为无符号值。同时,可以使用命令行选项 -fsigned-char 将默认类型设置为有符号, -funsigned-char 将默认类型设置为无符号。

1.3.4 浮点类型与长度

KF32 编译器使用 IEEE-754 浮点格式。

表 2-3 浮点类型

类型	位	字节
float	32	4
double	64	8
long double	64	8

1.3.5 指针类型与长度

KF32 编译器中指针长度为 32 位整数。

1.4 属性

汇编语言需要编写完整的代码,但可以使用伪指令进行数据、条件等的定义。

1.4.1 Interrupt 中断函数

在 C 语言中使用中:使用 __attribute__((interrupt)) 修饰函数。即函数处理需要在首尾添加适当的压栈出栈保护。但在汇编语言中,所有的 Rx 使用情况需要自行做应用使用, R0-R4 中断和中断外代码具有独立的资源,即不用进行保护。中断函数仅通过向量表中 weak 修饰,后续直接编写代码即可。

1.4.2 section 指定类型段

在 C 语言中,将函数或变量放入由 “name” 指定的段。

例如, void __attribute__((section(“new_sect1”))) foo()
{return;}

函数 foo 将被放入 new_sect1 段。

unsigned int var __attribute__((section(“new_sect2”)))

变量 var 将被放入 new_sect2 段。

常见的段定义为 text (flash 空间), data(RAM 空间)。

在汇编语言中,直接使用伪指令 section 进行段类型定义,如:

```
.section .text$_NMI_exception
.section .text
.section .indata
```

需要注意的是汇编语言没有准确的函数概念，即 `section` 对后面的代码仍然有效，除非遇到新的 `section` 声明。

1.4.3 配合伪指令的完整汇编代码函数示例

```
.section .text$_NMI_exception //指定存储空间，可以$间隔扩展独立名
.align 1 //指定对齐方式,1:地址 2 对齐,2:地址 4 对齐
.func _NMI_exception // .func .endfunc 是确保能调试控制的前提
.export _NMI_exception // 导出符号，可供外部识别。
.type _NMI_exception, @function //定义类型
_NMI_exception: // 标签，这里等效函数名字意义
    JMP lr
    ... // 函数自身的代码
.size _NMI_exception, .-_NMI_exception //代码段大小内存字节大小
.endfunc
典型段名: (代码) .text .text$XXX
          (变量) .data .data$XXX .bss$XXX .comm .comm$XXX
          (RAM 函数) .indata
```

1.5 命令行格式

1.5.1 设置环境变量

在使用命令行环境之前，先设置环境变量。

在命令行下输入如下命令添加编译命令至环境变量(以 IDE 默认安装目录为例):

```
PATH=XXXX;%PATH%
```

在命令行下输入如下命令关闭 DOS 路径警告信息“MS-DOS style path detected”:

```
set CYGWIN=nodosfilewarning
```

ChipON IDE 环境自动集成了环境的自动配置。

1.5.2 命令行选项

KF32 编译工具链提供了许多控制编译、链接的选项，**区分大小写**。

表 2-4 C 编译器命令行选项

命令	说明
-O0	不使用优化编译
-O2	开启时间优化编译
-Os	开启空间优化编译
-I dir	添加头文件搜索路径
-o file	生成输出文件 file
-gstabs+	调试编译选项
-c	编译、汇编，但不链接
-save-temps	保留中间文件

-Wno-packed-bitfield-compat	不警告位域 packed 属性
-funsigned-char	设置 char 类型为无符号类型
-fsigned-char	设置 char 类型为有符号类型
-funsigned-bitfields	设置位域为无符号类型
-fsigned-bitfields	设置位域为有符号类型
-ffunction-sections	函数独立为 section
-fdata-sections	全局变量独立为 section
-Wa,<options>	传递逗号分隔的命令给汇编器
-Wl,<options>	传递逗号分隔的命令给链接器

表 2-5 汇编器命令行选项

命令	说明
-I dir	添加库头文件搜索路径
-o file	生成输出文件 file

表 2-6 链接器命令行选项

命令	说明
-L dir	添加库文件搜索路径
-l libname	链接库文件
-T file	获取链接脚本
--gc-sections	删除无效段(section)
-o file	生成输出文件 file
-Map file	生成输出 map 文件

一般情况下，不需要修改属性，ChipON IDE 集成了常规的语言属性，比较常见的修改属性包括添加库或引用路径，是否输出调试信息，编译优化等级等选项。

2 汇编语言与库说明

2.1 简介

将一些常规方法集成到库，方便用户程序开发与功能集成。

KF32 工具库位于编译器安装目录的 lib 子目录中，通过 KF32 链接器将这些库直接链接到应用程序中。

如果汇编语言调用 C 语言形式的函数是，根据 C 语言规则传递参数并调用函数即可。

2.2 汇编伪指令

所有的汇编器命令名都由句号('.')开头。命令名的其余是字母,通常使用小写。

2.2.1 .abort

本命令立即停止汇编过程。这是为了兼容设计。如果发送源码的程序要退出，它可以使用本命令通知 `as` 退出。将来可能不再支持使用 `.abort`。

2.2.2 .align *abs-expr*, *abs-expr*, *abs-expr*

增加位置计数器(在当前的子段)使它指向规定的存储边界。

第一个表达式参数(结果必须是纯粹的数字)是必需参数：边界基准,见后面的描述。

第二个表达式参数(结果必须是纯粹的数字)给出填充字节的值，用这个值填充位置计数器越过的地方。这个参数(和逗号)可以省略，如果省略它，填充字节的值通常是 0。但在某些系统上，如果本段标识为包含代码，而填充值被省略，则使用 `no-op` 指令填充这个空间。

第三个参数表达式的结果也必须是纯粹的数字，这个参数是可选的。如果存在第 3 个参数，它代表本对齐命令允许越过字节数的最大值。如果完成这个对齐需要跳过的字节比指定的最大值还多,则根本无法完成对齐。您可以在边界基准后简单地使用两个逗号，以省略填充值参数(第二参数)；如果您想在适当的时候，对齐操作自动使用 `no-op` 指令填充，这个方法将非常奏效。

第一个表达式是边界基准，单位是字节。例如 `‘.align 8’` 向后移动位置计数器至 8 的倍数。如果地址已经是 8 的倍数，则无需移动。

2.2.3 .ascii "string"...

`.ascii` 可不带参数或者带多个由逗号分开的字符串。它把汇编好的每个字符串(在字符串末不自动追加零字节)存入连续的地址。

2.2.4 .asciz "string"...

`.asciz` 类似与 `.ascii`,但在每个字符串末自动追加一个零字节。`‘.asciz’` 中的 `‘z’` 代表“zero”。

2.2.5 .balign[*wl*] *abs-expr*, *abs-expr*, *abs-expr*

增加位置计数器(在当前子段)使它指向规定的存储边界。第一个表达式参数(结果必须是纯粹的数字)是必需参数：边界基准,单位为字节。例如，`‘.balign 8’` 向后移动位置计数器直至计数器的值等于 8 的倍数。如果位置计数器已经是 8 的倍数，则无需移动。

第 2 个表达式参数(结果必须是纯粹的数字)给出填充字节的值，用这个值填充位置计数器越过的地方。第 2 个参数(和逗号)可以省略。如果省略它，填充字节的值通常是 0。但在某些系统上，如果本段标识为包含代码，而填充值被省略，

则使用 `no-op` 指令填充空白区。

第 3 个参数的结果也必须是纯粹的数字，这个参数是可选的。如果存在第 3 个参数，它代表本对齐命令允许跳过字节数的最大值。如果完成这个对齐需要跳过的字节数比规定的最大值还多，则根本无法完成对齐。您可以在边界基准参数后简单地使用两个逗号，以省略填充值参数(第二参数)；如果您在想在适当的时候，对齐操作自动使用 `no-op` 指令填充，本方法将非常奏效。

`.balignw` 和 `.balignl` 是 `.balign` 命令的变化形式。`.balignw` 使用 2 个字节来填充空白区。`.balignl` 使用 4 字节来填充。例如，`.balignw 4,0x368d` 将地址对齐到 4 的倍数，如果它跳过 2 个字节，GAS 将使用 `0x368d` 填充这 2 个字节(字节的确切存放位置视处理器的存储方式而定)。如果它跳过 1 或 3 个字节，填充值则不明确。

2.2.6 .byte expressions

`.byte` 可不带参数或者带多个表达式参数，表达式之间由逗号分隔。每个表达式参数都被汇编成下一个字节。

2.2.7 .comm symbol , length

`.comm` 声明一个符号名为 `symbol` 的通用符号(common symbol)。当连接时，目标文件中的通用符号可能被并入其它目标文件中已定义的符号，或者被并入其他目标文件中同名通用符号。如果 `ld` 无法找到该符号的定义——只有一个或多个通用符号——则分配 `length` 个字节的未初始化内存。`length` 必须是一个纯粹的表达式。如果 `ld` 发现多个同名的通用符号，并且它们的长度不同，`ld` 将按照它们之中最大的 `length` 值为符号分配内存。

`.comm` 可以使用第 3 个参数。它代表符号需要对齐的边界基准(例如，边界基准为 16 时意味着符号存放地址的最低 4 位应该是零)。第 3 个参数表达式结果必须是纯粹的数字，而且一定是 2 的幂。当 `ld` 为通用符号分配未初始化内存时，在存放符号时要使用到这个参数。如果没有规定边界基准，`as` 将把边界基准设置成以 2 为底的该符号长度的对数，并向下取整。最大值为 16。

`.comm` 的语法在 HPPA 上稍微有些不同。语法是 `'symbol .comm, length'`；其中参数 `symbol` 是可选的。

2.2.8 .data subsection

`.data` 通知 `as` 汇编后续语句，将它们追加在编号为 `subsection`(`subsection` 必须是纯粹的表达式)数据段末。如果参数 `subsection` 省略，则默认是 0。

2.2.9 .desc symbol, abs-expression

本命令用一个纯粹表达式的低 16 位的值设置符号 symbol 的描述符。

2.2.10 .double flonums

.double 后跟着零个或由逗号分开多个的浮点数。本指令汇编出浮点数字。生成的浮点数的确切类型视 as 的配置而定。

2.2.11 .else

.else 是支持 as 进行的条件汇编指令之一;如果前面.if 命令的条件不成立,则表示需要汇编.else 后的一段代码。

2.2.12 .elseif

.elseif 是支持 as 进行的条件汇编指令之一。它可以在.eslc 段中快速产生一个新的.if 块。

2.2.13 .end

.end 标记着汇编文件的结束。as 不处理.end 命令后的任何语句。

2.2.14 .endfunc

.endfunc 标志着一个由.func 命令定义的函数的结束。

2.2.15 .endif

.endif 是支持 as 进行的条件汇编的指令之一.它标志着条件汇编代码块的结束。

2.2.16 .equ symbol, expression

本命令把符号 symbol 值设置为 expression。它等同与'.set'命令。

.equiv symbol, expression

.equiv 类似与.equ & .set 命令, 不同之处在于, 如果符号已经定义过, as 会发出错误信号。

除了错误信息的内容之外, 它大体上等价与:

```
.ifndef SYM  
.err  
.endif  
.equ SYM,VAL
```

2.2.17 .err

如果 as 汇编一条.err 命令，将打印一条错误信息，除非使用了 -Z 选项，as 不会生成目标文件。可以在条件编译代码中使用它来发出错误信息。

2.2.18 .exitm

从当前宏定义体中提前退出。

2.2.19 .export symbol

全局变量的描述。使符号 symbol 对连接器 ld 可见。

2.2.20 .extern

.extern 可以在源程序中使用--以便兼容其他的汇编器—但会被忽略。as 将所有未定义的符号都当作外部符号处理。

2.2.21 .fail expression

生成一个错误(error)或警告(warning)。如果 expression 的值大于或等于 500，as 会打印一条“警告”消息。如果 expression 的值小于 500,as 会打印一条“错误”消息。消息中包含了 expression 的值。这在复杂的宏嵌套或条件汇编时偶尔用到。

2.2.22 .fill repeat, size, value

该汇编指令会产生数个(repeat 个)大小为 size 字节的重复拷贝。大小值 size 可以为 0 或者某个值，但是若 size 大于 8，则限定为 8。每个重复字节内容取自一个 8 字节数。高 4 字节值为 0，低 4 字节的数值 value。这 3 个参数都是绝对值，size 和 value 是可选的。如果第二个逗号和 value 省略，value 默认值为 0 值；如果后连个参数都省略，则 size 默认值为 1。如.fill 1,1 对应一个字节的 0 值，往往对应全局未初始化变量的内容表达。

2.2.23 .float flonums

本命令汇编 0 个或多个浮点数，浮点数之间由逗号分隔。它和 .single 的汇编效果相同。

2.2.24 .func name[,label]

.func 发出一个调试信息用以指示函数 name，这个信息将被忽略，除非文件使用调试使能方式的汇编。目前只支持‘—gstabs[+]’。label 是函数的入口点，如果 name 被省略则使用预定的‘引导符’。‘引导符’通常可以是_ 或者什么也没有，视目标机设计而定。所有函数当前被定义为 void 返回类型，函数体必须使用 .endfunc 来结束

2.2.25 .global symbol, .globl symbol

.global 使符号 symbol 对连接器 ld 可见。如果您在局部过程中定义符号 symbol，其它和此的局部过程都可以访问它的值。另外，symbol 从连接到本过程的另一个文件中的同名符号获取自己的属性。另外可以使用 .export 进行辅助变量的描述。

2.2.26 .hidden names

这是一条可见度的命令。其它两条是 .internal 和 .protected。本命令取消指定符号的缺省可见度(可见度由其他命令捆绑设定：local,global,weak)。本命令把可见度设置为 hidden,这意味着本符号对其他部分不可见。这最好是一些需要长期保护的符号。

2.2.27 .hword expressions

本命令后可以不带或带多个 expressions,并且为每个参数生成一个 16 位数。本命令等同与'.short'命令。

2.2.28 .ident

本命令被汇编器用来在目标文件中加入标饰，as 简单地接受本命令，但实际上不产生东西。

2.2.29 .if absolute expression

.if 标志着一段代码的开始，这段代码只有在参数 absolute expression(必须是

一个独立的表达式)不为 0 时才进行汇编。这段条件汇编代码必须使用 `.endif` 标志结束。另外, 可以使用 `.esle` 来标记一个代码块, 这个代码块与前面那段代码只有一个会进行汇编。如果您需要检查数个汇编条件, 可以在使用 `.elseif` 命令, 以避免在 `.else` 代码块中进行 `if/else` 语句块的嵌套。

同样可以使用下面 `.if` 的变体:

`.ifdef symbol`

如果指定的符号 `symbol` 已经定义过, 汇编下面那段代码。

`.ifc string1,string2`

如果两个字符串相同的话, 汇编下面那段代码。字符串可以可选地使用单引号。如果不使用引号则第 1 个字符串在逗号处结束。第 2 个字符串在本行末结束。包含空白的字符串应该使用引号标注。字符串比较时是区分大小写的。

`.ifeq absolute expression`

如果参数的值为 0, 汇编下面那段代码。

`.ifeqs string1,string2`

这是 `.ifc` 的另一种形式, 字符串必须使用双引号标注。

`.ifge absolute expression`

如果参数的值大于等于 0, 汇编下面那段代码。

`.ifgt absolute expression`

如果参数的值大于 0, 汇编下面那段代码。

`.ifle absolute expression`

如果参数的值小于等于 0, 汇编下面那段代码。

`.iflt absolute expression`

如果参数的值小于 0, 汇编下面那段代码。

`.ifnc string1,string2.`

类似与 `.ifc`, 不过使用反向的测试: 如果两个字符串不相等的话, 汇编下面那段代码。

`.ifndef symbol`

`.ifnotdef symbol`

如果指定的符号 `symbol` 不曾被定义过, 汇编下面那段代码。上面两种写法是等效的。

`.ifne absolute expression`

如果参数的值为不等于 0, 汇编下面那段代码。(换句话说, 这是 `.if` 的另一种写法)。

`.ifnes string1,string2`

类似与 `.ifeqs`, 不过使用反向的测试: 如果两个字符串不相等的话, 汇编下面那段代码。

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

这条 `incbin` 命令在当前位置逐字地引入 `file` 文件的内容。您可以使用命令行选项参数“-I”来控制搜索路径。文件名必须使用引号。

参数 `skip` 表示需要从文件头跳过的字节数目。参数 `count` 表示读入的最大字节数目。注意，数据没有进行任何方式的对齐操作，所以用户需要在 `.incbin` 命令的前后进行必要的边界对齐。

2.2.31 `.include "file"`

本命令提供在源程序中指定引入支撑文件的手段。`file` 中的代码如同紧跟 `.include` 后一样被汇编。当引入文件汇编结束，继续汇编原来的文件。您可以使用命令行选项参数“-I”来控制搜索路径。文件名必须使用引号来标注。

2.2.32 `.int expressions`

可以不带参数或带多个 `expressions`, 参数之间由逗号分隔。每个 `expressions` 都生成一个数字, 这个数字等于表达式在目标机器运行时的值。字节顺序和数字的位数视汇编的目标机器而定。

2.2.33 `.internal names`

这是一条可见度相关的命令。另外的两条是 `.hidden` 和 `.protected`。

本命令取消指定符号的缺省可见度(可见度由其他命令捆绑设定：`local, global, weak`)。本命令把指定符号可见度设置为 `internal`, 这意味着此符号需要被隐藏(即对其他部分不可见)，另外，符号还必须经过处理器的特别的处理。

2.2.34 `.irp symbol, values ...`

加工一个需要用 `values` 替代 `symbol` 的语句序列。语句序列从 `.irp` 命令开始，在 `.endr` 命令前结束。对于每个 `value` 都进行如下加工：用 `value` 替代 `Symbol`，并对此语句序列进行汇编。如果没有给出 `value`，则用空字符串(`null sting`)替代 `symbol`，并将此语句序列汇编一次。使用 `\symbol`，把参数 `symbol` 提交给语句序列。

例如下列代码

```
.irp param,1,2,3
    mov R\param,sp
.endr
```

等同与

```
mov R 1,sp
mov R 2,sp
mov R 3,sp
```


2.2.35 .irpc symbol,values. . .

加工一个需要用 values 替代 symbol 的语句序列。语句序列从.irpc 命令开始，在.endr 命令前结束。对于 value 中的每个字符，都进行如下加工：用此字符替代 symbol，并对此语句序列进行汇编。如果没有给出 value 参数，则用空字符串(null sting)替代 symbol，并将此语句序列汇编一次。使用\symbol，把参数 symbol 提交给语句序列。

例如下列代码

```
.irpc param,123
    mov R\param,sp
.endr
等同与
    mov R1,sp
    mov R2,sp
    mov R3,sp
```

2.2.36 .lcomm symbol , length

为一个本地通用符号 symbol 预留 length 个字节的内存。symbol 的段(属性)和值(属性)被设置为一个新的本地通用符号应有的属性：内存是在 bss 段中分配的，所以在运行时,这些字节开始都是零。因为 symbol 没有被声明为全局性的符号，所以 symbol 对 ld 通常不可见。

2.2.37 .line line-number

更改逻辑行号，参数 line-number 必须是个纯粹的表达式。本命令后的下一行将被赋予此逻辑行号。因此在当前行之前任何其他的语句(在语句分隔符后)的逻辑行号将被视作 line-number - 1。以后 as 将不在支持这条命令：只是为了兼容现存的汇编器而接受本命令。

尽管这是与 a. out 或 b. out 目标代码格式相关的命令，在生成 COFF 输出时 as 仍然接受它，并且如果‘.line’出现在.def/endif 之外的话，就把它视为‘.ln’命令。如果‘.line’在.def 语句块中的话，.line 命令则是一条编译器使用的命令，用来为调式生成辅助符号信息。

2.2.38 .ln line-number

‘.ln’命令等同与‘.line’。

2.2.39 .list

控制(和.nolist 命令配合)是否生成汇编清单。这两个命令维护一个内部的计

数器(计数器初始值为 0) .list 命令增加计数器的值, .nolist 减少计数器的值。当计数器的值大与 0 时将汇编列表。

缺省状态汇编列表的生成是关闭的。当您打开它的时候(使用带-a 选项的命令行), 内部计数器的初始值为 1。

2.2.40 .long expressions

.long 是.int 的等价命令。

2.2.41 .macro

本命令.macro 和.endm 命令允许您定义宏来生成汇编输出。例如, 下面的语句定义了一个宏 sum, 这个宏把一个数字序列放入内存。

```
.macro sum from=0, to=5
.long \from
.if \to-\from
sum "(\from+1)",\to
.endif
.endm
```

使用上述定义, 'SUM 0,5'语句就等于输入下面的汇编语句:

```
.long 0
.long 1
.long 2
.long 3
.long 4
.long 5
```

```
.macro macname
```

```
.macro macname macargs ...
```

开始定义一个名为 macname 的宏。如果您的宏需要使用参数, 则在宏的名字后指定他们的名字, 参数之间用逗号或空格分隔。您可以为任意的参数提供参数的缺省值, 只需要在参数后使用“=deflt”, 。例如, 下列都是合法的宏定义语句:

```
.macro comm
```

定义一个名为 comm 宏,不使用参数。

```
.macro plus1 p, p1
```

```
.macro plus1 p p1
```

两个语句都声明要定义一个名为 plus1 的宏, 这个宏需要两个参数, 在宏定义体内, 使用'p'或'p1'来引用参数的值。

```
.macro reserve_str p1=0 p2
```

声明要定义一个名为 reserve_str 的宏, 使用两个参数。第一个参数有缺省值, 第二个没有缺省值。宏定义完成后, 您可以通过'reserve_str a, b'(宏体中'p1'引用 a 的值, 'p2'引用 b 值)或通过'reserve_str ,b'('p1'使用缺省值, 在此为'0', 'p2'引用 b 的值)来调用这个宏。

当调用一个宏时，您既可以通过位置指定参数值，也可以通过关键字指定参数值。例如，‘sum 9,17’和‘sum to=17, from=9’是等价的。

`.endm` 标志宏定义体的结束。

`.exitm` 提前从当前宏定义体中退出。

`\@` 这个伪变量其实是 `as` 维护的一个计数器，用来统计执行了多少个宏。您可以通过使用 `\@` 把这个数字复制到您的输出中，但仅限于在宏定义体中使用。

2.2.42 `.nolist`

控制(和 `.list` 命令配合)是否生成汇编列表。这两个命令维护一个内部的计数器(计数器初始值为 0)。`.list` 命令增加计数器的值，`.nolist` 减少计数器的值。当计数器的值大与 0 时将汇编列表。

2.2.43 `.octa bignums`

本命令可以不带参数或多个由逗号分隔开的巨数 `bignum`，针对每个巨数 `bignum`，它生成一个 16 个字节的整数。

术语“octa”来源：word 为 2 个字节，故此 `octa-word` 为 16 个字节。

2.2.44 `.org new-lc , fill`

向后移动当前段的位置计数器至 `new-lc`。`new-lc` 要么是一个纯粹的表达式，要么这个表达式与当前子段在同一个段中。换句话说，就是您不能使用 `.org` 进行段超越。如果 `new-lc` 指向错误的段，则忽略 `.org` 命令。为了兼容以前的汇编器，如果 `new-lc` 指向一个地址独立的段，`as` 发出一个警告，并假定 `new-lc` 指向当前子段。

`.org` 仅仅可以增大位置计数器，或者保持位置计数器不变；您不能使用 `.org` 命令把位置计数器向回移动。

因为 `as` 尽量一次完成程序汇编，所以不能使用未定义的 `new-lc`。如果您厌恶这个限制，我们急切期待有机会分享经过您改进的汇编器。

注意起点相对于段的首地址，而不是子段的首地址。这与其他汇编器相兼容。

当(当前语句块)位置计数器到达指定位置，用 `fill` 填充该字节，`fill` 必须是纯粹的表达式。如果没有给出逗号和 `fill`，`fill` 值缺省为 0。

2.2.45 `.p2align[wl] abs-expr, abs-expr, abs-expr`

增加位置计数器(在当前的子段)使它指向规定的存储边界。第一个表达式参数(结果必须是纯粹的数字) 代表位置计数器移动后，计数器中连续为 0 的低序位数量。例如‘`.align 3`’向后移动位置指针直至 8 的倍数(指针的最低的 3 位为 0)。

如果地址已经是 8 倍数，则无需移动。

第二个表达式参数(结果必须是纯粹的数字)给出填充字节的值。用这个值填充位置计数器越过的地方。这个参数(和逗号)可以省略。如果省略它，填充字节的值通常默认为 0。但在某些系统上，如果本段标识为包含代码，而填充值被省略，则使用 `no-op` 指令填充填充区。

第 3 个参数表达式的结果也必须是纯粹的数字，这个参数是可选的。如果存在第 3 个参数，它代表本对齐命令允许越过字节数的最大值。如果完成这个对齐需要跳过的字节比指定的最大值还多，则根本无法完成对齐。您可以在边界基准后简单地使用两个逗号，以省略填充值参数(第二参数)；如果您想在适当的时候，对齐操作自动使用 `no-op` 指令填充，这个方法将非常奏效。

`.p2alignw` 和 `.p2alignl` 是 `.p2align` 命令的变化形式。`.p2alignw` 使用 2 个字节来填充填充区。`.p2alignl` 使用 4 字节来填充。例如，`.p2alignw 2,0x368d` 将地址对齐到 4 的倍数，如果它跳过 2 个字节，GAS 将使用 `0x368d` 填充这 2 个字节(字节的准确的位置视处理器的存储方式而定)。如果它跳过 1 或 3 个字节，填充值则不明确。

2.2.46 .previous

这是一个 ELF 段堆栈操作命令。其他的段堆栈操作命令还有 `.section`, `.subsection`, `.pushsection`, 和 `.popsection`。

本命令交换当前段(及其子段)和最近访问过的段(及其子段)。多个连续的 `.previous` 命令将使当前位置两个段(及其子段)之间反复切换。

用段堆栈的术语来说，本命令使当前段和堆顶段交换位置。

2.2.47 .popsection

这是一个 ELF 段堆栈操作命令。其他的段堆栈操作命令还有 `.section`, `.subsection`, `.pushsection`, 和 `.previous`。

本命令用堆栈顶段(及其子段)替代当前段(及其子段)。堆栈顶段出栈。

2.2.48 .print string

`as` 会在标准输出上打印 `string` 字符串。`String` 必须使用双引号。

2.2.49 .protected names

这是一条 ELF 可见度的相关命令。其它两条是 `.hidden` 和 `.internal`。

本命令将取消指定符号的可见度缺省值(可见度由其他命令捆绑设定：`local`, `global`, `weak`) 本命令将可见度设置为 `protected`, 这个可见度意味着：在定义此符号的部件内对此符号的任何访问，都必须解析到这个部件内的定义体。即使其他部件中存在一个正常情况下比此优先的定义体。

2.2.50 .psize lines , columns

当生成清单列表时，使用本命令声明每页的行数—还可以可选地声明列数。如果您不使用本命令，清单列表的行数为默认的 60 行。可以省略逗号和列参数：默认值为 200 列。

当指定的行数过多的话，as 会产生进纸操作。（如果您确实需要一个进纸动作，可以使用.eject 命令）

如果您指定行数为 0，则不产生进纸操作，除非您明确地使用了.eject 命令。

2.2.51 .purgem name

取消 name 的宏定义，后面使用字符串 name 不会被宏扩展。

2.2.52 .pushsection name , subsection

本命令是一个 ELF 段堆栈操作命令。其余的几个是.section , subsection ,.popsection, 和 .previous。

本命令与.section 命令是等价的。它将当前段(及子段)推入段堆栈的顶部。并使用 name 和 subsection 来替代当前段和子段。

2.2.53 .quad bignums

.quad 可带 0 或多个 bignum 参数，每个参数由逗号分隔。对于每个 bignum 都汇编成一个 8 字节的整数。如果某个 bignum 用 8 字节无法表示，则给出警告信息；只汇编这个 bignum 的最低 8 字节。

术语“quad”源于一个“word”代表 2 个字节，所以 quad-word 代表 8 个字节。

2.2.54 .rept count

汇编.rept 和.endr 之间的语句 count 次。

如，汇编下列语句：

```
.rept 3  
.long 0  
.endr
```

与下列语句是等价的：

```
.long 0  
.long 0  
.long 0
```

2.2.55 sbttl "subheading"

当生成汇编清单时，使用 subheading 作为标题（第 3 行，紧跟在标题行之后）。本命令对清单的后续页起作用，如果它位于当前页的前 10 行内，则对当前页也起作用。

2.2.56 .section name

本命令是 ELF 的段堆栈操作命令之一，其他的段堆栈命令为 .subsection, .pushsection, .popsection, and .previous .

当目标格式为 ELF 时，.section 命令应如下使用：

.section name [, "flags" [, @type]]

可选参数 flags 是被引号包围的字符串，可以由下列字符的任意组合：

a 可分配的段 (allocatable)

w 可写段

x 代码段

可选的参数 type 可以包含下列的任一常量：

@progbits 包含数据的段

@nobits 不包含数据的段(只占用空间的段)

如果本命令没有指定标志，则依靠段名来确定标志缺省值。如果段名不是标准的段名，则默认的该段不包含上述标志：该段不可分配内存，不可写，不可执行。该段是包含数据的段。

2.2.57 .set symbol, expression

设置 symbol 为 expression。这将改变 symbol 的值域和类型领域以符合 expression 参数。如果 symbol 已被标志为 external，则 symbol 保持它的标志。

您可以在同一个汇编程序中多次使用 .set 命令来设置同一个符号。

如果设置一个全局符号，该符号在目标文件中值为最后设定的值。

2.2.58 .short expressions

本命令通常和 '.word' 命令一样。

2.2.59 .single flonums

本命令可以汇编 0 个或多个浮点参数，各个参数之间使用逗号分隔。它的作用和 .float 相同。

2.2.60 .size name , expression

本命令经常用来设置符号 name 的内存大小。内存大小的单位是字节，通过计算参数 expression 得到，参数 expression 中可以使用标签进行计算。本命令常用来设置函数符号的长度。

2.2.61 .sleb128 expressions

sleb128 代表“signed little endian base 128”(低地址结尾的带符号 128 位基数)。这是一个紧凑的，变长的数字表示方法，当使用 DWARF 符号调试格式时使用。

2.2.62 .skip size , fill

本命令生成 size 个字节，每个字节的值都是 fill。参数 size 和 fill 都必须是纯粹的表达式。如果省略逗号和 fill,则默认 fill 的值为 0。这与'.space'相同。

2.2.63 .space size , fill

本命令生成 size 个字节，每个字节的值都是 fill。参数 size 和 fill 都必须是纯粹的表达式。如果省略了逗号和 fill,则默认 fill 的值为 0。这与'.skip'相同。

警告：在 gnu 汇编器大多数版本中，这个.space 命令和.block 命令等效。

2.2.64 .stabd, .stabn, .stabs

有 3 个以.stab 开头的命令。它们都用来产生符号，供符号调试器使用。这些符号没有收入 as 的散列表中：这些符号不能被源文件其他地方所访问。它们至少需要 5 个属性域：

string 这是符号的名字。它可以包含除'\000'之外的任何字符，故此可用名比普通符号名更广泛。很多调试器经常利用这个空间，把任意复杂的结构编码为符号名。

type 这是一个纯粹的表达式。符号的类型属性由这个表达式的低 8 位设定。任何的位组合 (bit pattern) 都可以，但连接器和调试器会被没有义的位组合所中断。

other 这是一个纯粹的表达式。由这个表达式的低 8 位设定此符号的“其它”属性。

desc 这是一个纯粹的表达式。由这个表达式的低 16 位设定此符号的描述符。

Value 这个纯粹的表达式将作为符号的值。

如果汇编.stabd, .stabn, 或 .stabs 语句时引发了一个警告，该符号有可能已经被创建；在目标文件中存在一个半成品的符号。

`.stabd type , other , desc`

生成符号的“名字”甚至不是空字符串，而是一个空指针 (null)，这样安排是出于对兼容性要求。早期的汇编器经常使用空指针，以避免空字符串在目标文件中浪费空间。

这个符号的值 (值域) 在重定位时设置为位置计数器的值。当程序连接之后，这个符号的值是 `.stabd` 命令汇编时位置计数器的地址。

`.stabn type , other , desc , value`

这个符号的名字被设置为空字符串“”。

`.stabs string , type , other , desc , value`

5 个属性域全部指定好。

2.2.65 `.string "str"`

将参数 `str` 中的字符复制到目标文件中去。您可以指定多个字符串进行复制，之间使用逗号分隔。除非另外指定了具体的机器，汇编器将在每个字符串后追加一个 0 字节作为标记。

2.2.66 `.struct expression`

切换到独立地址段，并用 `expression` 设定段的偏移量，`expression` 必须是个纯粹的表达式。您可以如下使用：

`.struct 0`

`field1:`

`.struct field1 + 4`

`field2:`

`.struct field2 + 4`

`field3:`

定义符号 `field1` 的值为 0，符号 `field2` 的值为 4，符号 `field3` 的值为 8。这段汇编程序将保存在独立地址段中，在进行下一步汇编前，您需要使用一个某种类型的 `.section` 命令，以切换到相应的段。

`.subsection name`

本命令是一个 ELF 段堆栈操作命令。其它的几个命令是。

`section` , `.pushsection` , `.popsection` , and `.previous`。

本命令用 `name` 子段替换当前子段。当前段并不改变。被替换的子段入段堆栈，成为段堆栈的新栈顶。

2.2.67 `.symver`

使用 `.symver` 命令把符号装订到在源文件里指定的节点。，如果当前汇编的文件被连接到一个共享库中时常常用到。有些情况下应该在目标文件中使用本命令，把目标文件自我装订进某个应用软件中，从而取代共享库中旧版本符号。对于 ELF 目标，`.symver` 命令可以这样使用：

`.symver name, name2@nodename`

如果符号 `name` 的定义在当前正在汇编的文件中，这个 `.symver` 命令实际用 `name2@nodename` 创建一个符号别名，而且我们不打算创建一个正规的别名，因为在符号名中是不允许存在 '@' 这个字符的。别名中 `name2` 才是符号的真正名字，外部访问是通过这个名字进行的。符号自己的名字 `name` 仅仅为了使用上的方便，这样在同一个源文件中的一个函数才可能有多个定义体；编译器才能够清楚当前使用的函数是哪个具体的定义。别名中的 `nodename` 部分应是某个节点的名字，这个节点的名字是在建立共享库时，提供给连接器的版本脚本中指定的。如果您想覆盖共享库中的旧版本符号，则 `nodename` 应该是将被取代符号的节点名。

如果符号 `name` 的定义不在当前正在汇编的文件中，则所有对 `name` 的访问都变为对 `name2@nodename` 的访问。如果根本没有对 `name` 的访问，将会把 `name2@nodename` 从符号表中删除。

`.symver` 命令的另一种用法：

`.symver name, name2@@nodename`

在这种情况下，符号 `name` 必须存在，并且它必须在当前正在汇编的文件中被定义。这类似与 `name2@nodename`。区别是 `name2@@nodename` 还被连接器用来解析对 `name2` 的访问。//注：对 `name2` 的访问被转向到 `nodename`

`.symver` 命令的第 3 种用法：

`.symver name, name2@@@nodename`

如果 `name` 不是在当前正在汇编的文件中被定义的时候，对符号的处理就如同 `name2@nodename`。如果 `name` 是当前正在汇编的文件中定义的，符号的名字 `name`，会被转换为 `name2@@nodename`。

2.2.68 .text subsection

通知 `as` 把后续语句汇编到编号为 `subsection` 的正文子段的末尾，`subsection` 是一个纯粹的表达式。如果省略了参数 `subsection`，则使用编号为 0 的子段。

2.2.69 .title "heading"

当生成汇编清单时，把 `heading` 作为标题使用（标题在第 2 行，紧跟在源文件名和页号后）。

如果这个命令出现在某页的前 10 行中，它不但作用影响到后续的页，也同样影响到当前页。

2.2.70 .type name , type description

本命令经常用来设置符号 `name` 的类型（属性）为函数符号或是目标符号两者之一。`type description` 部分允许使用 5 种不同的语法，以兼容众多的汇编器。这些语法是：

`.type <name>, #function`

`.type <name>, #object`

```
.type <name>,@function
.type <name>,@object

.type <name>,%function
.type <name>,%object

.type <name>,"function"
.type <name>,"object"

.type <name> STT_FUNCTION
.type <name> STT_OBJECT
```

2.2.71 .version "string"

本命令创建一个.note 段，并把一个 NT VERSION 类型 ELF 格式的 note 放入该.note 段。Note 的名字被设置为 string。

2.2.72 .vtable_entry table, offset

本命令寻找或创建一个符号表，并用 offset 作偏移量的增量，为此符号表产生一个 VTABLE_ENTRY 重定位。

2.2.73 .vtable_inherit child, parent

本命令寻找符号 child，并寻找或创建符号 parent，为符号 parent 产生一个 VTABLE_INHERIT 重定位，parent 的偏移量增量为符号 child 的值。一个特例，如果 parent 的名字为 0，则将它交给*ABS*段处理。

2.2.74 .weak names

本命令设置 names 中每个符号(由逗号分隔)的 weak 属性。如果这些符号尚不存在，则创建这些符号。

2.2.75 .word expressions

本命令可不带表达式或带多个表达式，这些表达式可以属于任意段，每个表达式由逗号分隔。

汇编生成的数字的大小，字节顺序视生成程序运行的目标机器而定。

2.3 汇编自定义伪指令

2.3.1 特殊伪指令 LD Ra,#label

用途：加载 32 位立即数(label 为 32 位立即数，符号或地址)

说明：该伪指令将 label 数值存入 flash 空间中常量表，并使用 LD Ra, [pc + #offset8] 加载，可以节约代码空间，如果有多个相同的常量还可以节约常量保存数量。

该伪指令执行 LD Ra, [pc + #offset8]，周期为 2，存储空间最小为 6 字节。

一般情况下，加载 32 位立即数时，使用该伪指令优于 MOVL/MOVH 指令。并且常量表输出在函数结尾，无需额外代码；函数代码较多，使常量表偏移超出 LD Rd, L 最大跳转范围 1024 字节时，常量表将插入在函数代码中，并通过添加 SJMP 指令连接函数代码。也就是说因为要生成常量表，如果在纯汇编语言中使用该伪指令，需要将汇编的代码按照函数段格式进行编写，即使用 .text 修饰新的起始，此时函数使用 LD 指令的后面代码应该小于当前指令的偏移可到底范围，纯汇编函数不支持函数中间插入常量表。

用法：

1、加载立即数

```
LD R5, #0X12345678 //将立即数 0x12345678 加载入 R5 中
```

2、加载符号

```
LD R5, #FUNCTION //将子函数入口地址 FUNCTION 加载至 R5
```

```
LJMP R5 //同调用子函数，并将子函数返回地址存入 LR
```

3、加载地址

```
LD R5, #OSC_CTL1 //将寄存器 OSC_CTL1 的地址 0x40000004 加载至 R5
```

```
LD.W R4, [R5] //读寄存器 OSC_CTL1
```

```
LD R3, #0X55AA55AA //准备向寄存器 OSC_CTL1 写入 0X55AA55AA
```

```
ST.W [R5], R3 //写寄存器 OSC_CTL1
```

```
SET [R5], #_FSCM //寄存器 OSC_CTL1 第 16 位(FSCM)置 1。
```

2.3.2 特殊伪指令 MOV Ra,#data

用途：加载 32 位立即数(label 为 32 位立即数或符号)

说明：该伪指令将依据 Ra 的使用及 data 的数值大小，选择 MOV R, #data8、MOVL R, #data16 或 MOVL/MOVH 实现。所以当 data 小于 0x10000 时，Ra 为 R0-R12 或 LR 时，此伪指令效率优于 LD Ra, #label。

因此，16 位的 MOV 指令，在 data 的不同取值的情况下，会被编译为一条 32 位指令或两条 32 位指令。

纯汇编语言下需要注意该伪指令的指令使用为条数或周期不确定，附近代码的跳转表达不要使用绝对数的表示。如 JMP \$-8 JMP \$+8。一旦该间隔范围内有使用这条伪指令，但数据范围根据需要跳转的处理条数不确定，此时跳转位置将与预期不一致。

2.3.3 相对跳转 SJMP/LJMP/JMP

相对跳转指令格式：

序号	格式	说明
1	SJMP/JMP/LJMP label	跳转至 label 处
2	SJMP/JMP/LJMP \$	表示原地跳转
3	SJMP/JMP/LJMP \$-M	表示向前跳转 M 条 16 位指令长度
4	SJMP/JMP/LJMP \$+M	表示向后跳转 M 条 16 位指令长度

JMP label 为 32 位指令，SJMP label 为 16 位指令，LJMP 分 16 位指令和 32 位指令。

对于序号 1，编译器将通过判断 label 的距离，优先选择 SJMP label 指令实现跳转功能。

后三种格式中，JMP 指令编码为 32 位，SJMP 编码为 16 位，不进行指令转换。LJMP 会执行超范围的提升。使用注意事项：

- 无法识别跳转范围内的 32 位指令，则会出现编译器无法发现的跳转错误。如下述例子中，MOVL 为 32 位指令，JMP 指令则会将 PC 跳转至 MOVL 指令编码的第 16 位，致使程序执行出错。

```
MOVL R5, #0x1234
```

```
JMP $-1
```

- 伪指令 LD R, #label 生成的常量表较长时，用户代码会被常量表截断，此时通过用户源码无法发现。所以可能跳入常量地址，以至出现编译器无法发现错误。

建议用户在使用这些跳转指令时，全部采用标号形式的跳转方式实现，如下：

```
JMP_BACK_ONE_INST:
```

```
MOVL R5, #0x1234
```

```
JMP JMP_BACK_ONE_INST //替代 JMP $-2 ,向上跳过一条指令
```

```
JMP_ENDLESS:
```

```
JMP JMP_ENDLESS //替代 JMP $ ,原地跳转
```

2.3.4 条件跳转 JZ JNZ 等编写

该机器指令为条件成立时，跳到转 PC=PC+SignExtend(imm8, "0")。这里的目标地址根据 (-128-127)*2 的偏移跳转，即支持约 512 字节的指令范围跳转。

编程代码时可以如 JMP 编写方法，即建立标签并使用标签表达，如

```
LabelA:
```

```
SUB R0, #1
```

```
LabelB:
```

```
CMP R0, #0
```

```
JNZ LabelA
```

2.4 汇编指令

汇编指令以数据手册介绍为准。

下文的指令说明中，将通过指令说明、指令操作、指令操作数、执行周期、受影响标志

位、指令编码和示例等方式进行说明，如下所示。

指令说明：描述汇编指令的格式。

指令操作：使用操作符号描述指令执行的操作，符号如下

= :结果赋值

== :逻辑等

!= :不等于

+ :加运算

- :减运算

NOT:按位取反操作

& :与操作

| :或操作

^ :异或操作

<< :逻辑左移

[] :寄存器寻址

.H :对高16 位操作

.L :对低16 位操作

.B :对低4 位操作

ZeroExtend(imm, "numb", 32):将imm 低位补numb，并零拓展成32 位，numb 可无。

SignExtend(imm, "numb", 32):将imm 低位补numb，并符号拓展成32 位，numb 可无。

{list}:分别对应LR/PC,R12-R6，当相应位置1 时，表示选中相应的Rx。

条件执行编码，按照N/Z/C/V 标志位的值来判断是否执行。

labelx: x 位相对PC 的偏移地址

offsetx: x 位偏移地址

bit5: 5 位操作数,指示第0 到第31 位。

immx: x 位立即数

逻辑左移: 左移，低位补0。

逻辑右移: 右移，高位补0

算术右移: 右移，高位补符号位。

循环右移: 右移，低位移出位补高位。

字对齐: 地址的低2 位无效。

半字对齐: 地址的最低位无效。

字节对齐: 所有地址均有效。

指令操作数：描述指令中用户可用的操作数

执行周期：描述指令执行阶段需要的周期数

受影响标志位：描述指令执行影响的标志位(N/Z/C/V)，不涉及的标志位会保持。

影响标志位的具体操作如下：

N: 当结果最高位为1(即结果为负数时)，使N 置1，否则清零。

Z: 当结果为0，使Z 置1，否则清零。

C: 当结果的最高位仍有进位时，使C 置1，否则清零。

V: 当结果溢出时(最高位和次高位进位异或为1)，将V 置1，否则清零。

指令不涉及的标志位会保持。

示例:描述指令使用的方式及具体的变化过程。

2.4.1 条件执行编码

跳转指令可以根据N/Z/C/V 标志位来决定是否转型跳转，具体如下表所示：

助记简写	助记全写	执行条件
JZ	Zero	$Z==1$
JNZ	Not Zero	$Z==0$
JC	Carry	$C==1$
JNC	No Carry	$C==0$
JMI	Minus,negative	$N==1$
JPL	Plus,positive or zero	$N==0$
JVS	Overflow	$V==1$
JVC	No Overflow	$V==0$
JHI	Unsigned Higher	$C==1 \ \& \ Z==0$
JLS	Unsigned Lower or Same	$C==0 \ \ Z==1$
JGE	Signed Greater or Equal	$N==V$
JLT	Signed Less Than	$N!=V$
JGT	Signed Greater Than	$Z==0 \ \& \ N==V$
JLE	Signed Less or Equal	$Z==1 \ \ N!=V$

2.4.2 指令功能分类

KungFu32 精简指令共有130 条，按功能可以分为如下所示的功能分类。

❖ 传送指令 7 条

序号	汇编语法	说明	影响标志位	周期数	限定
1	MOV Rd,#imm8	将 8 位立即数零拓展成 32 位送到 Rd	N/Z	1	$Rd \in [R0...R15]$ $imm8 \in [0...255]$
2	MOV Rd,Rs	将 Rs 中内容送到 Rd		1	$Rd \in [R0...R15]$ $Rs \in [R0...R15]$
3	MOVL Rd,#imm16	将 16 位无符号立即数送到 Rd,高半字清零		2	$Rd \in [R0...R15]$ $imm16 \in [0..65535]$
4	MOVH Rd,#imm16	将16 位立即数送到Rd 高半字,低半字内容不变		2	$Rd \in [R0...R15]$ $imm16 \in [0..65535]$
5	MOV SYS,Rs	将 Rs 中内容送到 MSP (0) 或 PSP (1)		1	$SYS \in [0,1]$ $Rs \in [R0...R15]$
6	MOV Rd,SYS	将 MSP 或 PSP 中内容送到 Rs		1	$SYS \in [0,1]$ $Rd \in [R0...R15]$
7	XCH Rd,Rs	将 Rd 和 Rs 中内容互换		2	$Rd \in [R0...R15]$ $Rs \in [R0...R15]$

❖ 存取指令 39 条

序号	汇编语法	说明	影响标志位	周期数	限制
1	LD.W Rd,[PC+#offset8]	Rd = [PC+ZeroExtend#(offset8,"00")] 将 (PC+ZeroExtend(#offset8,"00")) 指向的 32 位值保存到 Rd 中, PC 不变, 地址字对齐		2	Rd ∈ [R0...R7] offset8 ∈ [0...255]
2	LD.B Rd,[sp+#offset5]	Rd = [sp + ZeroExtend(#offset5)] 将 (sp + ZeroExtend (#offset5)) 指向的 8 位值高位 0 扩展成 32 位保存到 Rd 中, sp 不变, 地址字节对齐		2	Rd ∈ [R0...R7] Offset5 ∈ [0...31]
3	LD.H Rd,[sp+#offset5]	Rd = [sp + ZeroExtend(#offset5, "0")] 将 (sp + ZeroExtend (#offset5,"0")) 指向的 16 位值高位 0 扩展成 32 位保存到 Rd 中, sp 不变, 地址半字对齐		2	Rd ∈ [R0...R7] Offset5 ∈ [0...31]
4	LD.W Rd,[sp+#offset8]	Rd = [sp + ZeroExtend(#offset8,"00")] 将 (sp + ZeroExtend (#offset8,"00")) 指向的 32 位值保存到 Rd 中, sp 不变, 地址字对齐		2	Rd ∈ [R0...R7] offset8 ∈ [0...255]
5	LDS.B Rd,[Rs]	Rd = SignExtend([Rs], 32) 将 Rs 指向地址的 8 位值 符号拓展 成 32 位后保存到 Rd 中, 地址字节对齐		2	Rd ∈ [R0...R15] Rs ∈ [R0...R15]
6	LDS.H Rd,[Rs]	Rd = SignExtend([Rs], 32) 将 Rs 指向地址的 16 位值 符号拓展 成 32 位后保存到 Rd 中, 地址半字对齐		2	Rd ∈ [R0...R15] Rs ∈ [R0...R15]
7	LD.B Rd,[Rs]	Rd = ZeroExtend([Rs], 32) 将 Rs 指向地址的低 8 位值高位 0 拓展成 32 位后保存到 Rd 中, 地址字节对齐		2	Rd ∈ [R0...R15] Rs ∈ [R0...R15]
8	LD.H Rd,[Rs]	Rd = ZeroExtend([Rs], 32) 将 Rs 指向地址的 16 位值高位 0 拓展成 32 位后保存到 Rd 中, 地址半字对齐		2	Rd ∈ [R0...R15] Rs ∈ [R0...R15]
9	LD.W Rd,[Rs]	Rd = [Rs] 将 Rs 指向地址的 32 位值保存到 Rd 中, 地址字对齐		2	Rd ∈ [R0...R15] Rs ∈ [R0...R15]
10	LD.B Rd,[Rs++]	Rd = ZeroExtend([Rs], 32) 将 Rs 指向地址的低 8 位值高位 0 拓展成 32 位后保存到 Rd 中, 地址字节对齐, Rs 加1		2	Rd ∈ [R0...R15] Rs ∈ [R0...R15]
11	LD.H Rd,[Rs++]	Rd = ZeroExtend([Rs], 32) 将 Rs 指向地址的 16 位值高位 0 拓展成 32 位后保存到 Rd 中, 地址半字对齐。Rs 加2		2	Rd ∈ [R0...R15] Rs ∈ [R0...R15]
12	LD.W Rd,[Rs++]	Rd = [Rs] 将 Rs 指向地址的 32 位值保存到 Rd 中, 地址字对齐。Rs加4		2	Rd ∈ [R0...R15] Rs ∈ [R0...R15]
13	LD.B Rd,[Rs+#offset5]	将Rs+ZeroExtend(offset5)指向的8 位内容零扩展成32 位, 再传送到Rd。Rs 不变, 地址字节对齐		2	Rd ∈ [R0...R7] Rs ∈ [R0...R7] offset5 ∈ [0...31]
14	LD.H Rd,[Rs+#offset5]	将Rs+ZeroExtend(offset5,"0")指向的16 位内容零扩展成32 位, 再传送到Rd。Rs 不变, 地址半字对齐		2	Rd ∈ [R0...R7] Rs ∈ [R0...R7] offset5 ∈ [0...31]
15	LD.W Rd,[Rs+#offset5]	将 Rs+ZeroExtend(offset5,"00")指向的 32 位内容传送到 Rd。Rs 不变, 地址字对齐		2	Rd ∈ [R0...R7] Rs ∈ [R0...R7] offset5 ∈ [0...31]
16	LD.B Rd,[Rt+Rs]	将[Rt+Rs]指向的 8 位内容零扩展成 32, 再位		2	Rd ∈ [R0...R7]

		传送到 Rd。Rt + Rs 指向地址字节对齐			Rt ∈ [R0...R7] Rs ∈ [R0...R7]
17	LD.H Rd,[Rt+Rs]	将[Rt+Rs]指向的16 位内容零扩展成32 位，再传送到Rd。Rt + Rs 指向地址半字对齐		2	Rd ∈ [R0...R7] Rt ∈ [R0...R7] Rs ∈ [R0...R7]
18	LD.W Rd,[Rt+Rs]	将[Rt+Rs]指向的32 位内容传送到Rd。Rt + Rs 指向地址字对齐		2	Rd ∈ [R0...R7] Rt ∈ [R0...R7] Rs ∈ [R0...R7]
19	LDP.B Rd,[Rs]<<#imm2	temp = ZeroExtend([Rs], 32) Rd = temp << imm2 将 Rs 指向地址的 8 位值高位 0 拓展成 32 位，左移 imm2 位，低位补 0，然后保存到 Rd 中。Rs 字节地址对齐		2	Rd ∈ [R0...R7] Rs ∈ [R0...R7] imm ∈ [0...3]
20	LDP.H Rd,[Rs]<<#imm2	temp = ZeroExtend([Rs], 32) Rd = temp << imm2 将 Rs 指向地址的 16 位值零拓展成 32 位，左移 imm2 位，低位补 0，然后保存到 Rd 中。Rs 半字地址对齐		2	Rd ∈ [R0...R7] Rs ∈ [R0...R7] imm ∈ [0...3]
21	LDP.W Rd,[Rs]<<#imm2	temp = [Rs] Rd = temp << imm2 将 Rs 指向地址的 32 位值左移 imm2 位，低位补 0，然后保存到 Rd 中。Rs 字地址对齐		2	Rd ∈ [R0...R7] Rs ∈ [R0...R7] imm ∈ [0...3]
22	ST.B [sp+#offset5], Rs	将Rs 的低8 位内容传送到 sp+ZeroExtend(offset5)对应地址，其他不变		2	Rs ∈ [R0...R7] offset5 ∈ [0...31]
23	ST.H [sp+#offset5], Rs,	将 Rs 的低 16 位内容传送到 sp+ZeroExtend(offset5,"0")对应地址，其他地址不变。sp 不变，地址半字对齐		2	Rs ∈ [R0...R7] offset8 ∈ [0...255]
24	ST.W [sp+#offset8],Rs	将 Rs 的 32 位内容传送到 sp+ZeroExtend(offset8,"00")对应地址。sp 不变，地址字对齐		2	Rs ∈ [R0...R7] offset8 ∈ [0...255]
25	ST.B [Rt],Rs	将 Rs 的低 8 位内容传送到[Rt]对应地址，地址为字节对齐，其他字节内容不变。		2	Rt ∈ [R0...R15] Rs ∈ [R0...R15]
26	ST.H [Rt],Rs	将 Rs 的低 16 位内容传送到[Rt]对应地址，其他地址不变，地址半字对齐		2	Rt ∈ [R0...R15] Rs ∈ [R0...R15]
27	ST.W [Rt],Rs	将 Rs 的 32 位内容传送到[Rt]对应地址。地址字对齐		2	Rt ∈ [R0...R15] Rs ∈ [R0...R15]
28	ST.B [Rt++],Rs	将Rs 的低8 位内容传送到[Rt]对应地址，其他地址不变，Rt 加1		2	Rt ∈ [R0...R15] Rs ∈ [R0...R15]
29	ST.H [Rt++],Rs	将 Rs 的低 16 位内容传送到[Rt]对应地址，其他地址不变，地址半字对齐，Rt 加 2		2	Rt ∈ [R0...R15] Rs ∈ [R0...R15]
30	ST.W [Rt++],Rs	将 Rs 的 32 位内容传送到[Rt]对应地址，地址字对齐，Rt 加 4		2	Rt ∈ [R0...R15] Rs ∈ [R0...R15]
31	ST.B [Rt+#offset5],Rs	将 Rs 的低 8 位内容传送到 Ra+ZeroExtend(offset5)对应地址，地址字节对齐，其他地址不变		2	Rt ∈ [R0...R15] Rs ∈ [R0...R15] offset5 ∈ [0...31]
32	ST.H [Ra+#offset5],Rs	将 Rs 的低 16 位内容传送到 Ra+ZeroExtend(offset5,"0")对应地址，地址半字对齐，其他地址不变		2	Rt ∈ [R0...R15] Rs ∈ [R0...R15] offset5 ∈ [0...31]
33	ST.W [Ra+#offset5],Rs	将 Rs 的 32 位内容传送到 Ra+ZeroExtend(offset5,"00")对应地址，地址		2	Rt ∈ [R0...R15] Rs ∈ [R0...R15] offset5 ∈ [0...31]

		字对齐			
34	ST.B [Rt+Ra],Rs	将 Rs 的低 8 位内容传送到[Rt+Ra]对应地址，地址字节对齐，其他地址不变		2	Rt ∈ [R0...R7] Ra ∈ [R0...R7] Rs ∈ [R0...R7]
35	ST.H [Rt+Ra],Rs	将 Rs 的低 16 位内容传送到[Rt+Ra]对应地址，地址半字对齐，其他地址不变		2	Rt ∈ [R0...R7] Ra ∈ [R0...R7] Rs ∈ [R0...R7]
36	ST.W [Rt+Ra],Rs	将 Rs 的 32 位内容传送到[Rt+Ra]对应地址，地址字对齐		2	Rt ∈ [R0...R7] Ra ∈ [R0...R7] Rs ∈ [R0...R7]
37	STP.B [Rt],Rs <<#imm2	将 Rs 左移 imm2 位(左移范围为 0 到 3)的低 8 位内容传送到[Rs]对应地址，地址字节对齐。其他地址不变		2	Rt ∈ [R0...R7] Rs ∈ [R0...R7] imm2 ∈ [0...3]
38	STP.H [Rt],Rs <<#imm2	将 Rs 左移 imm2 位(左移范围为 0 到 3)的低 16 位内容传送到[Rs]对应地址，地址半字对齐，其他地址不变		2	Rt ∈ [R0...R7] Rs ∈ [R0...R7] imm2 ∈ [0...3]
39	STP.W [Rt],Rs <<#imm2	将 Rs 左移 imm2 位(左移范围为 0 到 3)的 32 位内容传送到[Rs]对应地址，地址字对齐		2	Rt ∈ [R0...R7] Rs ∈ [R0...R7] imm2 ∈ [0...3]

❖ 数学指令 19 条

序号	汇编语法	说明	影响标志位	周期数	限制
1	ADD Rd,#imm7	Rd = Rd + ZeroExtend(imm7)。将 imm7 高位 0 扩展成 32 位并与 Rd 相加，结果保存在 Rd 中。	N/Z/C/V	1	Rd ∈ [R0...R15] imm7 ∈ [0...127]
2	ADD Rd,Rs	Rd = Rd + Rs。将 Rd 和 Rs 相加，结果保存在 Rd 中。	N/Z/C/V	1	Rd ∈ [R0...R15] Rs ∈ [R0...R15]
3	ADDC Rd,Rs	Rd = Rd + Rs + C。将 Rd、Rs 和进位 C 相加，结果保存在 Rd 中。	N/Z/C/V	1	Rd ∈ [R0...R15] Rs ∈ [R0...R15]
4	ADD Rd,Rs,#imm3	Rd = Rs + ZeroExtend(imm3)。将 imm3 高位 0 扩展成 32 位并与 Rs 相加，结果保存在 Rd 中。	N/Z/C/V	1	Rd ∈ [R0...R7] Rs ∈ [R0...R7] imm3 ∈ [0...7]
5	ADD Rd,Rt,Rs	Rd = Rt + Rs。将 Rt 和 Rs 相加，结果保存在 Rd 中。	N/Z/C/V	1	Rd ∈ [R0...R7] Rt ∈ [R0...R7] Rs ∈ [R0...R7]
6	ADDC Rd,Rt,Rs	Rd = Rt + Rs + C。将 Rt、Rs 和 C 相加，结果保存在 Rd 中。	N/Z/C/V	1	Rd ∈ [R0...R7] Rt ∈ [R0...R7] Rs ∈ [R0...R7]
7	DIVS Rd,Rt,Rs	Rd = Rt / Rs。有符号除法，Rd 为结果的低 32 位，不影响标志位。当除数为 0 时，可引发一个算术错误中断。若判断结果符合应追加一条 NOP 后处理		2-12	Rd ∈ [R0...R7] Rt ∈ [R0...R7] Rs ∈ [R0...R7]
8	DIVU Rd,Rt,Rs	Rd = Rt / Rs。无符号除法，Rd 为结果的低 32 位，不影响标志位。当除数为 0 时，可引发一个算术错误中断。		2-12	Rd ∈ [R0...R7] Rt ∈ [R0...R7] Rs ∈ [R0...R7]
9	MULS Rd,Rt,Rs	Rd = Rt * Rs。有符号乘法，Rd 为结果的低 32 位，结果影响标志位。	N/Z	1	Rd ∈ [R0...R7] Rt ∈ [R0...R7] Rs ∈ [R0...R7]

10	SUB Rd,#imm7	$Rd = Rd + \text{NOT}(\text{ZeroExtend}(\#imm7)) + 1$ 同 $Rd = Rd - \text{ZeroExtend}(\#imm7, 32)$ 。	N/Z/C/V	1	$Rd \in [R0...R15]$ $imm7 \in [0...127]$
11	SUB Rd, Rs	$Rd = Rd - Rs$	N/Z/C/V	1	$Rd \in [R0...R15]$ $Rs \in [R0...R15]$
12	SUBC Rd, Rs	$Rd = Rd + \text{NOT}(Rs) + C$	N/Z/C/V	1	$Rd \in [R0...R15]$ $Rs \in [R0...R15]$
13	SUB Rd, Rs, #imm3	$Rd = Rs + \text{NOT}(\text{ZeroExtend}(\#imm3)) + 1$ 同 $Rd = Rs - \text{ZeroExtend}(\#imm3, 32)$	N/Z/C/V	1	$Rd \in [R0...R7]$ $Rs \in [R0...R7]$ $imm3 \in [0...7]$
14	SUB Rd, Rt, Rs	$Rd = Rt - Rs$	N/Z/C/V	1	$Rd \in [R0...R7]$ $Rt \in [R0...R7]$ $Rs \in [R0...R7]$
15	SUBC Rd, Rt, Rs	$Rd = Rt + \text{NOT}(Rs) + C$	N/Z/C/V	1	$Rd \in [R0...R7]$ $Rt \in [R0...R7]$ $Rs \in [R0...R7]$
16	SXT.B Rd, Rs	$Rd = \text{SignExtend}(Rs<7:0>, 32)$ 。将 Rs 的低 8 位内容符号位扩展成 32 位，结果保存到 Rd 中		1	$Rd \in [R0...R15]$ $Rs \in [R0...R15]$
17	SXT.H Rd, Rs	$Rd = \text{SignExtend}(Rs<15:0>, 32)$ 。将 Rs 的低 16 位内容符号扩展成 32 位，结果保存到 Rd 中。		1	$Rd \in [R0...R15]$ $Rs \in [R0...R15]$
18	ZXT.B Rd, Rs	$Rd = \text{ZeroExtend}(Rs<7:0>, 32)$ 。将 Rs 的低 8 位内容 0 扩展成 32 位，结果保存到 Rd 中		1	$Rd \in [R0...R15]$ $Rs \in [R0...R15]$
19	ZXT.H Rd, Rs	$Rd = \text{ZeroExtend}(Rs<15:0>, 32)$ 。将 Rs 的低 16 位内容 0 扩展成 32 位，结果保存到 Rd 中。		1	$Rd \in [R0...R15]$ $Rs \in [R0...R15]$

❖ 逻辑指令 7 条

序号	汇编语法	说明	影响标志位	周期数	限制
1	ANL Rd, Rs	$Rd = Rd \& Rs$ Rd 和 Rs 按位逻辑与，结果保存在 Rd 中。	N/Z	1	$Rd \in [R0...R15]$ $Rs \in [R0...R15]$
2	ANL Rd, Rt, Rs	$Rd = Rt \& Rs$ 将 Rt 和 Rs 按位逻辑与，结果保存在 Rd 中。	N/Z	1	$Rd \in [R0...R7]$ $Rt \in [R0...R7]$ $Rs \in [R0...R7]$
3	NOT Rd, Rs	$Rd = \sim Rs$ Rs 按位取反，结果保存在 Rd 中	N/Z	1	$Rd \in [R0...R15]$ $Rs \in [R0...R15]$
4	ORL Rd, Rs	$Rd = Rd Rs$ ，Rd 和 Rs 按位逻辑或，结果保存在 Rd 中	N/Z	1	$Rd \in [R0...R15]$ $Rs \in [R0...R15]$
5	ORL Rd, Rt, Rs	$Rd = Rt Rs$ 将 Rt 和 Rs 按位逻辑或，结果保存在 Rd 中	N/Z	1	$Rd \in [R0...R7]$ $Rt \in [R0...R7]$ $Rs \in [R0...R7]$
6	XRL Rd, Rs	$Rd = Rd \wedge Rs$ Rd 和 Rs 按位逻辑异或，结果保存在 Rd 中。	N/Z	1	$Rd \in [R0...R15]$ $Rs \in [R0...R15]$
7	XRL Rd, Rt, Rs	$Rd = Rt \wedge Rs$ 将 Rt 和 Rs 按位逻辑异或，结果保存在 Rd 中。	N/Z	1	$Rd \in [R0...R7]$ $Rt \in [R0...R7]$ $Rs \in [R0...R7]$

❖ 位操作指令 9 条

序号	汇编语法	说明	影响标志位	周期数	限制
----	------	----	-------	-----	----

1	CLR Rd,#bit5	将Rd 的bit5指定位 (0-31) 清零，其他位不变	N/Z	1	Rd ∈ [R0...R15] Bit5 ∈ [0...31]
2	CLR [Rs],#bit5	将[Rs]的bit5指定位清零，其他位不变	N/Z	2	Rs ∈ [R0...R15] bit5 ∈ [0...31]
3	CLR PSW,#bit5	将 PSW 寄存器中 bit5 指定的位清零	N/Z/C/V	1	bit5 ∈ [0...31]
4	DSI	关闭中断总使能，AIE = 0		1	
5	ENI	使能中断总使能，AIE = 1		1	
6	REV Rd, Rs	Rd<31:0> = RS<0:31> 位反序操作，将 Rs 中的值的第 n 位与第 31-n 位交换		1	Rd ∈ [R0...R15] Rs ∈ [R0...R15]
7	SET Rd, #bit5	将Rd 的bit5位 (0-31) 置1，其他位不变	N/Z	1	Rd ∈ [R0...R15] bit5 ∈ [0...31]
8	SET [Rt], #bit5	将 Rt 中指向地址内容的第 bit5 位置 1，并将结果保存到 Rt 指向地址，Rt 字对齐	N/Z	2	Rt ∈ [R0...R15] bit5 ∈ [0...31]
9	SET PSW, #bit5	将 PSW 寄存器中 bit5 指定的位置 1	N/Z/C/V	1	bit5 ∈ [0...31]

❖ 循环/移位指令 11 条

序号	汇编语法	说明	影响标志位	周期数	限制
1	ASR Rd,#imm5	Rd = Rd 算术右移 imm5 位 (0-31)。高位补符号位，低位移出到标志位 C。移位 0 时保持不变	N/Z/C	1	Rd ∈ [R0...R15] imm5 ∈ [0...31]
2	ASR Rd,Rs	Rd = Rd 算术右移Rs<7:0>位。高位补符号位，低位移出到标志位 C。移位0时保持不变	N/Z/C	1	Rd ∈ [R0...R15] Rs ∈ [R0...R15]
3	ASR Rd,Rt,Rs	Rd = Rt 算术右移Rs<7:0>位。高位补符号位，低位移出到标志位 C。移位0时保持不变	N/Z/C	1	Rd ∈ [R0...R7] Rt ∈ [R0...R7] Rs ∈ [R0...R7]
4	LSL Rd,#imm5	Rd = Rd 逻辑左移 imm5 位。低位补 0，高位移出到 C 标志位中，当移位为 0 时 C 保持不变	N/Z/C	1	Rd ∈ [R0...R15] imm5 ∈ [0...31]
5	LSL Rd,Rs	Rd = Rd 逻辑左移Rs<7:0>位。低位补 0，高位移出到 C 标志位中，当移位为 0 时 C 保持不变	N/Z/C	1	Rd ∈ [R0...R15] Rs ∈ [R0...R15]
6	LSL Rd,Rt,Rs	Rd = Rt 逻辑左移Rs<7:0>。低位补 0，高位移出到 C 标志位中，当移位为 0 时 C 保持不变	N/Z/C	1	Rd ∈ [R0...R7] Rt ∈ [R0...R7] Rs ∈ [R0...R7]
7	LSR Rd,#imm5	Rd = Rd 逻辑右移 imm5 位 (0-31)。高位补 0，低位移出到 C 标志位中。当移位为 0 时 C 保持不变	N/Z/C	1	Rd ∈ [R0...R15] imm5 ∈ [0...31]
8	LSR Rd,Rs	Rd = Rd 逻辑右移Rs<7:0>位。高位补 0，低位移出到 C 标志位中。当移位为 0 时 C 保持不变	N/Z/C	1	Rd ∈ [R0...R15] Rs ∈ [R0...R15]
9	LSR Rd,Rt,Rs	Rd = Rt 逻辑右移Rs<7:0>位。高位补 0，低位移出到 C 标志位中。当移位为 0 时 C 保持不变	N/Z/C	1	Rd ∈ [R0...R7] Rt ∈ [R0...R7] Rs ∈ [R0...R7]
10	ROR Rd, #imm5	Rd = Rd 循环右移 imm5 位 (0-31)。进位不	N/Z/C	1	Rd ∈ [R0...R15] imm5 ∈ [0...31]

		进入循环移位, 移出位更新进位 C, 当移 0 位时 C 不变			
11	ROR Rd, Rs	Rd = Rd 循环右移Rs<4:0>位。进位不进入循环移位, 移出位更新进位C, 当移 0 位时 C 不变	N/Z/C	1	Rd ∈ [R0...R15] Rs ∈ [R0...R15]

❖ 比较/跳过指令 8 条

序号	汇编语法	说明	影响标志位	周期数	限制
1	CMN Rt, Rs	比较Rt + Rs	N/Z/C/V	1	Rt ∈ [R0...R15] Rs ∈ [R0...R15]
2	CMP Rt, #imm7	比较Rt - ZeroExtend(imm7)	N/Z/C/V	1	Rt ∈ [R0...R15] imm7 ∈ [0...127]
3	CMP Rt, Rs	比较Rt - Rs	N/Z/C/V	1	Rt ∈ [R0...R15] Rs ∈ [R0...R15]
4	JB Rt, #bit5	Rt 的bit5位 (0-31) 为1 跳过, 否则顺序执行, 跳转地址+2 (后接16位指令) 或跳转地址+4 (后接32位指令)		2/4	Rt ∈ [R0...R7] bit5 ∈ [0...31]
5	JB [Rt], #bit5	[Rt]的bit5位 (0-31) 为1 跳过, 否则顺序执行。跳转地址+2 (后接16位指令) 或跳转地址+4 (后接32位指令)		3/5	Rt ∈ [R0...R7] bit5 ∈ [0...31]
6	JNB Rt, #bit5	Rt 的bit5位 (0-31) 为0 跳过, 否则顺序执行, 跳转地址+2 (后接16位指令) 或跳转地址+4 (后接32位指令)		2/4	Rt ∈ [R0...R7] bit5 ∈ [0...31]
7	JNB [Rt], #bit5	[Rt]的bit5位 (0-31) 为0 跳过, 否则顺序执行, 跳转地址+2 (后接16位指令) 或跳转地址+4 (后接32位指令)		3/5	Rt ∈ [R0...R7] bit5 ∈ [0...31]
8	TST Rt, Rs	Rt & Rs, 将 Rt 与 Rs 按位逻辑与, 结果影响标志位, 但不保留结果	N/Z	1	Rt ∈ [R0...R15] Rs ∈ [R0...R15]

注：如果没有跳过时条件跳过指令执行周期为2，跳过时执行周期为4

❖ 程序流指令指令 20 条

序号	汇编语法	说明	影响标志位	周期数	
1	JZ label8	Z=1 跳转, 否则顺序执行		1/3	label8 ∈ [-128...127]
2	JNZ label8	Z=0 跳转, 否则顺序执行		1/3	label8 ∈ [-128...127]
3	JC label8	C=1 跳转, 否则顺序执行		1/3	label8 ∈ [-128...127]
4	JNC label8	C=0 跳转, 否则顺序执行		1/3	label8 ∈ [-128...127]
5	JMI label8	N=1 跳转, 否则顺序执行		1/3	label8 ∈ [-128...127]
6	JPL label8	N=0 跳转, 否则顺序执行		1/3	label8 ∈ [-128...127]
7	JVS label8	V=1 跳转, 否则顺序执行		1/3	label8 ∈ [-128...127]
8	JVC label8	V=0 跳转, 否则顺序执行		1/3	label8 ∈ [-128...127]
9	JHI label8	C=1 且Z=0 跳转, 否则顺序执行		1/3	label8 ∈ [-128...127]

10	JLS label8	C=0 或Z=1 跳转，否则顺序执行		1/3	label8 ∈ [-128...127]
11	JGE label8	N=V 跳转，否则顺序执行		1/3	label8 ∈ [-128...127]
12	JLT label8	N!=V 跳转，否则顺序执行		1/3	label8 ∈ [-128...127]
13	JGT label8	Z=0 且N=V 跳转，否则顺序执行		1/3	label8 ∈ [-128...127]
14	JLE label8	Z=1 或N!=V 跳转，否则顺序执行		1/3	label8 ∈ [-128...127]
15	JMP label24	PC = PC + SignExtend(label24,"0",32) 32 位指令，相对跳转,PC=PC+SignExtend(label24,"0",32)。		3	label24 ∈ [-134217728...134217727]
16	JMP Rs	PC = Rs 跳转到 Rs 所指的地址,不保留跳转前地址。		3	Rt ∈ [R0...R15]
17	LJMP Rs	LR = PC+2 PC = {[Rs]<31:1>, 0} 将 PC+2 保存到链接寄存器 LR, 然后跳转到 Rs 所指的固定 bit0=0 地址。		3	Rs ∈ [R0...R15]
18	LJMP label8	LR = PC+2 PC = PC + SignExtend(label8, "0", 32) 将 PC+2 保存到链接寄存器 LR, 然后跳转到新的 PC 所指的地址。		3	label8 ∈ [-128...127]
19	LJMP label21	LR = PC<31:1>+4 PC = PC + SignExtend(label21, "0", 32) 将 PC+4 保存到链接寄存器 LR, 然后跳转到新的 PC 所指的地址。		3	label21 ∈ [-1048576...1048575]
20	SJMP label10	PC = PC + SignExtend{label10,"0"} 相对跳转, PC + SignExtend{label10,"0"}		3	label10 ∈ [-512...511]

注：汇编器支持以上指令直接输入标签，并根据标签计算实际指令编码值。跳转寻址范围应 PC + SignExtend{immXX,"0"}。周期：条件不匹配 1 个周期，匹配 3 个周期)

❖ 堆栈/出栈指令 4 条

序号	汇编语法	说明	影响标志位	周期数	限制
1	POP {list}	{list}为{PC/R12/R11/R10/R9/R8/R7/R6}寄存器，将堆栈中的值按顺序压出到{list}中值为 1 对应的寄存器 $xSP = xSP + 4n$ (n 为 list 中 1 的个数) 示例写法 POP {R6, R7} POP {R6-R8} POP {R6-R7, R9, R10} POP {R6-R12} POP {R6-R7, R9, PC}		n+1/n+4	list = {PC,R12,R11,R10,R9,R8,R7,R6}(可取任意个数) $xSP \in [MSP, PSP]$
2	POP Rd	将堆栈指针所指的内容值压出到 Rd, $xSP=xSP+4$ 支持文法 POP LR 等效 R13		2	Rd ∈ [R0...R15] $xSP \in [MSP, PSP]$
3	PUSH {list}	{list}为{LR/R12/R11/R10/R9/R8/R7/R6}寄存器，将列表中 1 的对应寄存器压入堆栈。SP 通过 SPSEL(SYS_MCTL<16>)来选择。 $xSP = xSP - 4n$ (n 为 list 中 1 的个数) 示例写法 PUSH {R6, R7} PUSH {R6-R8} PUSH {R6-R7, R9, R12} PUSH {R6-R12, LR}。		n+1	list = {LR,R12,R11,R10,R9,R8,R7,R6}(可取任意个数) $xSP \in [MSP, PSP]$
4	PUSH Rs	将 Rs 内容压入堆栈。SP 通过 SPSEL(SYS_MCTL<16>)来选择。 $xSP = xSP - 4$		2	Rs ∈ [R0...R15] $xSP \in [MSP, PSP]$

支持文法 PUSH LR 等效 R13

注：n 为list 列表中的1 的个数

n+4 为list 列表中包含PC(R15)的情况

注2：函数调用推荐使用 PUSH LR保护返回值，并返回时POP LR；JMP LR退出函数。

注3：中断自动非代码压栈XPSR, PC, LR, R4-R0；并根据LR值自动非代码出中断保护值，如
R0-R4 LR PC XPSR

❖ 控制指令 6 条

序号	汇编语法	说明	影响标志位	周期数	限制
1	BREAK	设置断点（模拟调试方案使用）		1	
2	NOP	空操作		1	
3	RESET	复位内核系统 (DEBUG模块除外)		1	
4	SLEEP	休眠		1	
5	SVC 或 SVC #imm8	触发SVC 中断, 默认值0. 触发一次 SVC 中断, 如果 SVC 优先级高于当前优先级则进入 SVC 中断处理, 若低于当前优先级, 则引发一个 HardFault 中断。imm8 用于记录一个 8 位立即数, 指令执行忽略立即数。		1/2	imm8∈[0...255]
6	SYNC	流水线同步		2	

2.5 RAM 函数

若将函数定位到 RAM 中，以提高性能或特殊功能实现。工具链接脚本中默认已将段.indata 存储到 RAM 中并设定在起始位置，用户只需将代码定义至.indata 段中即可。程序启动代码“**startup 函数**”会将指定为 RAM 函数的函数指定复制到 RAM 中。(注：在项目的 config 文件下的 startup 文件中定义该函数，并在 vector 文件中引导实现函数先调用再转向 main 函数)。

汇编代码中“.section .indata”语句用于分配后续段代码至.indata 段，见嵌汇编部分介绍。

注意：由于 FLASH 与 RAM 之间的距离较远，所以两者间的跳转需要使用绝对跳转指令。即使用 LJMP Rx 的指令。

3 中断函数

3.1 简介

中断处理用来使软件操作与实时发生的事件同步。当发生中断时，软件的正常执行流程被打断，调用专门的函数来处理事件。当中断处理结束时，恢复先前的现场信息并继续正常执行流程。

KF32 器件支持多个内部和外部中断源。另外，允许高优先级中断中断任何正在处理的低优先级中断。

KF32 编译器完全支持在 C 或汇编代码中进行中断处理。

3.2 中断处理函数

3.2.1 中断函数现场保护

中断处理函数用于实现现场保护和恢复，以确保从中断返回时，程序现场恢复进入中断前的状态。

响应中断时，KF32 硬件约定将 R0、R1、R2、R3、R4、R13(LR)、返回地址(当前 PC 值)和程序状态寄存器(xPSR)压入当前激活的堆栈空间中，因此不需要主动保存它们。而其他被中断代码使用到的寄存器会被编译器进行入栈保护。

3.2.2 中断处理程序

默认下，KF32 编译器已将中断向量表转为中断向量配置文件(vector.c 或 vector.asm)。可根据喜好修改文件中对应位置的符号为函数名，或修改对应符号为自定义中断处理函数名。示例如下。

如 T1 中断的入口地址被配置为“.long _T1_exception”，则其中断处理程序写为：

```
.export _T1_exception
_T1_exception:
.....
```

或用户修改中断向量配置文件将_T1_exception 修改为自定义函数名。

KF32 汇编处理函数必须遵守函数地址必须与中断向量一致，即函数名称必须与 vector.asm 文件中指定位置的向量名称一致；使用的中断必须建立该中断函数，否则芯片将运行错误。

3.2.3 中断向量配置

KF32 系列支持多个内核中断和多个外设中断，256 个向量的中断向量表参见 KF32 芯片数据手册。默认情况下使用的中断向量表的起始单元位于存储器的 0x0000_0000 地址处。当响应中断时，处理器自动从向量表中的相应位置加载中断向量入口地址。所以中断处理函数的地址必须为中断向量中对应的地址。

中断向量配置文件中的代码将被链接器分配到芯片起始地址 0x0 处，其中每一中断类型分别对应各自地址。若用户程序未使用或未全部使用中断处理，则可修改中断向量配置文件，以节省内存空间。

中断向量配置文件的修改需遵循以下两点：

- 1、0x0 地址和 0x4 地址的向量不可删除，即初时 SP 和复位向量不可删除；
- 2、删除向量的操作需从后向前开始，不可跳过，否则无法正确进入中断服务程序；**【如果应用中重映射中断向量入库，不能删除，必须满 256 地址空间占用，否则空闲地址存在的代码因向量表切换会获取错误的内容并执行错乱。】**

中断向量配置文件修改示例说明：

用户程序使用定时器 1 全局中断(T1，向量地址 0x0000_0064)和定时器 5 全局中断(T5，向量地址 0x0000_0074)，而其他中断向量均未使用。则用户可将中断向量配置文件中“.long _T5_exception”和“.weak

_T5_exception”之后的几行信息删除，而之前的中断向量不可修改，包括未使用的中断向量。

4 特殊功能寄存器的操作

由于位域处理的效率较低，建议用户对 SFR 进行 32 位操作，减少位域操作。用户如下需要可自行定义，本章节仅供参考。

4.1 特殊功能寄存器的操作

特定于处理器的头文件是一些包含了在汇编语言中使用的特殊功能寄存器 (Special Function Register, SFR) 的外部声明的文件。依照约定，每个 SFR 都使用数据手册中的相同名称进行命名，如 OSC_CTL1 代表振荡器控制寄存器 1。定义结构以实现寄存器的位声明，如_HFCLKCAL。同时约定前面添加前缀“_”，如果具有多个 bit 位并名结尾是数字，通过“_”间隔每一个 bit 编号。

举例，地址为 0x40000004 的特殊功能寄存器 OSC_CTL1，

特殊功能寄存器 OSC_CTL1 在汇编头文件的声明如下：

```
.EQU  OSC_CTL1, 0x40000004
.EQU  _SCLKOUTDIV2,      31
.EQU  _SCLKOUTDIV1,      30
.EQU  _SCLKOUTDIV0,      29
.EQU  _SCLKOE,           28
.EQU  _FSCM,             16
.EQU  _HFCLKCAL7,        15
.EQU  _HFCLKCAL6,        14
.EQU  _HFCLKCAL5,        13
.EQU  _HFCLKCAL4,        12
.EQU  _HFCLKCAL3,        11
.EQU  _HFCLKCAL2,        10
.EQU  _HFCLKCAL1,        9
.EQU  _HFCLKCAL0,        8
.EQU  _HFCLKTRIM4,        4
.EQU  _HFCLKTRIM3,        3
.EQU  _HFCLKTRIM2,        2
.EQU  _HFCLKTRIM1,        1
.EQU  _HFCLKTRIM0,        0
```

在应用程序中使用 SFR 时，需要执行 2 个步骤。

1. 需包含芯片头文件：

```
.include "KF32XXXX.inc"
```

2. 汇编项目通过地址和位号直接操作。

2.1 如，设置 SCLK 输出时钟 1/32 分频：

```
汇编：LD  R5, #OSC_CTL1
      SET [R5], #_SCLKOUTDIV0
      CLR [R5], #_SCLKOUTDIV1
```

```
SET [R5], #_SCLKOUTDIV2
```

2.2 如，设置 OSC_CTL1 第 30 位为 1:

```
汇编: LD R5, #OSC_CTL1
```

```
SET [R5], #30
```

2.3 如，将 OSC_CTL1 所有位置 1，即寄存器整体赋值:

```
汇编: LD R5, #OSC_CTL1
```

```
MOV R0, #0
```

```
NOT R0, R0
```

```
ST.w [R5], R0
```

4.2 寄存器使用约定

在 C 语言中寄存器使用规则如表，因此可以编写汇编带参数或返回的库，参数或结果在如下对应功能的寄存器中。另外编写汇编库应该适应 C 规则下占用寄存器的压栈出栈保护。

表 6-1 寄存器约定

寄存器名称	用途	跨函数调用时是否保存
R0	gp/传参/返回值	不保存
R1	gp/传参/返回值	不保存
R2 - R4	gp/传参	不保存
R5	gp/scratch	不保存
R6	gp/fp	
R7 - R12	gp	
R13	lr/gp/函数返回地址	
R14	sp/fixed	否
R15	pc/fixed	否
R16 - R17		
R18 - R19		
R20 - R21		
R22 - R23		
R24 - R25		
R26 - R27		
R28 - R29		
R30 - R31		

注：具体寄存器数量根据型号确定，即支持 R0~R15 或支持 R0~R31。

在表 6-1 中，gp 表示该寄存为通用寄存器，可参与任何操作，fixed 表示该寄存器为特殊寄存器，具有固定的用途，寄存器分配时，不参与分配。R6-R13 用于保存生命周期跨函数调用的变量。R6 在特殊情况下可替代实现 FP 指针的功能，如临时栈空间开辟。ACC0 - ACC7 为 8 个由 32 位寄存器组成 64 位寄存器。如 ACC0 对应 R16 和 R17。R13 别名为 LR，R14 别名为 SP，R15 别名为 PC。

如果中断和外部函数同时使用 R6 以上的寄存器时，应添加压栈出栈指令。

若中断逻辑中使用 R5，可在中断顶层函数添加压栈出栈保护 R5。

附录 1：中断向量表实现示例与说明

汇编项目直接根据汇编语言格式进行编写。根据芯片的对于中断偏移位置，需要同时修改 .long 后的名字，和 weak 后面的名字，即 weak 修饰该中断符号有效不参与优化。当未编写对于的中断函数，该偏移位置默认编译结果入口地址为 0。即一旦未编写中断函数却配置使发生该中断，此时入口为初始的 SP 地址，即不存在的存储空间，此时运行将发生错误。

不同系列芯片的向量表可能存在不同的情况，具体根据芯片数据手册或根据 ChipON IDE 建立项目下自动生成的文件文件为准。该内容存放在项目的 config 文件夹下的 vector 文件中，并链接脚本声明在 Flash 的起始地址。如果应用代码使用向量表重映射，应该在其他文件中重新编写向量表，并关联到新的中断处理函数上。

汇编语言格式：

```
.global    _start
.text

_start:
.long    __initial_sp
.long    startup
.long    _NMI_exception
.long    _HardFault_exception
.long    _Soft4_exception
.long    _StackFault_exception
.long    _AriFault_exception
.long    0
.long    _Soft8_exception
.....
.weak    startup
.weak    _NMI_exception
.weak    _HardFault_exception
.weak    _Soft4_exception
.weak    _StackFault_exception
.weak    _AriFault_exception
.weak    _Soft8_exception
.....
.weak    _Soft125_exception
.weak    _Soft126_exception
.weak    _Soft127_exception
.end
```

附录 2：示例变量与函数格式书写

对照 C 代码示例

```
unsigned char var_noinit;

unsigned int var_init=0x17777777;

union {
    unsigned char v_char;
    unsigned short v_short;
    unsigned int v_int;
    unsigned long v_long;
    struct{
        unsigned short L_Short;
        unsigned short H_Short;
    };
}s_abc;

unsigned short v_static_noinit;
unsigned char v_static_value[]={1,2,3,4};

void fun1(void)
{
    asm("NOP");
}

__attribute__((section("indata")))
int fun2(unsigned int addr,unsigned int length){
    return addr+ length;
}
```

汇编文件代码编写示例

```
// 应该经常写.section 归属到变量和函数单元,
// 否则后面的内容将均归属该段,从而段落不清晰或错误归属
// .text .data 为默认非唯一化名字的定义程序段或数据段,等效如.section .text
// 段明使用$并后缀不是必须需要。
//但若不唯一化,即使开启死段优化,任意一个被使用,其他同名均会被分配空间的占用资源
//#####
.export v_static_value // 全局变量对外声明
.section .data$init$v_static_value // 归属 ram 数据对象,初始化默认.data 归属
.align 2 // 根据数据类型控制对齐,1 为 2 字节对齐,2 为 4 字对齐
.type v_static_value, @object // 类型声明
.size v_static_value, 4 // 可选大小修饰
v_static_value: // 名与值
.byte 1 // byte 伪指令定义字节的值 0-255。同类伪指令.word .long
.byte 2
.byte 3
.byte 4
```

```

#####
.export v_static_noinit
.section .bss$comm$v_static_noinit// 归属 ram 数据对象 , 无初始化默认.bss 归属
.align 1
.type v_static_noinit, @object
.size v_static_noinit, 2
v_static_noinit:
.fill 2, 1 // 无初始值的 使用 .fill 的单个大小 1 字节的重复 2 次, 即 2 字节空间
#####
.export s_abc
.export s_abc_v_char
.export s_abc_v_short
.export s_abc_v_int
.export s_abc_v_long
.export s_abc_v_L_Short
.export s_abc_v_H_Short

.section .bss$comm$s_abc
.align 2
.type s_abc, @object
.size s_abc, 4
s_abc:
s_abc_v_char:
s_abc_v_short:
s_abc_v_int:
s_abc_v_long:
s_abc_v_L_Short:
.fill 1, 1
.fill 1, 1
s_abc_v_H_Short:
.fill 1, 1
.fill 1, 1
#####
.export var_init
.section .data$init$var_init
.align 2
.type var_init, @object
.size var_init, 4
var_init:
.long 24606583 // 支持 16 进制格式: 0x1777777
#####
.export var_noinit
.section .bss$comm$var_noinit
.type var_noinit, @object
.size var_noinit, 1
var_noinit:
.fill 1, 1
#####
.section .text$fun1 // 默认.text 的代码属性, $及后缀为唯一段名扩展
.func fun1 // 使能调试的依赖需求
.align 1 //典型最小 16bit 指令的要求 2 字节对齐
.export fun1
.type fun1, @function
fun1: // 标签或函数名
NOP
JMP lr
.size fun1, .-fun1 // 辅助函数段大小信息
.endfunc // 使能调试的依赖需求

```

```

#####
.section .indata$fun2 // 与链接器脚本呼应的定义为 ram 函数

.func fun2           // 使能调试的依赖需求
.align 1
.export fun2
.type fun2, @function
fun2:
    ADD r0, r1, r0
    JMP lr
    .size fun2, .-fun2
.endfunc             // 使能调试的依赖需求
#####
.end                // 文件提前结束，或结束，若不书写，最后行应该存在一个空行

```

引入宏的常规信息简化

举例通过编写宏来声明一个函数的起始和结束，宏提供函数的伪指令描述信息，示例如下：

```

//*****
__FUNCTION_STAET funname
.section .text$funname
.align 1
.func \funname
.export \funname
.type \funname, @function
\funname:
.endm

__FUNCTION_END funname
.size \funname, .-\funname
.endfunc
.endm
//*****
__FUNCTION_STAET main
//*****
LoopMain:
    JMP LoopMain
//*****
__FUNCTION_END main
//*****

```

C 文件使用汇编文件变量与函数

```

extern unsigned char var_noinit;

extern unsigned int var_init;

extern unsigned short v_static_noinit;
extern unsigned char v_static_value[];

extern unsigned char s_abc_v_char;
extern unsigned short s_abc_v_short;
extern unsigned int s_abc_v_int;
extern unsigned long s_abc_v_long;
extern unsigned short s_abc_v_L_Short;

```

```
extern unsigned short    s_abc_v_H_Short;

void fun1(void);

//__attribute__((section(".indata")))
int fun2(unsigned int addr,unsigned int length);

//Main Function
int main(void)
{
    var_noinit++;
    var_init++;
    v_static_noinit++;
    v_static_value[0]++;

    s_abc_v_char++;
    s_abc_v_short++;
    s_abc_v_long++;
    s_abc_v_L_Short++;
    s_abc_v_H_Short++;
    while(1)
    {
        fun1();
        fun2(0,3);
    }
}
```